

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

1	Wat je kunt verwachten	ii
2	Over de bronnen	ii
3	Benodigdheden	ii
4	Dankwoord	iii

1 De eerste stappen

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#

2.1	Keywords: de woordenschat	34
2.2	Variabelen, identifiers 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

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

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 OOP 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 geüpdated in de zomer van 2024, na reeds een grote herziening in 2022.

Net zoals spreektaLEN, 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 programmeurplezier! 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 explicet 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 onderwijservorming 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 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 obscurer 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 spreektaLEN 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 “hoger” 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.

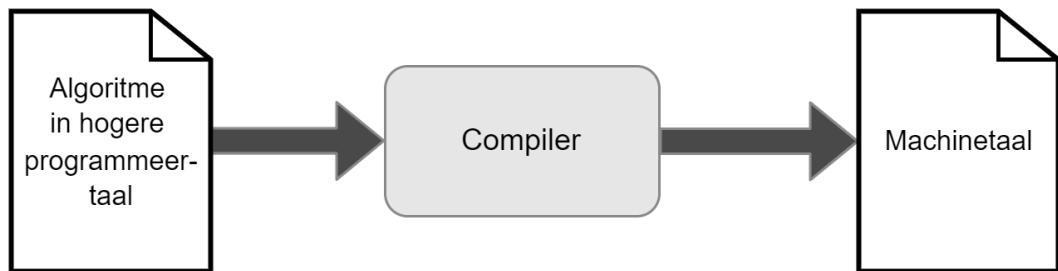
Je zou denken dat hij na 3 programmeertalen wel op z’n lauweren zou rusten, maar zo werkt Anders niet. In 2012 begon hij te werken aan een JavaScript alternatief, wat uiteindelijk het immens populaire TypeScript werd. Dit allemaal om maar te zeggen dat als je één poster in je slaapkamer moet ophangen,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

Tekens er een nieuwe .NET framework werd *released* 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 verwarring maakte. Om dit te vereenvoudigen bestaat sinds 2020 enkel nog .NET gevuld 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 onder andere een debugger, code editor en compiler in.

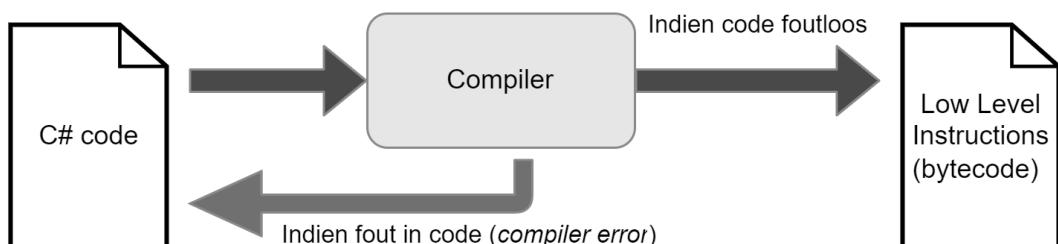
VS is een zogenaamde **IDE (“Integrated Development Environment”)** en is op maat gemaakt om in C# geschreven applicaties te ontwikkelen. Je bent echter verre van verplicht om enkel C# applicaties in VS te ontwikkelen. Je kan gerust VB.NET, TypeScript, Python en andere talen gebruiken. Ook vice versa 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.

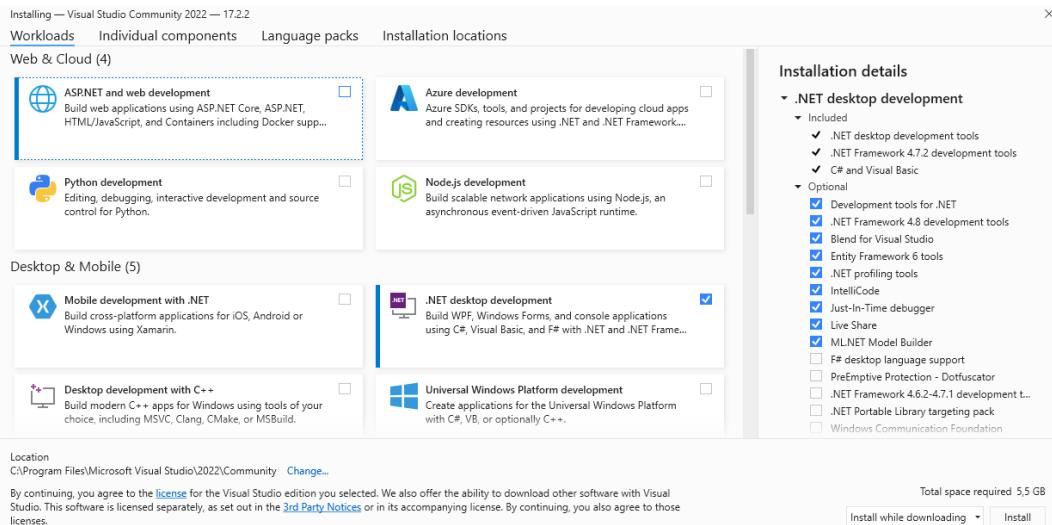


Figuur 1.2: Vereenvoudigd compiler overzicht.

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)

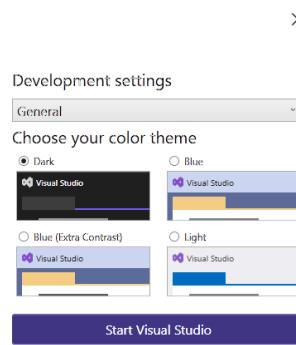
De allereerste keer dat je VS opstart krijg je 2 extra schermen te zien:

- 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.

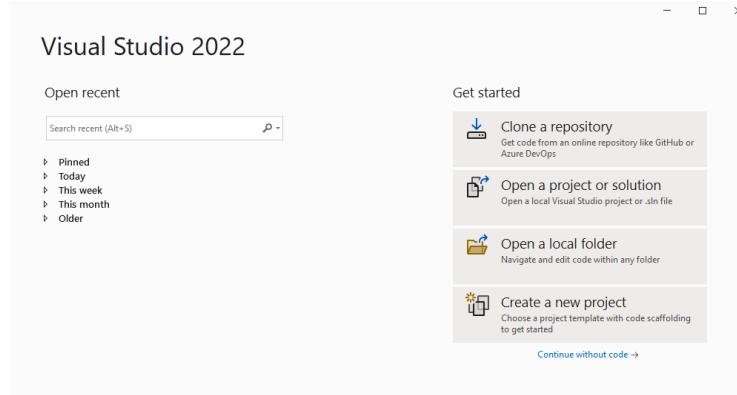
De keuze voor Development Setting kan je naar “Visual C#” veranderen, maar General is even goed (je zal geen verschil merken in eerste instantie). Je kan dit achteraf nog aanpassen in VS via “Tools” in de menubalk, dan “Import and Export Settings” en kiezen voor “Import and Export Settings Wizard”.



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

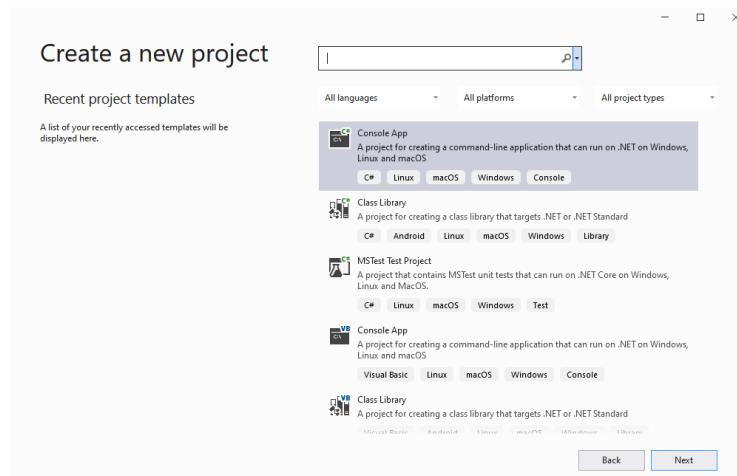
Na het opstarten van VS krijg je het startvenster te zien van waaruit je verschillende dingen kan doen. Van zodra je projecten gaan aanmaken zullen deze in de toekomst ook op dit scherm getoond worden zodat je snel naar een voorgaand project kunt gaan.



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.**



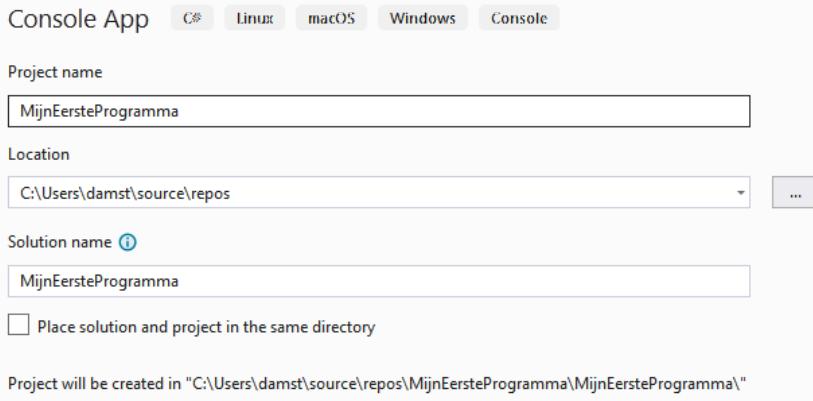
Het “Solution name” tekstveld blijf je af. Hier zal automatisch dezelfde tekst komen als die dat je in het “Project name” tekstveld invult.



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.

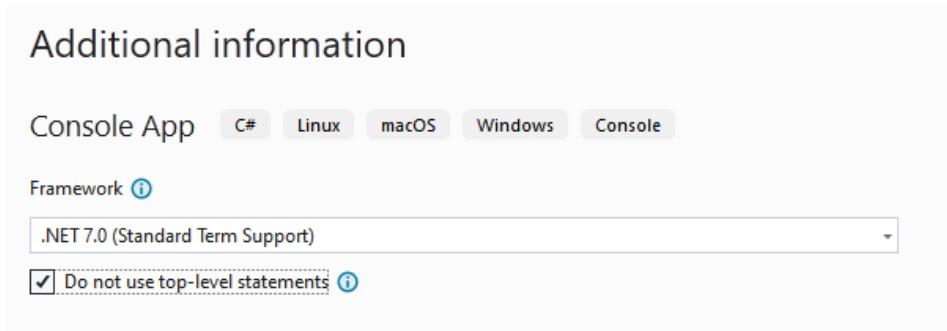
Geef je project de naam “MijnEersteProgramma” en kies een goede locatie. **Ik raad aan om de checkbox “Place solution and project in the same directory” onderaan niet aan te vinken.** In de toekomst zal het nuttig zijn dat je meer dan 1 project per solution zal kunnen hebben. Lig er nog niet van wakker.

Configure your new project



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.



Figuur 1.10: Gebruik alsjeblieft geen top-level statements!

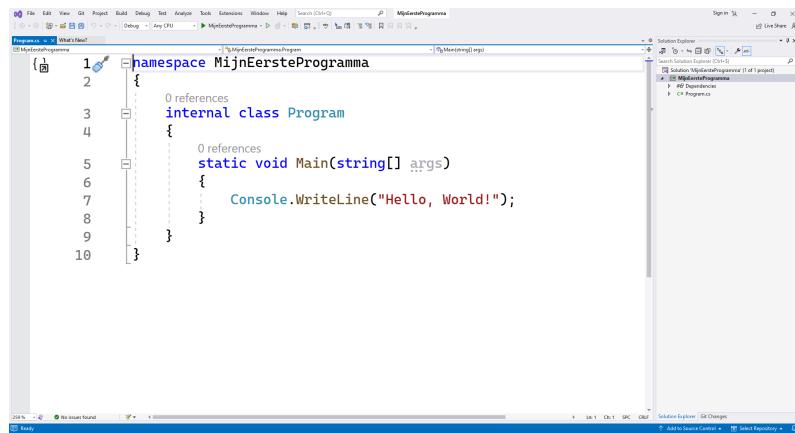
VS heeft nu reeds een aantal bestanden aangemaakt die je nodig hebt om een ‘Console Applicatie’ te maken.

³De auteur van dit boek kan fier melden dat die checkbox er staat mede dankzij zijn gezaag op [git-hub.com/dotnet/docs/issues/2742](https://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 **geen** 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:

```
Microsoft Visual Studio Debug Console
Hello World!
C:\Program Files\dotnet\dotnet.exe (process 9888) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

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/shellscherm 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 foutbericht 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 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 aanroeft 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 lijn⁴7⁴. Dit is het startpunt van ons programma en de uitvoer ervan. Al de zaken ervoor kan je voorlopig keihard nergeren.

Het programma zal alles uitvoeren dat tussen de accolades van het Main-blok staat. Dit blok wordt aangebakend 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 ingetypt)

```
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 Aanhalingssteken of niet?

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

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

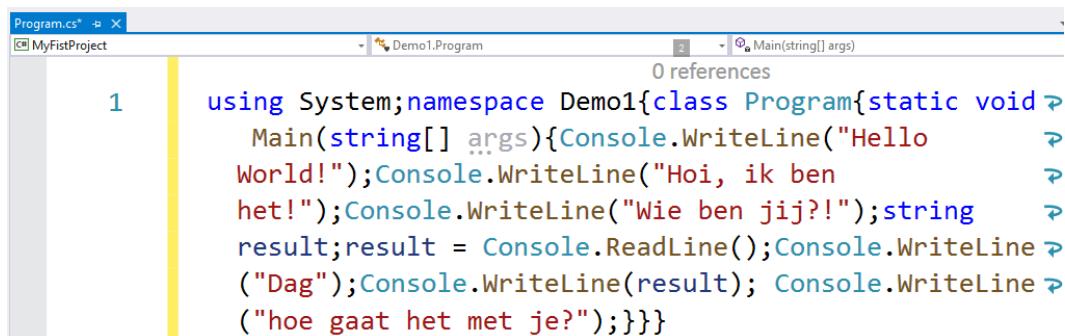
```
1 Console.WriteLine("Dag ");
2 Console.WriteLine();
3 Console.WriteLine(" hoe gaat het met je?");
```

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 niet binnen aanhalingstekens staan**. Zowel spaties, enteren en tabs worden genegeerd. Met andere woorden: je kan het voorgaande programma perfect in één lange lijn code typen, zonder enteren. Dit is echter niet aangeraden want het maakt je code een pak onleesbaarder.



The screenshot shows a Microsoft Visual Studio interface with a code editor window. The file is named 'Program.cs'. The code is written in a single, extremely long line. A vertical yellow bar highlights the first character of the line. The code itself is as follows:

```
1 using System; namespace Demo1 { class Program { static void Main(string[] args) { Console.WriteLine("Hello World!"); Console.WriteLine("Hoi, ik ben het!"); Console.WriteLine("Wie ben jij?"); string result; result = Console.ReadLine(); Console.WriteLine("Dag"); Console.WriteLine(result); Console.WriteLine("hoe gaat het met je?"); } }}
```

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.WriteLine("Dag_");
3 Console.WriteLine(result_);
4 Console.WriteLine("hoe gaat het met je?");
```

En een correct gebruik:

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

1.3.9 Zinnen aan elkaar plakken

We kunnen dit allemaal nog een pak koper 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 koper 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]**

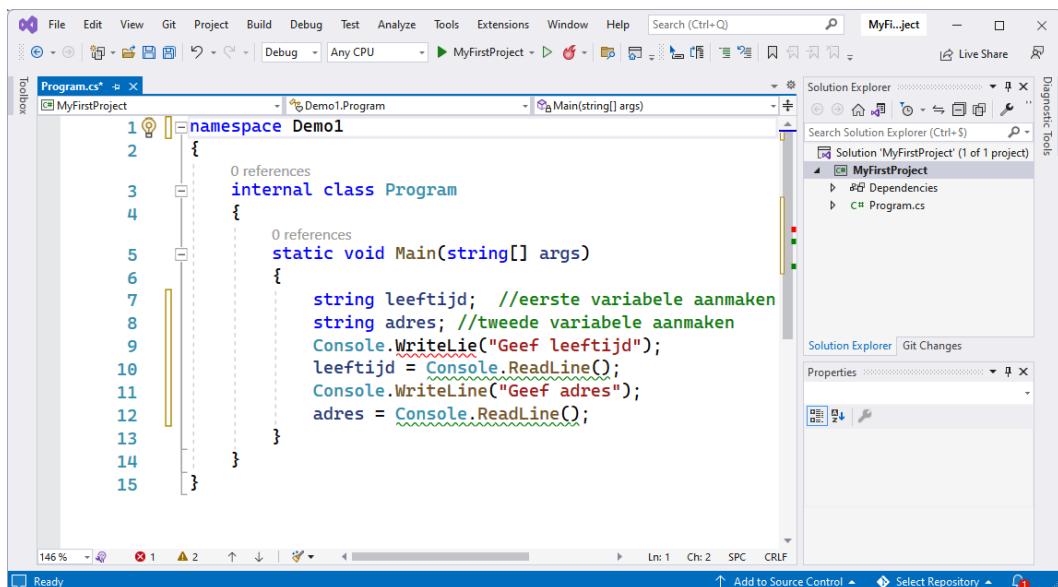
Als je dus cw schrijft en dan twee maal op de tab-toets van je toetsenbord duwt verschijnt daar *automatisch* een verse lijn met `Console.WriteLine();`.

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(en) meestal is dat gewoon het ontbreken van een komma of punt achter 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?!

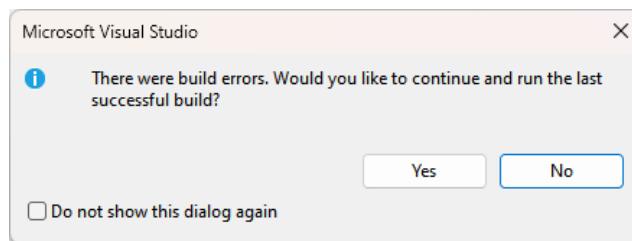
Dit zal de “error list” openen (**een schermdeel van VS dat ik aanraad om altijd open te laten én dus niet weg te klikken**). Warnings kunnen we - voorlopig - meestal negeren en deze ‘filter’ hoeft dus niet aan te zetten.

Error List						
Entire Solution		1 Error	0 of 2 Warnings	0 Messages	7	
Code	Description	Project	File	Line	Suppression State	
CS0115	'Console' does not contain a definition for 'WriteLie'	MyFirstProject	Program.cs	9	Active	

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):

```
6
7     string leeftijd; //eerste variabele aanmaken
8     string adres; //tweede variabele aanmaken
9     Console.WriteLine("Geef leeftijd");
10    Change 'WriteLie' to 'WriteLine'.
11    Introduce local for 'Console.WriteLine("Geef leeftijd")'
12    adres = Con
13    }
14 }
```

The screenshot shows a code editor in Microsoft Visual Studio. A yellow lightbulb icon with a red 'X' is positioned next to the word 'WriteLie' in line 9. A tooltip above the cursor says 'Change 'WriteLie' to 'WriteLine''. Below the code, a context menu is open with the option 'Introduce local for 'Console.WriteLine("Geef leeftijd")''. The code itself is as follows:

```
string leeftijd; //eerste variabele aanmaken
string adres; //tweede variabele aanmaken
Console.WriteLine("Geef leeftijd");
Change 'WriteLie' to 'WriteLine'.
Introduce local for 'Console.WriteLine("Geef leeftijd")'
adres = Con
{
}
```

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 gevoelighed 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 **achtergrondkleur** (`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 niet 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 slechtziend 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#:

- **Hooflettergevoelig:** C# is hooflettergevoelig. 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:

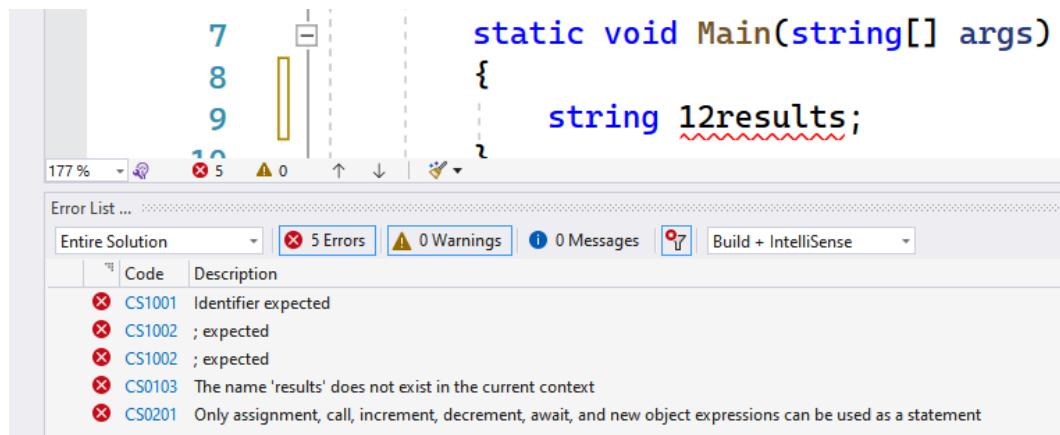
- De **identifier** waarmee we snel aan de variabele-waarde kunnen. Dit is de gebruiksvriendelijke naam die we geven aan een geheugenplek.
- Het **datatype** dat aangeeft wat voor soort data we wensen op te slaan. Enkel en alleen dat soort type data zal in deze variabele kunnen bewaard worden.

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:

- **Hooflettergevoelig:** 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 foutberichten. 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	geserveerd keyword
WriteLine	ja	
_____	ja	

2.2.2 Naamgeving afspraken

Er zijn geen vaste afspraken over hoe 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:

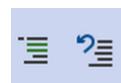
```
1 //De start van het programma
2 int getal = 3;
3 //Nu gaan we rekenen
4 int result = getal * 5;
5 // result = 3*5;
6 Console.WriteLine(result); //We tonen resultaat op scherm: 15
```

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 verscheine 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 signed: m.a.w. 1 bit wordt gebruikt om het + of - teken te bewaren.
- De u vooraan **ushort**, **uint** en **ulong** staat voor unsigned. 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, leestekens, 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 zevenhonderdvijfenzeventigduizend 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 hetzelfde 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 **toekennings-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 getalatypes 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 **uint 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 aanhalingstekens 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 explicet de waarde 4 wilt toekennen aan een variabele dan is het getal 4 in je code een zogenaamde **literal**.

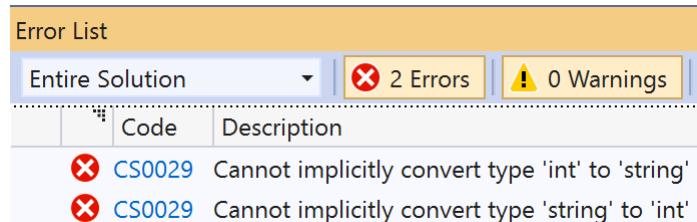
Voorbeelden van een literal toekennen:

```
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 foutberichten:



Figuur 2.3: Foutbericht 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 foutbericht 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
getal++;	getal = getal+1;	variabele met 1 verhogen
getal--;	getal = getal-1;	variabele met 1 verlagen
getal+=3;	getal = getal+3;	variabele verhogen met een getal
getal-=6;	getal = getal-6;	variabele verminderen met een getal
getal*=7;	getal = getal*7;	variabele vermenigvuldigen met een getal
getal/=2;	getal = getal/2;	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 hetzelfde, 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 wonderlijke 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 vél 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 kommagallen 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 afspreek 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

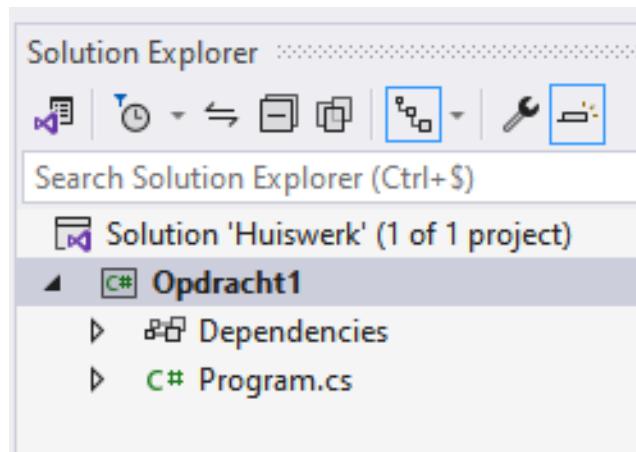
Wanneer je in VS een nieuw project start ben je niet verplicht om de “Project name” en “Solution name” dezelfde waarde te geven. Je zal wel merken dat bij het invoeren van de “Project name” de “Solution name” dezelfde invoer krijgt. Je mag echter vervolgens perfect de “Solution name” aanpassen.

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

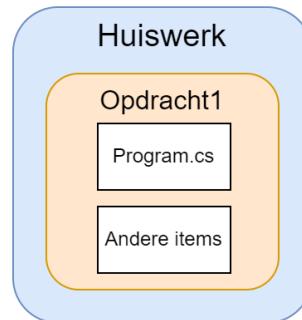
1. Naam van het project = **Opdracht1**
2. Naam van de solution = **Huiswerk**

En plaatsen deze in de folder C:\Temp.

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

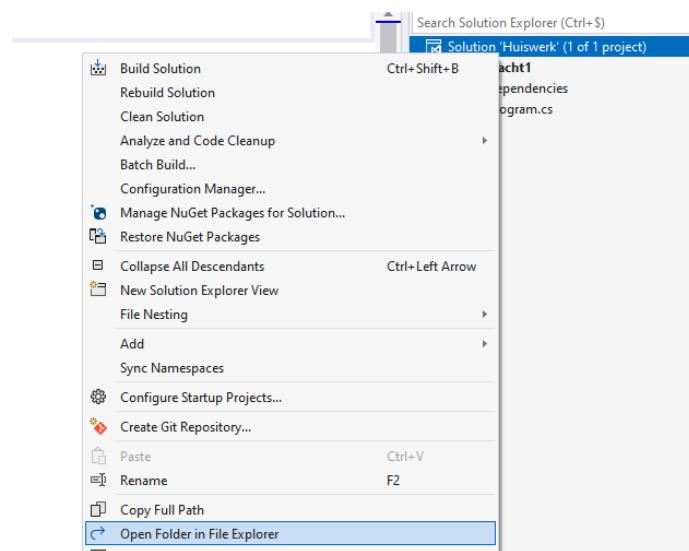


Figuur 2.4: Je ziet duidelijk een hiërarchie: bovenaan de solution *Huiswerk*, met daarin een project *Opdracht1*, gevuld met informatie zoals het *Program.cs* bestand. Deze hiërarchie zal je ook terugzien als je via de verkennner vervolgens naar de aangemaakte folder zou gaan.



Figuur 2.5: De hiërarchie anders voorgesteld.

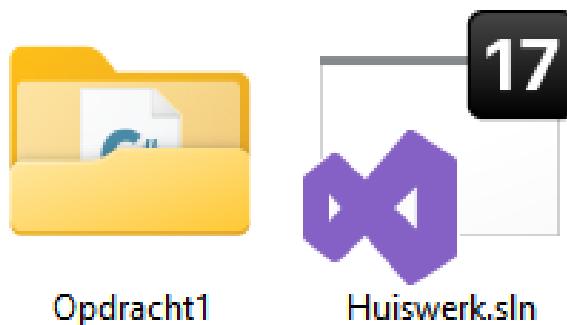
Rechtershik nu op de solution en kies “Open folder in file explorer”. Je kan deze optie kiezen bij eender welk item in de solution explorer. Het zal er voor zorgen dat de verkennner wordt geopend op de plek waar het item staat waar je op rechtershikte. Op die manier kan je altijd ontdekken waar een bestand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



Figuur 2.6: Tip: rechterklikken 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.

Deze pc > Windows (C:) > Temp > Huiswerk >



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. Rechtersklik 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.

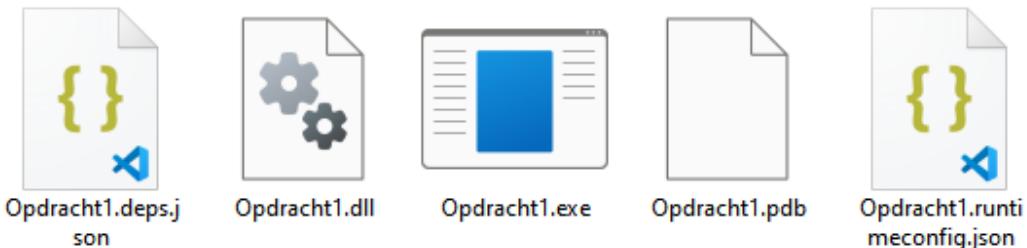


Een .cs-bestand rechtstreeks vanuit de verkenner openen werkt niet zoals je zou verwachten. VS zal weliswaar de inhoud van het bestand tonen, maar je kan verder niets doen. Je kan niet compileren, debuggen, enz. De reden is eenvoudig: een .cs bestand op zichzelf is nutteloos. Het heeft pas een bestaansreden 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

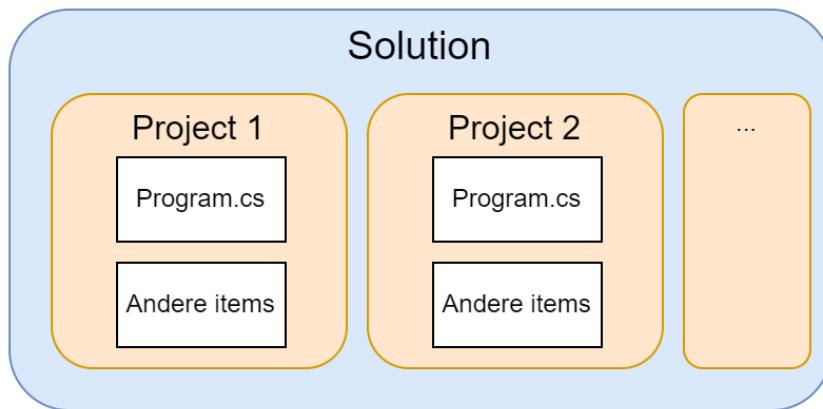


Figuur 2.8: Inhoud van bin/debug/net8.0 nadat project werd gecompileerd

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 verkenner gestart worden, hoera!

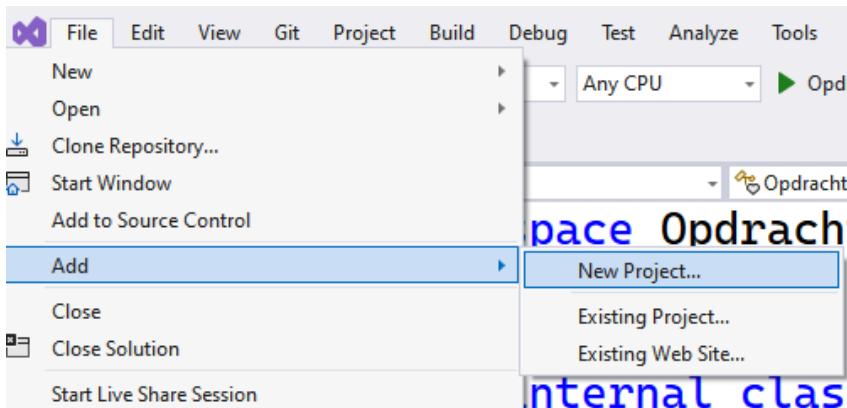
Merk op dat je de volledige inhoud van deze folder moet meegeven indien je je gecompileerde resultaat aan iemand wilt geven om uit te voeren.

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.

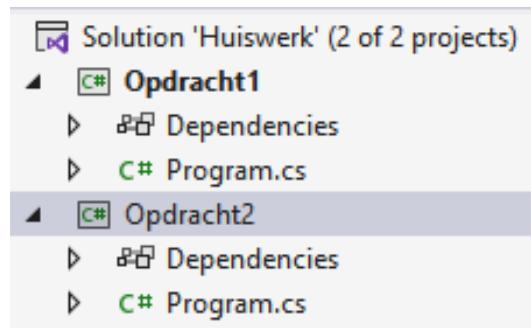
Ik zei net dat een solution meerdere projecten kan bevatten. Maar hoe voeg je een extra project toe? Terwijl je huidige solution open is (waar je een project wenst aan toe te voegen) kies je in het menu voor *File->Add->New project...*



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.



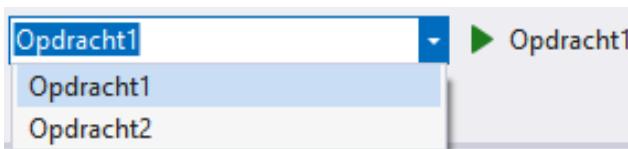
Figuur 2.11: Gelukkig kan je zaken dichtklappen m.b.v. driehoekjes naast iedere item.

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 modular 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*.

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

3 Tekst gebruiken in code

Ieder teken dat je op je toetsenbord kunt intypen is een **char**. Je toetsenbord bevat echter maar een kleine selectie van alle mogelijke tekens. Vergelijk jouw toetsenbord maar eens met dat van iemand uit bijvoorbeeld Spanje, Tunesië of China.

Voordat we leren hoe je in C# input van het toetsenbord uitleest, moeten we begrijpen hoe al die tekens in een computer worden voorgesteld. Dit gebeurt via de UNICODE standaard. Lang geleden was er de ASCII-standaard, die bepaalde welk teken bij welke hexadecimale waarde hoorde. Iedereen die ASCII volgde, kon zo berichten met die tekens naar elkaar sturen.

UNICODE volgt de ASCII-standaard op. Door de verdere digitalisering van de wereld bleek de ASCII-standaard als snel te klein. ASCII kan maar 128 karakters voorstellen, via 7 bits. Dit is weinig vergeleken met de meer dan 1 miljoen tekens in UNICODE, die een 16-bit (UTF-16) voorstelling gebruikt. Er is ook een 32-bit voorstelling mogelijk (UTF-32).

UNICODE bevat ook de eerste 128 tekens van ASCII. Daardoor zijn beide standaarden compatibel. Dankzij UNICODE kunnen we nu wereldwijd elke smiley, letter, en pictogram op dezelfde manier delen. Voor statistiek liefhebbers: er zijn 1.111.998 UNICODE tekens mogelijk. In versie 15.1, uitgebracht in september 2023, zijn daarvan 149.813 tekens gedefinieerd. Er is dus nog ruimte over.

De eerste 32 karakters zijn “onzichtbare” karakters die een historische reden (in ASCII) hebben om in de lijst te staan, maar sommige ervan zijn ondertussen niet meer erg nuttig. Origineel werd ASCII ontwikkeld als standaard om in de vorige eeuw via de Telex te communiceren. Vandaar dat vele van deze karakters commando’s lijken om oude typemachines aan te sturen (*line feed, bell, form feed, enz.*), wat ze dus ook effectief zijn!

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	:	91	5B	{	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	\
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	1	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figuur 3.1: De eerste 128 karakters met hun waarden (bron Wikipedia).

3.1 Tekst datatypes

In het vorige hoofdstuk werkten we vooral met getallen en haalden we maar kort het **string** en **char** datatype aan. In dit hoofdstuk ga ik dieper in op deze 2 veelgebruikte datatypes.

3.1.1 Char

Een **enkel karakter** (cijfer, letter, leestekens, enz.) als ‘tekst’ opslaan kan je doen door het **char**-type te gebruiken. Je kan één enkel karakter als volgt tonen:

```
1 char eenLetter = 'X';
2 Console.WriteLine(eenLetter);
```

Het is belangrijk dat je de apostrof ('') niet vergeet voor en na het karakter dat je wenst op te slaan daar dit de literal voorstelling van **char**-literals is. Zonder die apostrof denkt de compiler dat je een variabele wenst aan te roepen van die naam.

Je kan eender welk UNICODE-teken in een **char** bewaren, namelijk een letter, een cijfer of een speciaal teken zoals %, \$, *, #, enz. **Intern wordt de UNICODE van het character bewaard in de variabele, zinnde een 16 bit getal**. Deze laatste zin is belangrijk: voor een computer zijn **char**-variabelen niet meer dan getallen met een speciale betekenis.

Merk dus op dat volgende lijn: **char eenGetal = '7'**; weliswaar een getal als teken opslaat, maar dat intern de compiler deze variabele steeds als een **char** zal gebruiken. **Als je dit cijfer zou willen gebruiken als effectief cijfer om wiskundige bewerkingen op uit te voeren, dan zal je dit eerst moeten converteren naar een getal** (we zullen dit in hoofdstuk 4 uitleggen).

3.1.2 String

Een **string** is een reeks van 0, 1 of meerdere **char**-elementen.

We gebruiken het **string** datatype om tekst voor te stellen. Je begrijpt waarschijnlijk zelf wel waarom het **string** datatype een belangrijk en veelgebruikt type is in elke programmeertaal: er zijn maar weinig applicaties die niet minstens enkele lijnen tekst tonen aan de gebruiker.



In hoofdstuk 8 zullen we ontdekken dat strings eigenlijk zogenaamde arrays zijn.

3.1.2.1 Strings declareren

Merk op dat we bij een **string** literal gebruik maken van aanhalingstekens ("") terwijl bij een **char** literal we een apostrof gebruiken (''). Dit is de manier om een **string** met lengte 1 van een **char** te onderscheiden.

Volgende code geeft drie keer het cijfer 1 onder elkaar op het scherm. Maar de eerste keer gaat het om het een **char** (enkelvoudig teken), dan om een **string** (1 of meerdere tekens) en dan een **int** (effectief getal, bestaande uit 1 of meer cijfers):

```
1 char eenKarakter = '1';
2 string eenString = "1";
3 int eenGetal = 1;
4
5 Console.WriteLine(eenKarakter);
6 Console.WriteLine(eenString);
7 Console.WriteLine(eenGetal);
```

Het programma zal driemaal een 1 onder elkaar tonen. Boeiend programma, hoor.

3.2 Escape characters



De voorman hier! Escape characters zijn niet de boeiendste materie om te bespreken. Je zou nog kunnen hopen dat het een opvolger is van Prison Break of zo. Helaas is dat niet zo. Echter: als je escape characters beheerst zal je veel eenvoudiger én mooier tekst op je scherm kunnen toveren. Let dus even goed op a.u.b.

Naast letters en tekens mogen in string en chars ook escape characters staan. In C# hebben bepaalde tekens namelijk een speciale functie. Denk maar aan de dubbele aanhalingstekens ("") om het begin en einde van een string-literal mee aan te geven.

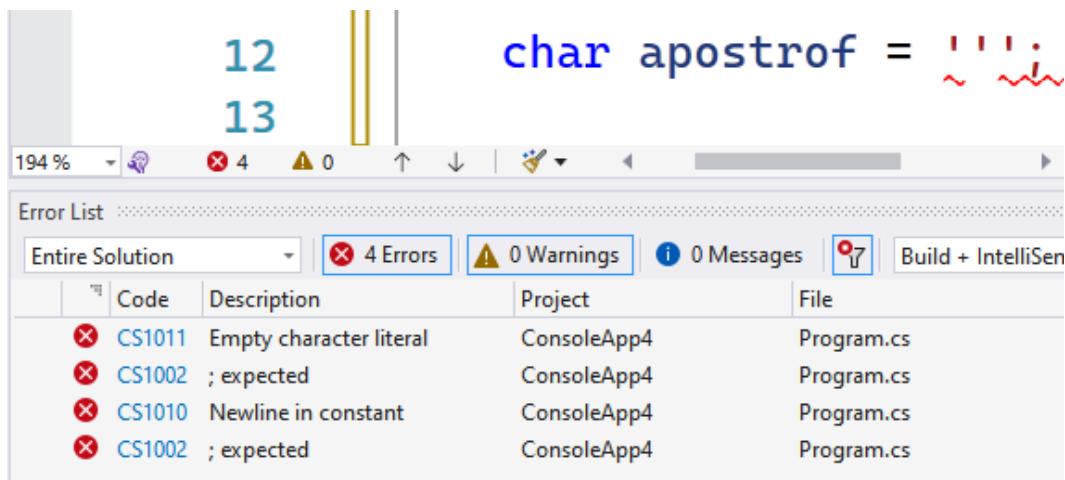
We hebben dus een manier nodig om aan te duiden wanneer de compiler het eerstvolgende teken, zoals een " als een **char** moet beschouwen. We lossen dit op met behulp van escape characters. **Deze worden met een backslash (\) aangeduid, gevuld door het karakter dat we wensen te gebruiken.**

3.2.1 Voorbeeld van escape characters

Laten we eens kijken naar de werking van het weglatingssteken als voorbeeld (de zogenaamde apostrof of afkappingsteken, om bijvoorbeeld 's avonds te schrijven) De volgende code zal de compiler verkeerd interpreteren, omdat hij denkt dat we een leeg karakter willen opslaan:

```
1 char weglatingssteken = ''';
```

Het gevolg is een berg aan foutberichten omdat er na het sluitende weglatingssteken (het tweede) plots nog één (het derde) verschijnt. VS is volledig in de war en weet niet wat doen.



Figuur 3.2: Hulp! VS snapt er niets van!

Escape characters to the rescue! We gaan met de backslash aanduiden dat het volgende teken (het tweede weglatingsteken) een **char** voorstelt en niet het sluitende teken in de code.

```
1 char weglatingsteken = '\'';
```

Een backslash in een char of string-literal geeft aan dat het volgende teken als een literal moet worden gezien en niet als een speciaal teken in C#.

3.2.2 Veel gebruikte escape chars

Er zijn verschillende escape characters in C# toegelaten. We lijsten hier de belangrijkste¹ op:

```
1 \'      //de apostrof zoals zonet besproken.
2 \"      //een aanhalingsteken.
3 \\      //een backslash in je tekst tonen.
4 \\\\"    //twee backslashes.
5 \\n     //een nieuwe lijn (zogenaamde *enter* of *newline*).
6 \\t     //Horizontale tab.
7 \\uxxxx  //een teken met als hexadecimale UNICODE waarde xxxx.
```

¹Voor een totaal overzicht kijk eens op docs.microsoft.com/dotnet/csharp/programming-guide/strings.

3.2.3 Escape characters in strings

Aangezien strings eigenlijk bestaan uit 1 of meerdere char-elementen, is het logisch dat je ook in een string met escape characters kunt werken. Het woord “'s avonds” schrijf je bijvoorbeeld als volgt:

```
1 string woord = "\'s avonds";
```

Idem met aanhalingstekens. Stel je voor dat je een programma wilt schrijven dat C# code op het scherm toont. Dat doe je dan met volgende, nogal Inception-achtige, manier:

```
1 string inceptionCode = "Console.WriteLine(\"Cool he\");";
2 Console.WriteLine(inceptionCode);
```

Merk op dat we voorgaande code nog meer Inception-like kunnen maken door de string ineens in de WriteLine methode te plaatsen:

```
1 Console.WriteLine("Console.WriteLine(\"Cool he\");");
```

Beide voorbeelden zullen dus volgende tekst op het scherm geven: `Console.WriteLine("Cool he")`;

3.2.4 Biep biep

\a mag je enkel gebruiken als je een koptelefoon op hebt daar dit het escape character is om de computer een biepje te laten doen.

Volgende code voorbeeld zal, als alles goed gaat, een zin op het scherm tonen en dan ogenblikkelijk erna een biepje:

```
1 Console.WriteLine("Een zin en dan nu ... de biep\a");
```

3.2.5 Witregels en tabs

We gebruiken vooral escape characters in strings om bijvoorbeeld witregels en tabs aan te geven. Test bijvoorbeeld volgende lijn code eens:

```
1 string eenString = "Een zin.\t na een tab \nDan eentje op een nieuwe
  regel";
2 Console.WriteLine(eenString);
```

Dit zal als output geven:

```
1 Een zin.          na een tab
2 Dan eentje op een nieuwe regel
```

3.2.6 Over tabstops

Als je het niet gewoon bent de tab-toets op je toetsenbord te gebruiken dan is de eerste werking van mogelijk verwarrend. Nochtans is in een string gebruiken exact hetzelfde als op de tab-toets duwen.

In je console-scherm zijn de tab stops vooraf bepaald. Wanneer je dus een tab invoegt zal de cursor zich verplaatsen naar de eerstvolgende tab stop. In volgende tekstuitvoer zie je de tabstops op de tweede lijn “gevisualiseerd”:

```
1 01234567890123456789012345678901234567890123456789
2           1     2     3     4     5
```

Bovenstaande uitvoer werd als volgt gemaakt:

```
1 Console.WriteLine("01234567890123456789012345678901234567890");
2 Console.WriteLine("\t1\t2\t3\t4\t5");
```

Tabstops zijn nuttig om je data mooi uitgelijnd in een tabel te plaatsen. Als je dat dan nog eens combineert met de UNICODE karakters om tabellen te tekenen kan je toffe dingen maken. Deze karakters, de zogenaamde “Box Drawing” subset, staan in UNICODE gedefinieerd als de tekens met hexadecimale code 0x2500 en verder. Bekijk zeker eens een datasheet met alle tekens.².

3.2.7 Het verbatim karakter @

Het apenstaartje (@) voor een **string** literal plaatsen is zeggen “beschouw alles binnen de aanhalingstekens als effectieve karakters die deel uitmaken van de inhoud van de tekst”. Dit teken heet daarom binnen C# niet voor niets het **verbatim** karakter. Het is belangrijk te beseffen dat **escape characters genegeerd worden** wanneer we het verbatim karakter gebruiken. Dit is vooral handig als je bijvoorbeeld een netwerkadres wilt schrijven en niet iedere \ wilt escappen:

```
1 string zonderAt = "C:\\Temp\\Myfile.txt";
2 string metAt = @"C:\\Temp\\Myfile.txt";
```

Merk op dat aanhalingstekens nog steeds ge-escape'd moeten worden. Heb je dus een stuk tekst met een aanhalingsteken in dan zal je zonder het apenstaartje moeten werken.

²www.unicode.org/charts/PDF/U2500.pdf

Uiteraard kan je ook het apenstaartje gebruiken in `Console.WriteLine`. Volgende zal dus de escape karakters tonen in plaats van “uitvoeren”:

```
1 Console.WriteLine(@"Om een tab te tonen gebruik je \t in C#.");
```

Wat zal resulteren in volgende uitvoer:

```
1 Om een tab te tonen gebruik je \t in C#.
```

3.3 Strings samenvoegen

Tot nogtoe gebruikten we de `+`-operator om strings aan elkaar te plakken. We gaan deze manier meer in detail bekijken, gevolgd door een moderner alternatief: door middel van string interpolatie met de `$`-notatie.

In de volgende sectie gaan we van volgende informatie uit:

- Stel dat je 2 variabelen hebt `int leeftijd = 13` en `string naam = "Finkelstein"`.
- We willen de inhoud van deze variabelen samenvoegen in een nieuwe `string` `zin` die zal bestaan uit de tekst: Ik ben Finkelstein en ik ben 13 jaar.

3.3.1 String samenvoegen met de `+`-operator

Je kan strings en variabelen eenvoudig bij elkaar ‘optellen’ zoals we in het begin van dit boek hebben gezien. Ze worden dan achter elkaar geplakt (**geconcateneerd**).

```
1 string zin = "Ik ben " + naam + " en ik ben " + leeftijd+ " jaar.;"
```

Let er op dat je tussen de aanhalingsteken (binnen de strings) spaties zet indien je het volgende deel niet tegen het vorige deel wilt *plakken*. Is hiermee alles gezegd?! Nee, toch even goed opletten hier. **De volgorde van strings met andere types samenvoegen bepaalt wat de uitvoer zal zijn.**

Kijk zelf:

```
1 Console.WriteLine("1"+1+1);
2 Console.WriteLine(1+1+"1");
3 Console.WriteLine("1" + (1 + 1));
```

Geeft als uitvoer:

```
1 111
2 21
3 12
```

Was dit de uitvoer die je voorspeld had?

Ook in dit soort code wordt de volgorde van bewerkingen gerespecteerd. De **concatenatie gebeurt van links naar rechts en de linkse operand zal steeds bepalen wat het resultaat van de bewerking zal zijn indien er twijfel is**. Dit nieuw samengevoegde deel wordt dan de linkse operand voor het volgende deel.

Kijken we dus naar `"1"+1+1` dan wordt dit eerst `"11"+1` en vervolgens dit `"111"`.

Bij `1+1+"1"` krijgen we eerst `2+"1"`. Dit geeft vervolgens 21. Aangezien C# niet kan bepalen dat de string iets bevat wat een getal kan zijn, en dus besluit om beide operanden als een **string** te zien wat altijd de veiligste oplossing is.

3.3.2 String interpolation met \$-notatie

Het nadeel van de +-operator is dat je strings soms erg lang en onleesbaar worden.

Dankzij *string interpolation* kan dit wel **waarbij we het \$-teken gebruiken vooraan de string om aan te geven dat specifieke delen van de zin geïnterpoleerd moeten worden**

Door het \$-teken **VOOR** de string te plaatsen geef je aan dat alle delen in de string die *tussen accolades staan* als code mogen beschouwd worden. Een voorbeeld maakt dit duidelijk:

```
1 string zin = $"Ik ben {naam} en ik ben {leeftijd} jaar.";
```

In dit geval zal de inhoud van de variabele naam tussen de string op de plek waar nu `{naam}` staat geplaatst worden. Idem voor `leeftijd`. Zoals je kan zien is dit veel meer leesbare code dan de eerste manier.

Het resultaat zal dan worden: `Ik ben Finkelstein en ik ben 13 jaar.`

3.3.2.1 Berekeningen doen bij string interpolatie

Je mag eender welke *expressie* tussen de accolades zetten bij string interpolation, denk maar aan:

```
1 string zin = $"Ik ben {leeftijd+4} jaar.";
```

Alle expressies tussen de accolades zullen eerst uitgevoerd worden voor ze tussen de string worden geplaatst. De uitvoer wordt nu dus: `Ik ben 17 jaar.`

Eender welke expressie is toegelaten, dus je kan ook complexe berekeningen of zelfs andere methoden aanroepen:

```
1 string zin = $"Ik ben {leeftijd*leeftijd+(3*2)} jaar.";
```



Uiteraard mag je dit dus ook gebruiken wanneer je eenvoudigere zaken naar het scherm wenst te sturen gebruik makende van `Console.WriteLine` en interpolatie:

```
1 Console.WriteLine($"3 maal 9 is {3*9}");
```

3.3.2.2 Mooier formatteren

Bij string interpolation kan je ook extra informatie meegeven hoe het resultaat juist weergegeven moet worden. Dit noemen we *formatteren*. Je geeft dit aan door na de expressie, binnen de accolades, een dubbelpunt te plaatsen gevolgd door de manier waarop moet geformatteerd worden.

Wil je een kommagetal tonen met maar 2 cijfers na de komma dan schrijf je:

```
1 double number = 12.345;  
2 Console.WriteLine($"{number:F2}");
```

Er zal 12.35 op het scherm verschijnen. F2 na het dubbelpunt geeft aan dat je een *float* wilt met 2 beduidende cijfers na de komma.

Merk op dat bij string formattering er **afgerond** wordt, en dus niet *afgekapt*.

Nog enkele nuttige vormen:

- D5: toon een geheel getal als een 5 cijfer getal. 123 wordt 00123. Maar 123456 zal volledig getoond worden. De Dx formattering werkt enkel op gehele getallen.
- E2: wetenschappelijke notatie met 2 cijfers precisie (12000000 wordt 1,20E+007 “1 komma 2 maal tien tot de zevende”)
- C: geldbedrag. 12,34 wordt € 12,34. Het teken en het aantal beduidende cijfers is van de landinstellingen van de pc waarop je code wordt uitgevoerd. Het euro teken zal mogelijk als een ? getoond worden. In de volgende sectie tonen we hoe je dit kan oplossen.

Alle overige format specifiers kan je in de documentatie opzoeken³.

³Zie docs.microsoft.com/dotnet/standard/base-types/standard-numeric-format-strings.

3.3.2.3 Formateren met een masker

Een andere eenvoudige manier om strings te formatteren is door middel van een masker bestaande uit 0'n. Dit ziet er als volgt uit:

```
1 double number = 12.345;  
2 Console.WriteLine($"{{number:0.00}}");
```

We geven hierbij aan dat de variabele tot 2 cijfers na de komma moet getoond worden. Indien deze maar 1 cijfer na de komma bevat dan deze toch met twee cijfers getoond worden. Volgende voorbeeld toont dit:

```
1 double number = 12.3;  
2 Console.WriteLine($"{{number:0.00}}");
```

Er zal 12,30 op het scherm verschijnen.

Je kan dit masker ook gebruiken om te verplichten dat getallen bijvoorbeeld steeds met **minimum** 3 cijfers voor de komma getoond worden. Volgende voorbeeld toont dit:

```
1 double number = 12.3;  
2 double number2 = 99999.3;  
3 Console.WriteLine($"{{number:000.00}}");  
4 Console.WriteLine($"{{number2:000.00}}");
```

Geeft als uitvoer:

```
1 012.30  
2 99999.30
```



Vanaf nu zal ik bijna altijd string interpolatie gebruiken doorheen het boek. Dit is de meest moderne aanpak en zal 99% van de tijd meer leesbare code geven.

In de appendix leg ik uit hoe je vroeger met behulp van `String.Format()` strings moest samenvoegen (daar je dit soms nog in *legacy* code zal tegenkomen).

3.4 Optellen van char variabelen

We hebben al gezien dat intern een **char** als een geheel getal wordt voorgesteld. Stel dat we volgende **char**-variabelen aanmaken:

```
1 char letter1 = 'A';
2 char letter2 = 'B';
```

Bij string mogen we de +-operator gebruiken om 2 strings aan elkaar te plakken. **Bij char mag dat niet!** Of beter, dit mag maar zal niet het resultaat geven dat je mogelijk verwacht wanneer je voor het eerst hiermee leert werken. Oordeel zelf:

```
1 Console.WriteLine(letter1 + letter2);
```

Wanneer je deze code uitvoert dan krijg je 131 te zien (en dus niet “AB” zoals je misschien had verwacht).

Had je dit verwacht? Denk eraan dat het char-type z’n waarde als getallen bijhoudt, de zogenaamde UNICODE-voorstelling van het karakter. Als de compiler het volgende ziet staan:

letter1 + letter2

dan zal de compiler deze twee waarden letterlijk optellen en het nieuw verkregen getal als resultaat geven:

- De UNICODE-voorstelling van A is 0x041 oftewel **65**. In het geheugen staat dus het geheel getal 65.
- B wordt voorgesteld door **66**.
- Als we dus de variabelen **letter1** en **letter2** optellen geeft dit **131**.

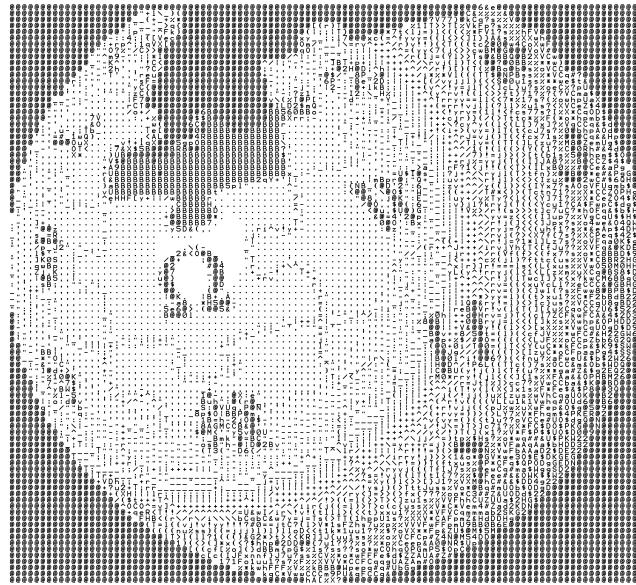


Je zou misschien verwachten dat C# vervolgens het element op plaats 131 in de UNICODE tabel zou tonen.

Dat is niet juist: de +-operator is niet gedefinieerd voor het **char** datatype, maar wel voor het **int** datatype. Daarom beschouwt de compiler de operanden **letter1** en **letter2** als **int**. De som van twee **int** waarden geeft een **int** resultaat. We zien daarom 131 op het scherm in plaats van het UNICODE karakter met waarde 131 (een Latijnse i zonder punt). In het volgende hoofdstuk leren we hoe je dit wel kunt doen.

3.5 Vreemde tekens in console tonen

Niets is zo leuk als de vreemdste UNICODE tekens op het scherm tonen. In oude console-games werden deze tekens vaak gebruikt om complexe tekeningen op het scherm te tonen. Om je ietwat saaie applicaties dus wat toffer te maken, leg ik daarom uit hoe je dit kan doen.



Figuur 3.3: Dit Wikipedia logo bestaat volledig uit UNICODE karakters.

3.5.1 UNICODE karakters tonen

Je toetsenbord heeft maar een beperkt aantal toetsen. Er zijn echter tal van andere tekens gedefinieerd die console-applicaties ook kunnen gebruiken. We zagen reeds dat al deze tekens, UNICODE-karakters, een eigen unieke code hebben die je kan opzoeken om vervolgens dat teken in je code te gebruiken.

Dit gaan als volgt in z'n werk:

1. Zoek het teken(s) dat je nodig hebt in een UNICODE-tabel⁴ en noteer de hexadecimale waarde.
 2. Plaats bovenaan in je Main: `Console.OutputEncoding = System.Text.Encoding.UTF8;`
 3. Je kan nu op 2 manieren dit teken op het scherm krijgen.

Stel je voor dat we het copyright karakter wensen te gebruiken (de letter c in een cirkeltje) in onze applicatie. Deze heeft hexadecimale UNICODE waarde 0x00A9.

⁴Zie [UNICODE-table.com](#)).

3.5.1.1 Manier 1: copy/paste

Kopieer het karakter zelf en plaats het in je code waar je het nodig hebt, bijvoorbeeld:

```
1 Console.WriteLine("<plak hier je speciale teken>");
```

Merk op dat niet alle lettertypes dit karakter kennen en dus mogelijk als een vierkantje dit op je scherm zullen tonen. Dit hangt af van het lettertype dat jouw shell-venster gebruikt.

3.5.1.2 Manier 2: hexadecimale code casten naar char

Casting leg ik pas in het volgende hoofdstuk uit, maar het kan geen kwaad om al eens een voorproefje hiervan te krijgen.

Noteer de hexadecimale code van het karakter dat in de tabel staat. In dit geval is dat dus 0x00A9. Om dit te tonen schrijf je dan:

```
1 char copyright = (char)0x00A9;
2 Console.WriteLine(copyright);
```

Dit kan ook korter. Door gebruik te maken van de \u-notatie om hexadecimale waarden voor te stellen:

```
1 Console.WriteLine("\u00A9");
```

3.5.2 UNICODE-kunst tonen

Soms zou je *multiline* UNICODE-kunst (ook wel ASCII-art genoemd) willen tonen in je C# applicatie. Dit kan je eenvoudig oplossen door gebruik te maken van het @ teken voor een string.

Stel dat je een toffe titel of tekening bijvoorbeeld via [ASCIIflow.com](https://asciiflow.com) maakt. Je kan het resultaat eenvoudig naar je klembord kopiëren en vervolgens in je C#-code integraal copy & paste als literal voor een **string** op voorwaarde dat je het laat voorafgaan door @" en uiteraard eindigt met ";

Bijvoorbeeld:

```

1 string myname = @""
2
3 \_ _ _ / \
4 | | | |
5 | | | |
6 | | | |
7 | | | |
8 Console.WriteLine(myname);

```



Zowel de \$-notatie (voor string interpolatie) als het @-teken kan je gecombineerd gebruiken bij een string:

```
1 Console.WriteLine(${@"1/1={1+1}. \tGeen tab"});
```

Dit geeft als output (wordt door het apenstaartje genegeerd):

```
1 1/1=2. \tGeen tab
```



In de vorige sectie legde ik uit dat we tekst kunnen formateren als een geld bedrag m.b.v. `Console.WriteLine(${12.3456:C})` .

Het probleem was dat het euro-teken als een ? op het scherm verscheen. Dit is omdat het euro-teken een nieuwe karakter is en dus binnen de UNICODE tabellen bestaat, maar niet binnen de klassieke ASCII-tabel.

Willen we dit teken dus gebruiken dan moeten we nog eerst de juiste *encoding* aanduiden bovenaan:

```
1 Console.OutputEncoding = System.Text.Encoding.UTF8;
2 Console.WriteLine(${12.3456:C})
```

3.6 Environment bibliotheek

De Console bibliotheek is maar 1 van de vele bibliotheken die je in je C# programma's kunt gebruiken.

Een andere nuttige bibliotheek is de Environment-bibliotheek. Deze geeft je applicatie allerlei informatie over de computer waarop het programma op dat moment draait. Denk maar aan het werkgeheugen, gebruikersnaam van de huidige gebruiker, het aantal processoren enz.



De laatste zin in vorige alinea is belangrijk: als je jouw programma op een andere computer laat uitvoeren zal je mogelijk andere informatie verkrijgen.

Wil je een programma dus testen dat deze bibliotheek gebruikt, is het aangeraden om het op meerdere systemen met verschillende eigenschappen te testen.

Hier enkele voorbeelden hoe je deze bibliotheek kunt gebruiken (kijk zelf of er nog nuttige properties over je computer in staan):

```
1 bool is64bit = Environment.Is64BitOperatingSystem;
2 string pcname = Environment.MachineName;
3 int procCount = Environment.ProcessorCount;
4 string username = Environment.UserName;
5 long memory = Environment.WorkingSet; //zal ongeveer 10 Mb zijn
```

Vervolgens zou je dan de inhoud van die variabelen kunnen gebruiken om bijvoorbeeld aan de gebruiker te tonen wat z'n machine naam is:

```
1 Console.WriteLine($"Je computernaam is {pcname}");
2 Console.WriteLine($"Dit programma gebruikt {memory} byte geheugen");
3 Console.WriteLine($"En je usernaam is {Environment.UserName}");
```

In de laatste lijn code tonen we dat je uiteraard ook rechtstreeks de variabelen uit Environment in je string interpolatie kunt gebruiken en dus niet met een *tussenvariabele* moet werken.

Je kan in de documentatie⁵ opzoeken welke nuttige zaken je nog met de bibliotheek kunt doen.

WorkingSet bijvoorbeeld geeft terug hoeveel geheugen het programma van Windows toegewezen krijgt. Als je dus op 'run' klikt om je code te runnen dan zal dit programma geheugen krijgen en via WorkingSet kan het programma dus zelf zien hoeveel het krijgt. Test maar eens wat er gebeurt als je programma maakt dat uit meer lijnen code bestaat.

⁵Zie docs.microsoft.com/dotnet/api/system.environment.

3.6.1 Programma afsluiten

De Environment bibliotheek heeft ook een methode om je applicatie af te sluiten. Je doet dit met behulp van `Environment.Exit(0)`. Het getal tussen haakjes mag je zelf bepalen en is de zogenaamde *exitcode* die je wilt meegeven bij het afsluiten. Als je dan later via logbestanden wilt onderzoeken waarom het programma stopte dan kan je dit zien aan de hand van deze *exitcode*.



Mogelijk was deze laatste sectie wat verwarring. Dat is bewust gedaan... sort of. C# leren kan in het begin soms nogal saai lijken. Daarom dat ik ervoor kies om hier en daar een iets meer geavanceerd aspect te bespreken.

Zoals al eerder verteld: C# komt met een hele grote hoop bibliotheken (denk maar aan `Environment` en `Console`). Voor zover ik weet, bestaat er niemand die iedere bibliotheek of klasse kent. Het is aan jou, als gepassioneererde programmeur, om zelf te ontdekken welke bibliotheken je nuttig lijken te geven voor een bepaald probleem.

4 Werken met data

Aah, Data, een geliefkoosd personage uit Star Trek. Maar daar ga ik het niet over hebben. Het wordt tijd dat we onze werkkleidij aantrekken en ons echt vuil gaan maken.

De wereld draait op data, en dus ook de meeste applicaties die wij gaan schrijven. We hebben al gezien dat C# met verschillende datatypes werkt. Maar wat gebeurt er als we de inhoud van twee verschillende datatypes willen combineren?! *In Star Trek resulteerde dat 50% van de tijd in een aanval van de Borg, 20% van de tijd van de Klingons en in de overige 30% in een oersaaie aflevering (Star Wars for life!). Ahum, sorry. I got carried away.*

Laten we eens onderzoeken hoe we data van ‘vorm’ kunnen veranderen.

May the force be with you! Euh, ik bedoel: Make it so!

4.1 Appelen en peren

Wanneer je de waarde van een variabele wilt toekennen aan een variabele van een ander type mag dit niet zomaar. Volgende code zal bijvoorbeeld een dikke foutboodschap geven:

```
1 int leeftijd = 4.3;
```

 CS0266 Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

Figuur 4.1: Een zogenaamde ‘impliciete casting’ error.

Je kan geen appelen in peren veranderen zonder magie: in het geval van C# zal je moeten **converteren of casten**.

Dit kan op 3 manieren:

- Via **casting**: de (klassieke) manier die ook werkt in veel andere programmeertalen.
- Via de **Convert. bibliotheek** van .NET.
- Via **parsing** : Deze manier is enkel bruikbaar om strings om te zetten naar andere datatypes.

4.2 Casting

Het is onmogelijk om een kommagetal aan een geheel getal toe te wijzen zonder dat er informatie verloren zal gaan. Toch willen we dit soms doen. **Van zodra we een variabele van het ene type willen toekennen aan een variabele van een ander type en er dataverlies zal plaatsvinden dan moeten we aan casting doen.**

4.2.1 Wat is casting

Casting heb je nodig om een variabele van een bepaald type voor een ander type te laten doorgaan. Stel dat je een complexe berekening hebt waar je werkt met verschillende datatypes. Door te casten voorkom je dat je vreemde resultaten krijgt. Je gaat namelijk bepaalde types even als andere types gebruiken.

Het is belangrijk in te zien dat het casten van een variabele naar een ander type enkel een gevolg heeft tijdens het uitwerken van de expressie waarbinnen je werkt. De variabele in het geheugen zal voor eeuwig en altijd het type zijn waarin het origineel gedeclareerd werd. Je kan dus nooit het datatype van een variabele veranderen!

Casting duid je aan door voor de variabele of literal het datatype tussen haakjes te plaatsen naar wat het omgezet moet worden:

```
1 int mijngetal = (int)3.5;
```

of

```
1 double kommagetal = 13.8;
2 int kommaNietWelkom = (int)kommagetal;
```

Door casting te gebruiken ze je eigenlijk aan de compiler 2 zaken:

1. Volgende variabele die van het type **double** is, moet aan deze variabele van het type **int** toegekend worden.
2. **Ik besef dat hierbij data verloren kan gaan** (het deel na de komma), maar zet de variabele toch maar om naar het nieuwe type. “Ik draag de volledige verantwoordelijkheid voor de gevolgen hiervan verderop in het programma.”



Je dient enkel aan casting te doen wanneer je aan *narrowing* doet. **Bij narrowing gaan we een datatype omzetten naar een ander datatype dat een verlies aan data met zich zal meebrengen.**



Het is een voorzorgsmaatregel om te voorkomen dat we de compiler later verwijten dat hij belangrijke gegevens heeft verloren tijdens de omzetting. Door casting te gebruiken, geven we aan dat we de compiler niet de schuld zullen geven van dit dataverlies.

4.2.2 Narrowing

Casting doe je wanneer je een variabele wilt toekennen aan een andere variabele van een ander type dat daar eigenlijk **niet inpast** zonder dat dataverlies. We moeten dan aan **narrowing** doen, letterlijk het versmullen van de data.

Bekijk eens het volgende voorbeeld:

```
1 double hoofdMetting;  
2 int secundaireMetting;  
3 hoofdMetting = 20.4;  
4 secundaireMetting = hoofdMetting;
```

Dit zal niet gaan. Je probeert namelijk een waarde van het type `double` in een variabele van het type `int` te steken. Dat gaat enkel als je informatie weggooit (namelijk het gedeelte na de komma). Je moet aan *narrowing* doen.

Dit gaat enkel als je expliciet aan de compiler zegt: *het is goed, je mag informatie weggooien, ik begrijp dat en zal er rekening mee houden. Dit proces van narrowing noemen we casting.*

En je lost dit op door voor de variabele die tijdelijk dienst moet doen als een ander type, het nieuwe type, tussen ronde haakjes te typen, als volgt:

```
1 double hoofdMetting;  
2 int secundaireMetting;  
3 hoofdMetting = 20.4;  
4 secundaireMetting = (int)hoofdMetting;
```

Het resultaat in `secundaireMetting` zal 20 zijn (alles na de komma wordt weggegooid bij casting van een `double` naar een `int`).



Merk op dat `hoofdMetting` nooit van datatype is veranderd; enkel de inhoud ervan (20.4) werd eruit gehaald, omgezet (“gecast”) naar 20 en dan aan `secundaireMetting` toegewezen dat enkel `int` aanvaardt.

4.2.3 Narrowing in de praktijk

Stel dat `tempGisteren` en `tempVandaag` van het type `int` zijn, maar dat we nu de gemiddelde temperatuur willen weten. De formule voor gemiddelde temperatuur over 2 dagen is:

```
1 int tempGemiddeld = (tempGisteren + tempVandaag) /2;
```

Test dit eens met de waarden 20 en 25. Wat zou je verwachten als resultaat? Inderdaad: 22,5 (omdat $(20+25)/2 = 22.5$). Nochtans krijg je 22 op scherm te zien en zal de variabele `tempGemiddeld` ook effectief de waarde 22 bewaren en niet 22,5.

Het probleem is dat het gemiddelde van 2 getallen niet noodzakelijk een geheel getal is. **Echter, omdat de expressie enkel integers bevat (`tempGisteren`, `tempVandaag` en de literal 2) zal ook het resultaat een `int` zijn.** In dit geval wordt alles na de komma gewoon weggegooid, vandaar de uitkomst. **Dit is narrowing.**

Hoe krijgen we de correctere uitslag te zien? Eens testen wat er gebeurt als we `tempGemiddeld` als `double` declareren:

```
1 double tempGemiddeld = (tempGisteren + tempVandaag) / 2;
```

Als we dit testen zal nog steeds de waarde 22 . 0 aan `tempGemiddeld` toegewezen worden. De expressie rechts van de toekenning bevat nog steeds enkel integers en de computer zal dus ook de berekening en het resultaat als integer beschouwen, ongeacht dat deze in een `double` moet gezet worden.

We moeten dus ook de rechterkant van de toekenning als `double` beschouwen. *We doen dit, zoals eerder vermeld, door middel van casting*, als volgt:

```
1 double tempGemiddeld = ((double)tempGisteren + (double)tempVandaag) /  
2;
```

Nu zal `tempGemiddeld` wel de waarde 22 . 5 bevatten.



Er zijn ook andere oplossingen die het gewenste resultaat geven, namelijk:

```
1 (tempGisteren + tempVandaag)/2.0;  
2 ((double)(tempGisteren + tempVandaag))/2;  
3 ((double)tempGisteren + tempVandaag)/2;  
4 (tempGisteren + (double)tempVandaag)/2;
```

Let echter op dat niet alle oplossingen bij dit soort oefeningen steeds dezelfde resultaten geeft. Goed testen is de boodschap en nadenken over de volgorde van berekeningen en wat het datatype van ieder tussenresultaat zal zijn. Laten we dat eens analyseren bij voorgaande 4 voorbeelden:

1. Eerst tellen we twee integers op, wat dus een nieuwe integer geeft, die we vervolgens delen door een **double**, wat dus een **double** als resultaat geeft.
2. Eerst tellen we twee integers op, wat weer een integer geeft, vervolgens zetten we dit resultaat om naar een **double** en delen dit door een **int** wat dus een **double** geeft.
3. Eerst zetten **tempGisteren** om naar een **double**. Vervolgens tellen we een **double** met een **int** op, wat een double als tussenresultaat geeft. Dit delen we dan door een **int** wat een **double** finaal geeft.
4. Hetzelfde als de vorige stap, maar nu zetten we eerst **tempVandaag** om naar een **double**.



Merk op dat er een subtiel verschil is tussen volgende 2 lijnen code:

```
1 (double)(tempGisteren + tempVandaag) / 2; //geeft 22.5  
2 (double)((tempGisteren + tempVandaag) / 2); //geeft 22
```

In het eerste zullen we het resultaat van de som naar **double** omzetten. In het tweede, door de volgorde van berekeningen door de haakjes, zullen we de casting pas doen **na de deling** en zal dus 22 in plaats van 22.5 als resultaat geven.

4.2.4 Widening

Casting is niet nodig als je aan **widening** doet: een *kleiner* type in een *groter* type steken (met groter/kleiner wordt de geheugengrootte van het datatype bedoeld), als volgt:

```
1 int hoofdMeting;
2 double secundaireMeting;
3 hoofdMeting = 20;
4 secundaireMeting = hoofdMeting; //secundaireMeting krijgt 20.0
```

Deze code zal zonder problemen werken: secundaireMeting zal de waarde 20.0 bevatten. De inhoud van hoofdMeting wordt *verbreed* naar een **double**, eenvoudigweg door er een kommagetal van te maken.

Er gaat **geen** inhoud verloren echter. Je hoeft dus niet expliciet de casting-notatie zoals (**double**)hoofdMeting te doen, de computer ziet zelf dat hij de inhoud van hoofdMeting zonder dataverlies kan toekennen aan secundaireMeting en is dus niet bang dat hij later aangeklaagd zal worden.

Merk op dat je perfect casting hier mag gebruiken, maar daar de conversie impliciet zonder problemen kan plaatsvinden hoeft dit dus niet. Deze code is echter even juist (en soms een veilige gewoonte om te doen, better safe than sorry):

```
1 secundaireMeting = (double) hoofdMeting;
```

4.3 Conversie

Casting is de ‘oldschool’ manier van data omzetten die vooral zeer nuttig is daar deze compacte code geeft en ook werkt in andere C#-gerelateerde programmeertalen zoals C, C++ en Java.

Echter, .NET heeft ook ingebouwde conversie-methoden die je kunnen helpen om data van het ene type naar het andere te brengen. Het nadeel is dat ze iets meer typwerk (en dus meer code) vereisen dan bij casting.

Al deze methoden zitten binnen de **Convert**-bibliotheek van .NET.

Het gebruik hiervan is zeer eenvoudig. Enkele voorbeelden:

```
1 int getal = Convert.ToInt32(3.2); //double to int
2 double anderGetal = Convert.ToDouble(5); //int to double
3 bool isWaar = Convert.ToBoolean(1); //int to bool
4 int leeftijd = Convert.ToInt32("19"); //string to int
5 int andereLeeftijd = Convert.ToInt32(anderGetal); //double to int
```

Je plaatst tussen de ronde haakjes de variabele of literal die je wenst te converteren naar een ander type. Merk op dat naar een **int** converteren met `.ToInt32()` moet gebeuren. Om naar een **short** te converteren is dit met behulp van `.ToInt16()`.



Convert.ToBoolean verdient extra aandacht: Wanneer je een getal, eender welk, aan deze methode meegeeft zal deze altijd naar True geconverteerd worden. Enkel indien je 0 (als **int**) of 0.0 (als **double**) ingeef, dan krijg je False. In quasi alle andere gevallen krijg je True.

De conversie zal zelf zo goed mogelijk de data omzetten en dus indien nodig widening of narrowing toepassen. Zeker bij het omzetten van een string naar een ander type kijk je best steeds de documentatie na om te weten wat er intern juist zal gebeuren.¹

4.4 Parsing

Naast conversie en casting bestaat er ook nog **parsing**.

Parsing is anders dan conversie en casting. Parsing zal je in dit boek enkel nodig hebben om tekst(**string**) naar getallen om te zetten. Echter, intern zal bijna altijd een **Convert.ToBoolean** methode gebruikt worden indien je een **Parse** methode aanroeft.

Ieder ingebouwd datatype in C# heeft een **.Parse()** methode die je kan aanroepen om **strings om te zetten naar het gewenste type**.

Voorbeeld van parsing:

```
1 int numVal = Int32.Parse("-105");
2 Console.WriteLine(numVal);
```

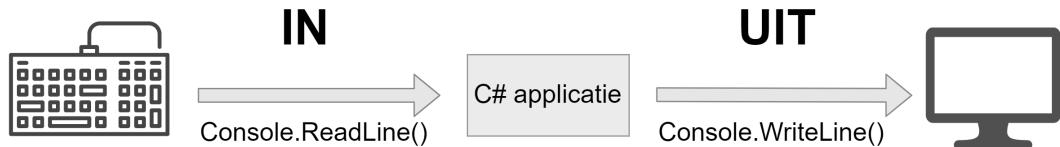
Gebruik parsing enkel wanneer je:

1. een **string** hebt waarvan je weet dat deze altijd van een specifieke vorm zal zijn die omgezet kan worden naar een ander datatype, bv. een **int**, dan kan je **Int32.Parse()** gebruiken.
2. input van de gebruiker vraagt (bv. via **Console.ReadLine()**) en niet 100% zeker bent dat deze een getal zal bevatten, gebruik dan **Int32.TryParse()** (meer info in de appendix).

¹Je kan alle conversie-mogelijkheden nalezen op msdn.microsoft.com/system.convert.

4.5 Invoer van de gebruiker verwerken

En applicatie die geen input van de gebruiker vergt kan even goed een screensaver zijn. We hebben reeds gezien hoe we met `Console.ReadLine()` de gebruiker tekst kunnen laten invoeren en die we dan vervolgens kunnen verwerken om bijvoorbeeld z'n naam op het scherm te tonen:



Figuur 4.2: Deze vereenvoudiging van de meeste van onze applicaties blijft gelden.

De uitdaging met `ReadLine` is dat deze ALTIJD een string teruggeeft:

```
1 string userInput = Console.ReadLine();
```

Dit mag dus niet: `int` `userInput` = `Console.ReadLine()`; en zal in een *conversion error* resulteren.

Willen we dat de gebruiker een getal invoert dan zal dit nog steeds als **string** moeten worden ingelezen. **Vervolgens zullen we dit vervolgens moeten converteren.**

Invoer van de gebruiker verwerken (dat een andere type dan **string** moet zijn) zal dus uit 3 stappen bestaan:

1. Input **uitlezen** met `Console.ReadLine()`.
2. Input **bewaren** in een **string** variabele.
3. De variabele **parsen** met de `Parse()` bibliotheek naar het gewenste type.

Stel dat we aan de gebruiker z'n gewicht vragen, dan moeten we dus doen:

```
1 Console.WriteLine("Geef je gewicht:");
2 string inputGewicht = Console.ReadLine();
3 double gewicht = double.Parse(inputGewicht);
```

Voorgaande code kan nog 1 lijtje sneller door `ReadLine` ogenblikkelijk als invoer aan de `Parse`-methode te geven:

```
1 Console.WriteLine("Geef je gewicht:");
2 double gewicht = double.Parse(Console.ReadLine());
```

4.5.1 Foutloze input

Voorgaande code veronderstelt dat de gebruiker géén fouten invoert². De conversie zal namelijk mislukken indien de gebruiker bijvoorbeeld `Ik weeg 10kg` invoert in plaats van `10,3`.

In de komende hoofdstukken mag je er altijd van uitgaan dat de gebruiker foutloze input geeft.

4.5.2 Kommagetallen in C#

Goed opletten nu.

De invoer van komaalgetallen door de gebruiker is afhankelijk van de landinstellingen van je besturingssysteem. Staat deze in Belgisch/Nederlands dan moet je komaalgetallen met een **komma** invoeren (`9,81`). Staat je computer in het Engels dan moet je een **punt** gebruiken (`9.81`).

Maar: **In je C# code moet je komaalgetallen literals altijd met een punt schrijven.** Dit is onafhankelijk van je taalinstellingen.

²En wat als je toch foute invoer wilt opvangen? Dan is `TryParse` je vriend. Meer informatie hierover in de appendix.

4.6 Berekeningen met System.Math

Een groot deel van je leven als ontwikkelaar zal bestaan uit het bewerken van variabelen in code. Meestal zullen die bewerkingen voorafgaan van berekeningen. De `System.Math` bibliotheek zal ons hier bij kunnen helpen. Zoals de naam al doet vermoeden staat deze bibliotheek voor *Mathematics*: wiskunde!

4.6.1 De Math-bibliotheek

De Math-bibliotheek bevat handige methoden voor een groot aantal typische wiskundige bewerkingen. Zaken die je er bijvoorbeeld in zal terugvinden:

- Sinus (`Sin`), cosinus (`Cos`), tangens (`Tan`), enz. berekenen aan de van de hoek (in radialen)
- Vierkantswortel (`Sqrt`) en macht (`Pow`) berekenen.
- Naar boven (`Ceiling`) of onder (`Floor`) afronden.
- Absolute (`Abs`) waarde berekenen.

Stel dat je de derde macht van een variabele getal wenst te berekenen. *Zonder* de Math-bibliotheek zou dat er zo uitzien:

```
1 double result = getal * getal * getal; //SLECHTE MANIER
```

Dit valt nog mee, maar wat als je 3 tot de zevende macht moest berekenen? Laten we eens kijken hoe Math ons kan helpen, dankzij de Pow methode (**Power**, Engels voor macht):

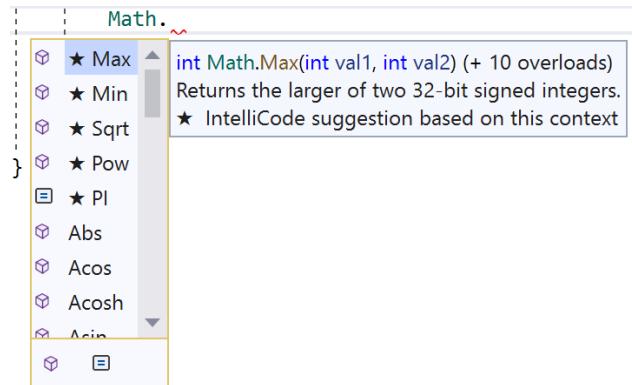
```
1 double result = Math.Pow(getal, 3);
```

Deze methode vereist twee parameters:

- De eerste is het grondtal.
- De tweede is de exponent (“tot de hoeveelste macht”).

4.6.1.1 De Math bibliotheek ontdekken

Als je in Visual Studio Math schrijft in je code, gevolgd door een punt (.) krijg je alles te zien wat de Math-bibliotheek kan doen:



Figuur 4.3: De sterretjes geven de meestgebruikte methoden in deze bibliotheek aan. Vervolgens verschijnen alle overige methoden, properties, enz. alfabetisch.

Een kubusje voor een naam wil zeggen dat het om een **Methode** gaat (zoals `Console.ReadLine()`). Een vierkantje met twee streepjes in zijn constanten (zoals `PI` en het getal van Euler (`e`)).

4.6.1.2 Methoden gebruiken

De meeste methoden zijn zeer makkelijk in gebruik en werken bijna allemaal op een soortgelijk manier. Meestal moet je 1 of meerdere parameters tussen de haken meegeven en het resultaat moet je altijd in een nieuwe variabele opvangen.

Enkele voorbeelden:

```
1 double sineHoekA = Math.Sin(345); //in radialen!
2 double derdeMachtVan20 = Math.Pow(20, 3);
3 double complexer = 3 + derdeMachtVan20 * Math.Round(sineHoekA);
```

Twijfel je over de werking van een methode, gebruik dan de help als volgt:

1. Schrijf de Methode zonder parameters. Bijvoorbeeld `Math.Pow()` (je mag de foutboodschap negeren).
2. Plaats je cursor op Pow.
3. Druk op F1 op je toetsenbord.
4. Je krijgt nu de help-files te zien van deze methode.
5. In hoofdstuk 7 leg ik uit hoe je die help-files moet lezen.

4.6.1.3 PI

Ook het getal Pi (3.141...) is beschikbaar in de Math-bibliotheek. Het witte icoontje voor PI bij Intellisense toont aan dat het hier om een *field* gaat: een eenvoudige variabele met een specifieke waarde. In dit geval gaat het zelfs om een **const** field, met de waarde van Pi van het type double.

```
1 public const double PI;
```

Je kan deze als volgt gebruiken in berekeningen zoals

```
1 double straal = 5.5;
2 double omtrek = Math.PI * 2 * straal;
```

4.6.2 Bereik in code weten

Het bereik van datatypes ligt weliswaar vast (zie hoofdstuk 2). Maar het is nuttig om weten dat deze ook in de compiler gekend is. Ieder datatype heeft een aantal ingebouwde zaken die je kan gebruiken om onder andere de maximum en minimum-waarde van een datatype te gebruiken. Volgende voorbeeld toont hoe dit kan:

```
1 Console.WriteLine("Het bereik van het type double is:");
2 Console.WriteLine($"{double.MinValue} tot {double.MaxValue}.");
```

Dit geeft op het scherm:

Het bereik van het type **double** is: -1.7976931348623157*10^308 tot 1.7976931348623157E*10^308.

Je kan met andere woorden met **int.MaxValue** en **int.MinValue** het minimum- en maximumbereik van het type **int** verkrijgen.

Wil je dit van een **double**, dan gebruik je **double.MaxValue** enz.

Trouwens, zelfs oneindig is beschikbaar bij kommagetallen als **.PositiveInfinity** en **.NegativeInfinity**.

4.7 Random getallen genereren

Willekeurige (*random*) getallen genereren in je code kan leuk zijn om de gebruiker een interactievere ervaring te geven. Beeld je in dat je monsters steeds dezelfde weg zouden bewandelen of dat er steeds op hetzelfde tijdstip een orkaan op je stad neerdwaalt. Of wat de denken van een programma dat steeds dezelfde wiskunde opgaven genereert? **SAAI!**

4.7.1 Random generator

De Random-bibliotheek laat je toe om willekeurige gehele en komma-getallen te genereren. Je moet hiervoor twee zaken doen:

1. Maak **eenmalig** een Random-generator object aan.
2. Roep de Next methode aan op dit object telkens je een nieuw willekeurig getal nodig hebt.

Als volgt:

```
1 Random randomGenerator = new Random();
2 int mijnLeeftijd = randomGenerator.Next();
```

De eerste stap dien je maar 1 keer te doen. De naam die je het generatorobject geeft (hier randomGenerator) mag je kiezen. Dit is een variabele en moet dus aan de identifier regels voldoen.

Vanaf nu kan je telkens aan het generatorobject een nieuw getal vragen door middel van de Next-methode.

Volgende code toont bijvoorbeeld 3 random getallen op het scherm:

```
1 Random myGen = new Random();
2
3 int getal1 = myGen.Next();
4 int getal2 = myGen.Next();
5 int getal3 = myGen.Next();
6 Console.WriteLine(getal1);
7 Console.WriteLine(getal2);
8 Console.WriteLine(getal3);
```

Uiteraard mag dit ook

```
1 Console.WriteLine(myGen.Next());
2 Console.WriteLine($"Nog een getal: {myGen.Next()}");
```



De **new Random()** code is iets wat in hoofdstuk 9 en verder volledig uit de doeken zal gedaan worden. Lig er dus nog niet van wakker.

4.7.1.1 Next mogelijkheden

Je kan de `Next` methode ook 2 parameters meegeven, namelijk de grenzen waarbinnen het getal moet gegenereerd worden. De tweede parameter is exclusief dit getal zelf. Wil je dus een willekeurig geheel getal tot en met 10 dan schrijf je 11, niet 10, als tweede parameter:

Enkele voorbeelden:

```
1 Random someGenerator = new Random();
2 int a = someGenerator.Next(0,11); //getal tussen 0 tot en met 10
3 int b = someGenerator.Next(55,100); //getal tussen 55 tot en met 99
4 int c = someGenerator.Next(0,b); //getal tussen 0 tot en met (b-1)
```

4.7.1.2 Genereer kommagetallen met `NextDouble`

Met de `NextDouble` methode kan je kommagetallen genereren tussen 0.0 en 1.0 (1.0 zal niet gegenereerd worden).

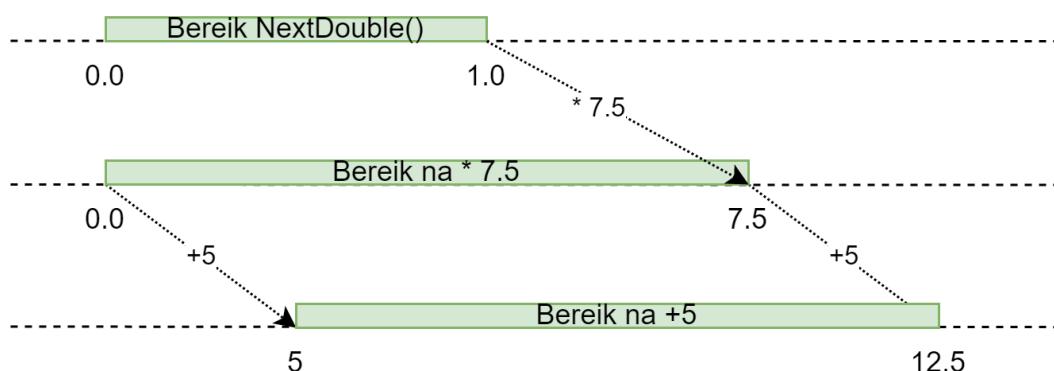
Wil je een groter kommagetal dan zal je dit gegenereerde getal moeten vermenigvuldigen naar de range die je nodig hebt. Stel dat je een getal tussen 0.0 en 10.0 nodig hebt, dan schrijf je:

```
1 Random myRan = new Random();
2 double randomGetal = myRan.NextDouble() * 10.0;
```

Je vermenigvuldigt eenvoudigweg je gegenereerde getal met het bereik dat je wenst (10.0 in dit geval)

En wat als je een kommagetal tussen 5.0 en 12.5 wenst? Als volgt:

```
1 Random myRan = new Random();
2 double randomGetal = 5.0 + (myRan.NextDouble() * 7.5);
```



Figuur 4.4: Visualisatie van hoe je het bereik van Random kan aanpassen (rechte is niet op schaal)

Je bereik is 7.5, namelijk $12.5 - 5.0$ en vermenigvuldig je het resultaat van je generator hiermee. Vervolgens verschuif je dat bereik naar 5 en verder door er 5 bij op te tellen. Merk op dat we de volgorde van berekeningen *sturen* met onze ronde haakjes.



“Help! Ik krijg steeds dezelfde random getallen? Wat nu?”

Wel wel, wie we hier hebben. Werkt je Random generator niet naar behoren? Wil je het ding in de vuilbak gooien omdat het niet zo willekeurig lijkt te werken als je hoopte? Gelukkig ben ik er! Zet je helm dus op en luister.

Wanneer je twee Random objecten aanmaakt op quasi hetzelfde tijdstip in je code, dan zullen deze twee generators ook dezelfde getallen genereren:

```
1 Random a = new Random();
2 Random b = new Random(); //Slecht idee!
3 Console.WriteLine(a.Next());
4 Console.WriteLine(b.Next());
```

De Random bibliotheek gebruikt de tijd als een soort “willekeurig” startpunt (de tijd is de zogenaamde *seed*). Het is namelijk een **pseudo-willekeurige getal generator**.

Dit is de reden waarom je in je code steeds maar **1 Random generator** mag aanmaken! Er zijn weinig redenen om er meerdere aan te maken. Bovenstaande code is dus niet aan te raden. Je schrijft beter:

```
1 Random a = new Random();
2 Console.WriteLine(a.Next());
3 Console.WriteLine(a.Next());
```

Wil je toch dezelfde willekeurige reeks getallen na elkaar genereren telkens je je programma opstart (bijvoorbeeld om je code te testen met steeds dezelfde reeks getallen) dan kan je bij het aanmaken van je generator ook een parameter meegeven die als seed zal werken.

In het volgende voorbeeld zal generator a steeds dezelfde reeks willekeurige getallen genereren, telkens je je programma uitvoert. De waarde die je meegeeft moet uiteraard niet 666 zijn. Ieder getal dat je meegeeft is een andere seed:

```
1 Random a = new Random(666);
2 Console.WriteLine(a.Next());
```

4.8 Debuggen

“Joepie!! M’n code werkt!” Je ontkurkt de champagne/bier/melk/frisdrank/water, doet een *Fortnite* danske en laat je programma door duizenden, neen... miljoenen gebruikers ontdekken. Nog geen uur later staat er een meute met hooivorken en toortsen voor je kantoor. Helaas, er zaten nog “een paar bugs” in je code...

Tijd dus om je **debugger** boven te halen en die (logische) fouten uit je code te halen.

4.8.1 Logische fouten vs C# fouten

Code die compileert is enkel code die foutloos geschreven is volgens de C# afspraken qua grammatica en syntax. De code zal met andere woorden gecompileerd worden, maar wat er daarna gebeurd is volledig afhankelijk van wat juist de betekenis is van wat je hebt geschreven.

Volgend algoritme bijvoorbeeld is perfecte Nederlandstalige code en zal dus door een fictieve compiler kunnen gecompileerd worden. Het vervolgens uitvoeren is echter niet aan te raden (natrium en water samen geeft een stevige exotherme reactie):

- 1 Neem natrium
- 2 Neem water
- 3 Voeg beide samen

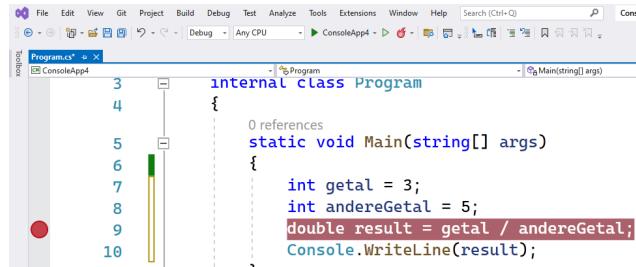
Dit is dus een logische fout: oftewel een bug (in dit geval zal dit trouwens een stevige explosie veroorzaken).

4.8.2 Debuggen met Visual Studio

Standaard wanneer je je code uitvoert met de grote groene playknop start jouw programma in zogenaamde “debug modus”. Dit laat je toe om je code ten allen tijde te onderbreken en naar de huidige staat van je programma te kijken. Je kan dan bijvoorbeeld onderzoeken wat de waarden van bepaalde variabelen zijn op dat moment en of die wel correct zijn. Dit is bughunting en zal je héél vaak doen in je programmeer-carrière.

Om dit te doen moet je één of meer **breakpoints** in je code plaatsen. Een breakpoint zet je aan een lijn code. Wanneer je programma dan aan deze lijn komt tijdens de uitvoer zal de debugger alles pauzeren.

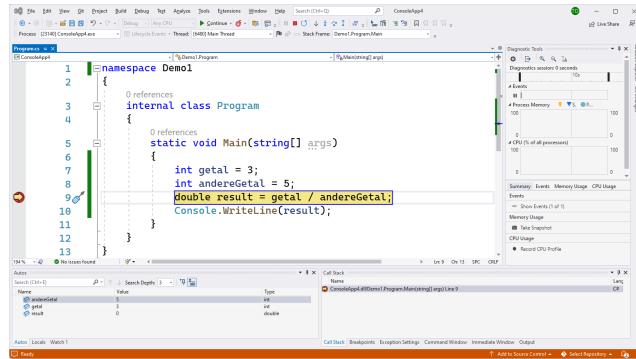
Een breakpoint plaats je door in VS op het grijze gedeelte links van de lijn code te klikken. Als alles goed gaat verschijnt er dan een grote rode “breakpointbol”:



Figuur 4.5: De rode bol! Merk op dat je er ook meer dan één mag plaatsen. Iedere bol stelt een breakpoint voor waar de uitvoer zal stoppen als de code hier aankomt (wat later niet altijd het geval zal zijn wanneer we met loops en methoden leren programmeren).

In bovenstaande figuur plaatsen we een breakpoint aan lijn 11. De code uitvoer zal dus nog wel lijn 10 uitvoeren, maar niet lijn 11.

Als je nu je project uitvoert zal de code pauzeren aan die lijn en zal VS in “debug modus” openspringen wat er vervolgens als volgt uit ziet:



Figuur 4.6: Als je niet alle debugknoppen ziet kan je deze ook aanroepen via het “Debug” in het menu bovenaan.

In dit “nieuwe” scherm zijn er momenteel 2 belangrijke delen:

- Onderaan zie je de **autos** en **locals**. In deze tabs kan je de waarden van iedere variabele in je huidige code zien op het moment van pauzeren. Ideaal om te onderzoeken waarom een bepaalde berekening of expressie niet doet wat ze moet doen.
- Bovenaan zijn enkele debug-knoppen verschenen. Deze licht ik in de volgende sectie toe.
- Voorts kan je in debug-modus met je muis over eender welke variabele of expressie hoveren om het resultaat van dat element te bekijken:

```
▶ double result = getal / andereGetal;
    Console.WriteLine(result);
    getal / andereGetal | 0 ->
```

Figuur 4.7: Kan je trouwens verklaren waarom deze deling 0 geeft en niet 1.667?

4.8.3 Door je code steppen

Wanneer je gepauzeerd bent kan je de nieuw verschenen debug-knoppen bovenaan VS gebruiken om het verdere verloop te bepalen:



Figuur 4.8: Debug knoppen.

Ik bespreek hier de knoppen die je zeker zal nodig hebben:

- De **continue** knop is logisch: hier op klikken zal je programma terug voortzetten vanaf het breakpoint waar je gepauzeerd bent. Het zal vervolgens verder gaan tot het weer een breakpoint bereikt of wanneer het einde van het programma wordt bereikt.
- De **step in** knop zullen we in hoofdstuk 7 toelichten daar deze knop je toelaat om in een methode te springen.
- De **rode stop** knop gebruik je indien je niet verder wilt debuggen en ogenblikkelijk terug je code wilt aanpassen.
- De **step-over** knop (het gebogen pijltje) is een belangrijke knop. Deze zal je code één lijn code verder uitvoeren en dan weer pauzeren. Het laat je dus toe om letterlijk doorheen je code te *stappen*. Je kan dit doen om de flow van je programma te bekijken (zie volgende hoofdstukken) en om te zien hoe bepaalde variabelen evolueren doorheen je code.



Pfft. Debuggen. Waarom moet ik me daar nu mee bezig houden?

Even je oren open zetten aub, ik ga iets roepen: “**Debugging is een ESSENTIELLE SKILL!!!**”. Ik laat mijn metselaars ook geen huizen bouwen zonder dat ze ooit een truweel hebben vastgepakt. Een programmeur die niet kan debuggen... is als een vis die niet kan zwemmen!

Zorg dus dat je vlot breakpoints kunt plaatsen om zo tijdens de uitvoer te pauzeren om de inhoud van je variabelen te bekijken (via het watch-venster). Gebruik vervolgens de “step”-buttons om door je code te ‘stappen’, lijn per lijn.

Is that all?! NEEN! Een goede programmeur zal telkens **eerst voorspellen** wat er gaat gebeuren: welke waarden zullen de variabelen hebben als ik naar de volgende lijn ga? Wat gaan er op het scherm komen? enz. Als je dan vervolgens naar de volgende lijn of breakpoint gaat en er gebeuren dingen die je niet voorspeld had, dan is de kans groot dat je een bug hebt gevonden.

De grootste fout die je kunt doen is gewoon door je code te “steppen” en hopen dat de bug magisch zal tevoorschijn komen. Nee, zo werkt het dus niet. Je moet actief mee denken of dat je programma effectief werkt zoals je zelfs bedoeld had.

Dit geldt trouwens ook wanneer je niet aan het debuggen bent, maar gewoon je programma uitvoert om het te testen. Eigenlijk ben je dan ook aan het debuggen. Ook dan moet je voorspellen wat het eindresultaat zal zijn en of dit overeen komt met wat er op het scherm gebeurt. **Wees kritisch!**

5 Beslissingen

Nu we de elementaire zaken van C# en VS kennen is het tijd om onze programma's wat interessanter te maken. De programma's die we tot nu toe hebben ontwikkeld waren steeds lineair van opbouw. Ze werden lijn per lijn uitgevoerd, van start tot einde, zonder de mogelijkheid om de **programmaworkflow** aan te passen. Het programma doorliep de lijnen code braaf na elkaar en wanneer deze aan het einde kwam sloot het zich af.

Onze programma's waren met andere woorden niet meer dan een eenvoudige lijst van opdrachten. Je kan het vergelijken met een lijst die je over hoe je een brood moet kopen:

- 1 Neem geld uit spaarpot
- 2 Wandel naar de bakker om de hoek
- 3 Vraag om een brood
- 4 Krijg het brood
- 5 Betaal het geld aan de bakker
- 6 Keer huiswaarts
- 7 Smullen maar

Alhoewel dit algoritme redelijk duidelijk is en goed zal werken, zal de realiteit echter zelden zo rechtlijnig zijn. Van zodra 1 van de stappen faalt (bijvoorbeeld omdat de bakker toe is) zal ook de rest van het algoritme niet meer werken.

Een beter algoritme zal afhankelijk van de omstandigheden (bakker gesloten, geen geld meer, enz.) andere stappen ondernemen. **Het programma zal beslissingen maken gebaseerd op keuzes** doorheen het programma:

- 1 Neem geld uit spaarpot
- 2 Geld op? Stop dan hier, anders: ga verder
- 3 Wandel naar de bakker om de hoek
- 4 Bakker toe? Stop dan hier, anders: ga verder
- 5 Vraag om een brood
- 6 Krijg het brood
- 7 Betaal het geld aan de bakker
- 8 Als je honger hebt, sla dan volgende lijn over, anders: ga verder
- 9 Keer huiswaarts
- 10 Smullen maar

5.1 Relationale en logische operators

Om beslissingen te kunnen nemen in C# hebben we een nieuw soort operators nodig. Operators waarmee we kunnen testen of iets waar of niet waar is. Met C# kun je een actie uitvoeren als een voorwaarde waar is, en iets anders doen (of een stap overslaan) als de voorwaarde niet waar is.

Dit doen we met de zogenaamde **relationele operators** en **logische operators**.

5.1.1 Booleaanse expressies

Een booleaanse expressie is een stuk C# code dat een **bool** als resultaat zal geven. De logische en relationele operators die ik hierna bespreek zijn operators die een **bool** teruggeven. Ze zijn zogenaamde test-operators: ze testen of iets waar is of niet.

5.1.2 Relationale operators

Relationele operators zijn het hart van booleaanse expressies. En guess what, je kent die al van uit het lager onderwijs. Enkel de “gelijk aan” ziet er iets anders uit dan we gewoon zijn uit onze lessen wiskunde:

Operator	Betekenis
>	groter dan
<	kleiner dan
==	gelijk aan
!=	niet gelijk aan
<=	kleiner dan of gelijk aan
>=	groter dan of gelijk aan

Deze operators hebben steeds twee operanden nodig en geven **een bool als resultaat terug**. Beide operanden **moeten van hetzelfde datatype zijn**. Je kan geen appelen met peren vergelijken!

Daar dit operators zijn kan je deze dus gebruiken in eender welke expressie. Het resultaat van de expressie `12 > 6` zal **true** als resultaat hebben daar 12 groter is dan 6. Eenvoudig toch.



We weten al dat je het resultaat van een expressie altijd in een variabele kunt bewaren. Ook bij het gebruik van relationele operators kan dat dus:

```
1 bool isKleiner = 65 > 67 ;
2 Console.WriteLine(isKleiner);
```

Er zal **false** als output op het scherm verschijnen.



Er is een groot verschil tussen de `=` operator en de `==` operator. De eerste is de toekenningsoperator en zal de rechtse operand aan de linkse operand toewijzen. De tweede zal de linkse met de rechtse operand op gelijkheid vergelijken en een **bool** teruggeven.

5.1.3 Logische operators

Vaak wil je meer complexe keuzes maken, zoals : “ga verder indien je hongerig bent **EN** je genoeg geld **bijhebt**”. Dit doen we met de zogenaamde **logische operators**.

Er zijn 3 (boolaans) operators die je hiervoor kunt gebruiken:

- `&& (EN)` : Geeft enkel **true** als beide operanden **true** zijn
- `|| (OF)` : Geeft **true** indien minstens 1 operand **true** is
- `! (NIET)` : Inverteert de waarde van de expressie (**true** wordt **false** en omgekeerd)

De logische operators geven ook steeds een **bool** terug **maar verwachten enkel operanden van het type bool**. Als je dus schrijft **true || false** zal het resultaat **true** zijn.

De **EN** en **OF** operators verwachten 2 operanden. Maar de **NIET**-operator verwacht maar 1 operand.

Aangezien onze relationele operators **bool** als resultaat geven, kunnen we dus de uitvoer van deze operators gebruiken als operanden voor de logische operators. We gebruiken hierbij haakjes om zeker de volgorde juist te krijgen:

```
1 bool result = (4 < 6) && ("ja" == "nee");
```

De haakjes zorgen ervoor dat eerste die delen worden berekend. Voorgaande zal dus in een tussenstap (die jij niet ziet) tijdens de uitvoer er als volgt uitzien:

```
1 bool result = true && false;
```

Vervolgens wordt dan de logische EN getest en krijgen we finaal **false** in **result**.

5.1.3.1 Niet-operator

Je kan de niet-operator voor een expressie zetten om het resultaat van de expressie te inverteren. Bijvoorbeeld:

```
1 bool result = !(0==2)
```

Eerst wordt weer het resultaat tussen de haakjes berekend. Dit geeft **false** (daar 0 niet gelijk is aan 2). Vervolgens passen we de NIET-operator toe op dit resultaat en zal er dus **true** bewaard worden.

Merk op dat we deze code ook kunnen schrijven als:

```
1 bool result = (0!=2)
```



Alhoewel we voorgaande ook zonder haakjes kunnen schrijven, raad ik dit af. Haakjes zorgen ervoor dat je code leesbaarder wordt. Maar nog belangrijker: het maakt de volgorde van bewerkingen expliciter. Als je niet zeker weet welke operator voorrang heeft, kun je haakjes gebruiken om de juiste volgorde af te dwingen. Dit helpt je om logische fouten te voorkomen die kunnen ontstaan door verkeerde veronderstellingen.

5.1.3.2 Volgorde van bewerkingen

De volgorde van bewerkingen wordt nu belangrijk¹! In C# hebben de operatoren die we al kennen volgende volgorde:

1. **Logische NIET: !**
2. Delen en vermenigvuldigen: *, /, %
3. Optellen en aftrekken: +, -
4. **Relationele operators: <, <=, >, >=**
5. **Gelijkheid: ==, !=**
6. **Logische EN: &&**
7. **Logische OF: ||**

Wil je deze volgorde dus veranderen in een samengestelde expressie, dan moet je gebruik maken van haakjes.

¹Bekijk zeker de tabel op docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators waar de volgorde van alle operators wordt beschreven.

5.1.4 Test jezelf

Wat zal de uitkomst zijn van volgende expressies?

- `3>2`
- `4 !=4`
- `4<5 && 4<3`
- `"a"=="A" || 4>=3`
- `(3==3 && 2<1) || 5!=4`
- `!(4<=3)`
- `true || false`
- `!true && false`

5.2 If

De **if** (*als*) uitdrukking is één van de meest elementaire uitdrukkingen in een programmeertaal. Het laat ons toe vertakkingen in onze programmaflow in te bouwen. Ze laat toe om “als dit waar is doe dan dat”-beslissingen te maken.

De syntax is als volgt:

```
1 if (boolaalse expressie)
2 {
3     //deze code wordt uitgevoerd indien
4     //de booleaarse expressie true is
5 }
```

Enkel indien de booleaarse expressie **waar** is (**true**), zal de code binnen de accolades van het **if**-blok uitgevoerd worden. Indien de expressie niet waar is (**false**) dan wordt het blok overgeslagen en gaat het programma verder met de code eronder.

Een voorbeeld:

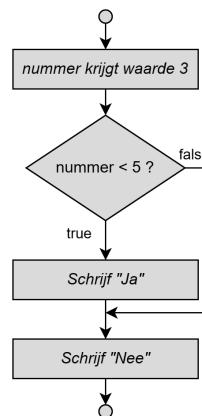
```
1 int nummer = 3;
2 if ( nummer < 5 )
3 {
4     Console.WriteLine ("Ja");
5 }
6 Console.WriteLine("Nee");
```

De uitvoer van dit programma zal zijn:

```
1 Ja
2 Nee
```

Indien nummer groter of gelijk aan 5 was dan zou er enkel Nee op het scherm zijn verschenen. De lijn `Console.WriteLine ("Nee") ;` zal sowieso uitgevoerd worden zoals je ook kan zien in de flowchart op de volgende pagina.²

²Code2flow.com is een handige tool om je reeds geschreven C# code om te zetten naar een flowchart. Het kan je helpen om vreemde bugs te ontdekken. Uiteraard is de eerste stap debuggen en door je code *steppen*: vaak zal je ogenblikkelijk zien waar je code verkeerd loopt.

**Figuur 5.1:** De flowchart van het eerste voorbeeld

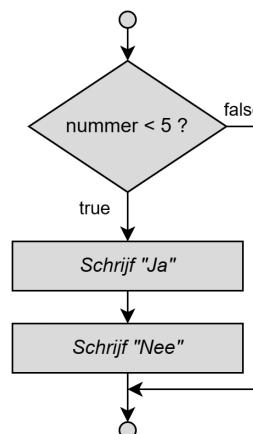
5.2.1 if met een block

Het is aangeraden om steeds na de if-expressie met accolades te werken. Dit zorgt ervoor dat alle code tussen het block (de accolades) zal uitgevoerd worden indien de booleaanse expressie waar was. **Gebruik je geen accolades dan zal enkel de eerste lijn na de if uitgevoerd worden bij true.**

Een voorbeeld:

```

1  if ( nummer < 5 )
2  {
3      Console.WriteLine ("Ja");
4      Console.WriteLine ("Nee");
5  }
  
```

**Figuur 5.2:** Accolades zijn duidelijk belangrijk.

5.2.2 Veelgemaakte fouten met if



Voorman hier! Je hebt me gemist. Ik merk het. Het ging goed de laatste tijd.

*Maar nu wordt het tijd dat ik je weer even wakker schud want de code die je nu gaat bouwen kan érg vreemde gedragingen krijgen als je niet goed oplet. Luister daarom even naar deze lijst van veel gemaakte fouten wanneer je met **if** begint te werken.*

5.2.2.1 Appelen en peren vergelijken

De types in je booleaanse expressie moeten steeds vergelijkbaar zijn. Volgende code zal niet compileren:

```
1 if( "4" > 3)
```

daar we hier een **string** met een **int** vergelijken. Meestal moeten dus beide operanden bij een relationele operator van hetzelfde type zijn (of er moet een implicietie, automatische casting kunnen gebeuren).

5.2.2.2 Accolades vergeten

Accolades vergeten plaatsen om een codeblock aan te duiden is een typische fout. Wanneer je bijvoorbeeld Python hebt geleerd, dan zou je verwachten dat je code zodanig kan uitlijnen (met tabs of spaties) om code bij een **if** te groeperen in een block. Wat dus niet zo is. **Je code uitlijnen heeft in C# géén invloed op de programflow.**

Gebruik je dus geen accolades dan zal enkel de eerste lijn na de **if** zal uitgevoerd worden indien **true**. Gebruiken we de **if** met block van daarnet maar zonder accolades dan zal de laatste lijn altijd uitgevoerd worden ongeacht de **if**:

```
1 if( tijd < 20 )
2     Console.WriteLine ("Doe zo voort.");
3     Console.WriteLine ("Je bent er bijna!"); //verschijnt altijd op
                                         scherm
```

Deze code zal dus 2 mogelijke outputs op het scherm geven. Indien de *tijd groter of gelijk is aan 20* dan krijgen we volgende output:

```
1 Je bent er bijna!
```

Maar indien de `tijd` kleiner is dan 20 krijgen we:

```
1 Doe zo voort.  
2 Je bent er bijna!
```

Dit is uiteraard niet de output die we verwachten. We willen de motiverende boodschappen (beide zinnen) enkel tonen indien de gebruiker nog tijd heeft. De juiste oplossing is:

```
1 if ( tijd < 20 )  
2 {  
3     Console.WriteLine ("Doe zo voort.");  
4     Console.WriteLine ("Je bent er bijna!");  
5 }
```

5.2.2.3 Een puntkomma plaatsen na de booleanse expressie

Dit zal ervoor zorgen dat er eigenlijk geen codeblock bij de `if` hoort en je dus een nietszeggende `if` hebt geschreven. De code na het puntkomma zal uitgevoerd worden ongeacht de `if`:

```
1 if ( naam == "neo" );  
2 {  
3     Console.WriteLine ("Take the red pill?");  
4     Console.WriteLine ("Or the blue pill?");  
5 }
```

De uitvoer van voorgaande zal altijd de volgende zijn, ongeacht of de gebruikersnaam gelijk is aan “neo”:

```
1 Take the red pill?  
2 Or the blue pill?
```

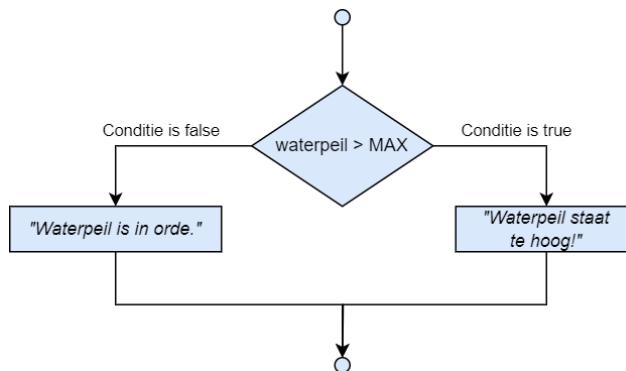
Indien de naam gelijk is aan “neo” dan zal de code *tussen de if en het kommapunt op lijn 1* uitgevoerd worden. Kortom, er wordt niets gedaan (daar hier geen code staat). Het block erachter dat de 2 zinnen op het scherm zet wordt altijd uitgevoerd.

5.2.3 If/else

Met “if - else” kunnen we niet enkel zeggen welke code moet uitgevoerd worden als de conditie waar is **maar ook specifieke code moet uitgevoerd indien de conditie niet waar is**. Volgend voorbeeld geeft een typisch gebruik van een “if - else” structuur om 2 waarden met elkaar te vergelijken:

```

1 const int waterpeil = 10;
2 int MAX = 5;
3
4 if ( waterpeil > MAX )
5 {
6     Console.WriteLine ("Waterpeil staat te hoog!");
7 }
8 else
9 {
10    Console.WriteLine ("Waterpeil is in orde.");
11 }
```



Figuur 5.3: Flowchart van bovenstaande code.



Een veel gemaakte fout is bij de **else** sectie ook een booleaanse expressie plaatsen. Dit kan niet: de **else** sectie zal gewoon uitgevoerd worden indien de **if** sectie NIET uitgevoerd werd. Volgende code MAG DUS NIET:

```

1 if(a > b)
2 {...}
3 else (a <= b) //<FOUT!
4 {...}
```

5.2.4 If - else if

Met een “if - else if” constructie kunnen we meerdere criteria opgeven die waar/niet waar moeten zijn voor een bepaald stukje code kan uitgevoerd worden. Sowieso begint men steeds met een **if**. Als men vervolgens een **else if** plaatst dan zal de expressie van deze **else if** getest worden enkel en alleen als de eerste expressie niet waar was. Als de expressie van deze **else if** wel waar is zal de bijhorende code uitgevoerd worden, zo niet wordt deze overgeslagen.

Een voorbeeld:

```
1 int x = 9;
2
3 if (x == 10)
4 {
5     Console.WriteLine ("x is 10");
6 }
7 else if (x == 9)
8 {
9     Console.WriteLine ("x is 9");
10}
11 else if (x == 8)
12 {
13     Console.WriteLine ("x is 8");
14}
```

Voorts mag men ook steeds nog afsluiten met een finale **else** die zal uitgevoerd worden indien geen enkele andere expressie ervoor waar bleek te zijn:

```
1 if(x>100)
2 {
3     Console.WriteLine("Groter dan 100");
4 }
5 else if(x>10)
6 {
7     Console.WriteLine("Groter dan 10");
8 }
9 else
10 {
11     Console.WriteLine("Getal kleiner dan of gelijk 10");
12 }
```



De volgorde van opeenvolgende “if - else if - else” tests is uiterst belangrijk. Als we in de voorgaande code de volgorde van de twee tests omdraaien, zal het tweede blok ($x > 100$) nooit worden bereikt.

Logisch: neem een getal groter dan 100 en laat het door onderstaande code lopen. Stel, we nemen 110. Al bij de eerste test ($x > 10$) is deze **true** en verschijnt er dus “Groter dan 10”. Alle andere tests worden daarna niet meer gedaan en de code gaat verder na het **else**-blok:

```
1  if(x>10)
2  {
3      Console.WriteLine("Groter dan 10");
4  }
5  else if(x>100)
6  {
7      Console.WriteLine("Groter dan 100");
8  }
9  else
10 //...
```



Hoe minder tests de computer moet doen, hoe meer performant de code zal uitgevoerd worden. Voor complexe applicaties die bijvoorbeeld in realtime veel berekeningen moeten doen kan het dus een gigantische invloed hebben of een reeks “if - else if else” testen vlot wordt doorlopen. Het is dan ook een goede gewoonte - indien de logica van het algoritme het toelaat - om de meest voorkomende test bovenaan te plaatsen.

Ditzelfde geldt ook binnen een test zelf wanneer we met logische operators werken. Deze worden altijd volgens de regels van de volgorde van berekeningen uitgevoerd. Volgende test wordt van links naar rechts uitgevoerd:

```
1  x > 100 && a != "stop"
```

Omdat beide operanden van de EN-operatie **true** moeten zijn om een juiste test te krijgen, zal de computer de test automatisch stoppen indien reeds de linkse operand ($x > 100$) niet waar is. Bij dit soort tests probeer je dus ervoor te zorgen dat de tests die het minste kans op slagen hebben (of beter: het vaakst niet zal slagen) eerst te laten testen, zodat de computer geen onnodige extra tests doet.

5.2.5 Nesting

We kunnen met behulp van *nesting* ook complexere programma flows maken. Nesting wil zeggen dat we meerdere codeblokken in elkaar plaatsen. Hierbij gebruiken we de accolades om het blok code aan te duiden dat bij een “if - else if - else” hoort. Binnen dit blok kunnen nu echter opnieuw beslissingsstructuren worden aangemaakt.

Volgende voorbeeld toont dit aan. We zien hoe nesting wordt toegepast in het else gedeelte **else**. Bekijk wat er gebeurt als je dokterVanWacht aan iets anders gelijkstelt dan een lege string:

```

1 const double MAX_TEMP = 40;
2 double huidigeTemperatuur = 36.5;
3 string dokterVanWacht = "";
4
5 if (huidigeTemperatuur < MAX_TEMP)
6 {
7     Console.WriteLine("Temperatuur normaal");
8 }
9 else
10 {
11     Console.WriteLine("Temperatuur te hoog!");
12     if (dokterVanWacht == "")
13     {
14         Console.WriteLine("Oei oei! Geen dokter van wacht!");
15     }
16     else
17     {
18         Console.WriteLine($"{dokterVanWacht} gecontacteerd");
19     }
20 }
```

5.2.6 Gebruik relationele en logische operators

We kunnen ook meerdere booleaanse expressie combineren zodat we complexere uitdrukkingen kunnen maken. Stel dat we een if nodig hebben waar enkel *ingegaan* mag worden indien de leeftijd van een gebruiker hoger is dan 18 EN hij heeft een identiteitskaart bij. We kunnen dergelijke samengestelde expressies schrijven gebruik makend van de **logische operators**.

Volgende code toont het gebruik hiervan:

```

1 if( leeftijd > 18 && heeftIdentiteitskaart == true)
2 {
3     Console.WriteLine("Welkom");
4 }
5 else
6 {
7     Console.WriteLine("Niet toegelaten!");
8 }
```



*Laat deze tiental bladzijden uitleg je niet de indruk geven dat code schrijven met **if**-structuren een eenvoudige job is. Vergelijk het met van je pa leren hoe je met pijl en boog moet jagen, wat vlekkeloos gaat op een stilstaande schijf, tot je in het bos voor een mammoet staat die op je komt afgestormd. Da's andere kak hé?*

Het is dan ook aangeraden om, zeker in het begin, om steeds een flowchart te tekenen van wat je juist wilt bereiken. Dit zal je helpen om je code op een juiste manier op te bouwen (denk maar aan nesting en het plaatsen van meerdere “if -else” structuren in of na elkaar). Bezint eer ge begint.

5.3 Scope van variabelen

De locatie waar je een variabele aanmaakt bepaalt de **scope** van de variabele. Binnen deze scope zal een variabele gebruikt kunnen worden door andere code. Je kan de scope vergelijken als verschillende kamers in een gebouw. Variabelen die zij aangemaakt in een kamer zijn enkel in die kamer bruikbaar.

Eenvoudig gezegd zullen steeds de **omliggende accolades de scope van de variabele bepalen**. Indien je de variabele dus buiten die accolades nodig hebt dan heb je een probleem: de variabele is enkel bereikbaar binnen de accolades vanaf het punt in de code waarin het werd gedeclareerd.

Zeker wanneer je begint met **if**, loops, methoden, enz. zal de scope belangrijk zijn: deze code-constructies gebruiken steeds accolades om codeblocks aan te tonen. Een variabele die je dus binnen een if-blok aanmaakt zal enkel binnen dit blok bestaan, niet erbuiten.

Volgende voorbeeld toont bijvoorbeeld de scope van de variabele **getal**:

```

1 if( iLoveCSharp == true)
2 {
3     Console.WriteLine("Hoeveel punten op 10 geef je C#?"):
4     int getal ; //Start scope getal
5     getal = int.Parse(Console.ReadLine());
6 } // einde scope getal
7
8 Console.WriteLine(getal); // FOUT! getal niet in deze scope

```

De variabele **getal** wordt aangemaakt tussen de accolades van de **if** en “verdwijnt” dus van zodra we die *kamer verlaten* (laatste accolade).

Wil je dus **getal** ook nog buiten de **if** gebruiken zal je je code moeten herschrijven zodat **getal** VOOR de **if** wordt aangemaakt. Nu is de scope van variabele groter: daar we steeds naar de *omliggende accolades* moeten kijken. In dit geval bepalen dus de accolades op lijn 1 en 9 de scope:

```

1 {
2     int getal = 0 ; //Start scope getal
3     if( iLoveCSharp == true)
4     {
5         Console.WriteLine("Hoeveel punten op 10 geef je C#?"):
6         getal = int.Parse(Console.ReadLine());
7     }
8     Console.WriteLine(getal);
9 } // einde scope getal

```

De buitenste accolades zetten we er even om de scope te benadrukken (maar hoeven dus niet).



Merk op dat indien je aan nesting doet, de scope doorheen de *inner* geneste codeblokken doorloopt en pas eindigt bij de accolade van het block waarbinnen de variabele werd gedeclareerd.

5.3.1 Variabelen met zelfde naam

Zolang je in de scope van een variabele bent, kan je geen nieuwe variabele met dezelfde naam aanmaken:

Volgende code is dus niet toegestaan:

```
1 int getal = 0;
2 {
3     int getal = 5; //Deze lijn is niet toegestaan
4 }
```

Je krijgt de foutbericht: “A local variable named ‘getal’ cannot be declared in this scope because it would give a different meaning to ‘getal’, which is already used in a ‘parent or current’ scope to denote something else.”

Enkel de tweede variabele een andere naam geven is toegestaan in het voorgaande geval.

In volgende voorbeeld is dit dus wel geldig, daar de scope van de eerste variabele afgesloten wordt door de accolades:

```
1 {
2     int getal = 0 ;
3     //...
4 }
5 //Verder in code
6 {
7     int getal = 5;
8 }
```

5.4 Switch

Een **switch** statement is een element om een veelvoorkomende constructie van **if/if else ...else** eenvoudiger te schrijven. Vaak komt het voor dat we bijvoorbeeld aan de gebruiker vragen om een keuze te maken (bijvoorbeeld een getal van 1 tot 10, waarbij ieder getal een ander menu-item uitvoert van het programma), zoals:

```

1 Console.WriteLine("Kies: 1)afbreken\n2)opslaan\n3)laden:");
2 int option = int.Parse(Console.ReadLine());
3
4 if (option == 1)
5     Console.WriteLine("Afbreken gekozen");
6 else if (option == 2)
7     Console.WriteLine("Opslaan gekozen");
8 else if (option == 3)
9     Console.WriteLine("Laden gekozen");
10 else
11     Console.WriteLine("Onbekende keuze");

```

Met een **switch** kan dit eenvoudiger wat ik zo meteen demonstreer. Eerst bekijken we hoe **switch** juist werkt. De syntax van een **switch** is speciaal dan de andere programma flow-elementen (**if**, **while**, enz.), namelijk als volgt:

```

1 switch (value)
2 {
3     case constant:
4         statements
5         break;
6     case constant:
7         statements
8         break;
9     default:
10        statements
11        break;
12 }

```

Laten we eens kijken welke nieuwe zaken we hier terugvinden³:

- **value** is de testvariabele die wordt gebruikt als test in de switch (**option** in het voorbeeld).
- Iedere mogelijke waarde van value begint met het **case** keyword. Gevolgd door de waarde (de **case constant**) die value moet hebben om in deze case te *springen*.
- Na het **dubbelpunt** volgt vervolgens de **code** die moet uitgevoerd worden in deze **case**. Deze code mag bestaan uit meerdere lijnen code. Accolades zijn hier trouwens niet verplicht.
- Iedere case moet afgesloten worden met het **break** keyword.

³Sinds C# 7 is de **switch** met enkele krachtige uitbreidingen vergroot. Ik heb bewust gekozen om deze niét in dit boek op te nemen omdat ze anders je eerste contact met **switch** noodeloos moeilijker maakt dan zou moeten. Toch nieuwsgierig wat de nieuwe **switch** kan? Lees dan zeker eens thomasclaudiushuber.com/2021/02/25/c-9-0-pattern-matching-in-switch-expressions voor een mooi overzicht van alle nieuwigheden.

- Tijdens de uitvoer zal de switch de inhoud van de testvariabele (`value`) vergelijken met iedere case constant. Dit gebeurt van boven naar beneden. Wanneer een gelijkheid wordt gevonden dan wordt die case uitgevoerd.
- Indien geen case wordt gevonden die gelijk is aan `value` dan zal de code binnen de `default`-case uitgevoerd worden (de `else` achteraan indien alle vorige `if else`-tests negatief waren).



We zijn dus 4 gereserveerde keywords rijker:

- `switch`
- `case`
- `break`
- `default`

Het menu van zonet kunnen we nu herschrijven naar een `switch`:

```
1 Console.WriteLine("Kies: 1)afbreken\n2)opslaan\n3)laden:");
2 int option = int.Parse(Console.ReadLine());
3
4 switch (option)
5 {
6     case 1:
7         Console.WriteLine("Afbreken gekozen");
8         break;
9     case 2:
10        Console.WriteLine("Opslaan gekozen");
11        break;
12    case 3:
13        Console.WriteLine("Laden gekozen");
14        break;
15    default:
16        Console.WriteLine("Onbekende keuze");
17        break;
18 }
```



De case waarden moeten constanten zijn en mogen dus geen variabelen zijn. Constanten zijn de welbekende *literals* (1, "1", 1.0, 1.d, '1', enz.). Uiteraard moeten de case waarden van hetzelfde datatype zijn als die van de testwaarde.

5.4.1 Fallthrough

Soms wil je dat dezelfde code uitgevoerd wordt bij 2 of meer cases. Je kan ook zogenaamde fallthrough cases beschrijven wat er als volgt uit ziet:

```
1 switch (option)
2 {
3     case 1:
4         Console.WriteLine("Afbreken gekozen");
5         break;
6     case 2:
7     case 3:
8         Console.WriteLine("Laden of opslaan gekozen");
9         break;
10    default:
11        Console.WriteLine("Onbekende keuze");
12        break;
13 }
```

In dit geval zullen zowel de waarden 2 en 3 resulteren in de zin “Laden of opslaan gekozen” op het scherm.

5.5 Enum



Helm op alsjeblieft! **enum** is een erg onderschat concept bij beginnende programmeurs. Enums zijn wat raar in het begin, maar van zodra je er mee weg bent zal je niet meer zonder kunnen en zal je code zoveel eleganter en stoerder worden. Zet je helm dus op en begin er aan!

5.5.1 De bestaansreden voor enums

Stel dat je een programma moet schrijven dat afhankelijk van de dag van de week iets anders moet doen. In een wereld zonder enums (**enumeraties**, letterlijk *opsommingen*) zou je dit kunnen schrijven op 2 zeer foutgevoelige manieren:

1. Met een **int** die een getal van 1 tot en met 7 kan bevatten.
2. Met een **string** die de naam van de dag bevat (bv. "woensdag")

5.5.1.1 Slechte oplossing 1: Met int

De waarde van de dag staat in een variabele **int** `dagKeuze`. We bewaren er 1 in voor maandag, 2 voor dinsdag, enz. Vervolgens kunnen we dan schrijven:

```
1 if(dagKeuze == 1)
2 {
3     Console.WriteLine("We doen de maandag dingen");
4 }
5 else if (dagKeuze == 2)
6 {
7     Console.WriteLine("We doen de dinsdag dingen");
8 }
9 else if
10 //enz.
```

Deze oplossing heeft 2 grote nadelen:

- Wat als we per ongeluk `dagKeuze` een niet geldige waarde geven, zoals 9, 2000 of -4 ?
- De code is niet erg leesbaar. Wat was `dagKeuze == 2` nu weer? Was 2 nu dinsdag of woensdag (want misschien was maandag 0 i.p.v. 1) ?

5.5.1.2 Slechte oplossing 2: Met strings

Laten we tweede manier eens bekijken: de waarde van de dag bewaren we in een variabele **string** dagKeuze. We bewaren de dagen als "maandag", "dinsdag", enz.

```
1 if(dagKeuze == "maandag")
2 {
3     Console.WriteLine("We doen de maandag dingen");
4 }
5 else if (dagKeuze == "dinsdag")
6 {
7     Console.WriteLine("We doen de dinsdag dingen");
8 }
9 else if //enz.
```

De code wordt nu wel leesbaarder, maar toch is ook hier 1 groot nadeel:

- De code is veel foutgevoeliger voor typefouten. Wanneer je "Maandag" i.p.v. "maandag" bewaart dan zal de if al niet werken. Iedere schrijffout of variant zal falen.

5.5.2 Enumeraties: het beste van beide werelden

Enumeraties (**enum**) zijn een C# syntax dat bovenstaand probleem oplost en het beste van beide slechte oplossingen samenvoegt :

1. **Leesbaardere code.**
2. Minder foutgevoelige code, en dus minder potentiële bugs.
3. VS kan je helpen met sneller de nodige code te schrijven.

Het keyword **enum** geeft aan dat we een nieuw datatype maken dat maar enkele mogelijke waarden kan hebben. Nadat we dit nieuwe datatype hebben gedefinieerd kunnen we variabelen van dit nieuwe datatype aanmaken. Deze variabelen mogen enkel waarden bevatten die in het datatype worden gedefinieerd. Ook zal IntelliSense van Visual Studio je de mogelijke waarden helpen invullen.



In C# zitten al veel enum-types ingebouwd. Denk maar aan ConsoleColor: wanneer je de kleur van het lettertype van de console wilt veranderen gebruiken we een enum-type. Er werd reeds gedefinieerd wat de toegelaten waarden zijn, bijvoorbeeld:
Console.ForegroundColor = ConsoleColor.Red;

5.5.3 Zelf enum maken

Zelf een **enum** type maken en gebruiken gebeurt in 2 stappen:

1. Het nieuwe datatype en de mogelijke waarden definiëren.
2. Waarden van het nieuwe enumtype aanmaken en gebruiken in je code.

5.5.3.1 Stap 1: het nieuwe datatype definiëren

We maken eerst een enum type aan. **In je console-applicaties moet dit binnen class Program gebeuren, maar niet binnen de Main methode:**

```
1 enum Weekdagen{Maandag,Dinsdag,Woensdag,Donderdag,Vrijdag,Zaterdag,  
Zondag};
```

Als volgt dus:

```
1 enum Weekdagen{Maandag,Dinsdag,Woensdag,Donderdag,Vrijdag,Zaterdag,  
Zondag};  
2  
3 static void Main(string[] args)  
4 {  
5     Console.WriteLine("Hello enum");  
6 }
```

We hebben nu letterlijk **een nieuw datatype aangemaakt**, genaamd Weekdagen.

5.5.3.2 Stap 2: variabelen van het nieuwe datatype aanmaken en gebruiken

Net zoals **int, double** enz. kan je nu ook variabelen van het type Weekdagen aanmaken. Hoe cool is dat!?

Bijvoorbeeld:

```
1 Weekdagen dagKeuze;  
2 Weekdagen andereKeuze;
```

En vervolgens kunnen we waarden aan deze variabelen toewijzen als volgt:

```
1 dagKeuze = Weekdagen.Donderdag;
```

Kortom: we hebben variabelen zoals we gewoon zijn, het enige verschil is dat we nu beperkt zijn in de waarden die we kunnen toewijzen. Deze kunnen enkel de waarden krijgen die in het type gedefinieerd werden. De code is nu ook een pak leesbaarder geworden.

5.5.4 Enums en beslissingen werken graag samen

Ook de beslissingsstructuren worden leesbaarder:

```
1 if(dagKeuze == Weekdagen.Woensdag)
```

of een switch:

```
1 switch(dagKeuze)
2 {
3     case Weekdagen.Maandag:
4         Console.WriteLine("It's monday!");
5         break;
6     case Weekdagen.Dinsdag:
7         //enz.
8 }
```



Visual Studio houdt van enums (*ik ook trouwens*) en zal je helpen bij het schrijven van een **switch** indien de test-variabele een enum-type bevat. Hoe?

- Schrijf **switch** en druk op 2 maal op tab. Normaal verschijnt er nu een “prefab” switch structuur met een test-waarde genaamd **switch_on** die een gele achtergrond heeft.
- Overschrijf **switch_on** met de variabele die je wilt testen (bv. **dagKeuze**).
- Klik nu met de muis eender waar binnen de accolades van de **switch**.
- Profit!

5.5.5 Conversie van en naar enum variabelen

De waarde van een enum-variabelen wordt intern als een **int bewaard.** In het geval van de Weekdagen zal maandag standaard de waarde 0 krijgen, dinsdag 1, enz.

Volgende conversies met behulp van **casting** zijn dan ook perfect toegelaten:

```
1 int keuze = 3;
2 Weekdagen dagKeuze = (Weekdagen)keuze;
3 //dagKeuze zal de waarde Weekdagen.Donderdag hebben
```

Wil je dus bijvoorbeeld 1 dag bijtellen dan kan je schrijven:

```
1 Weekdagen dagKeuze= Weekdagen.Dinsdag;
2 int extradag= (int)dagKeuze + 1;
3 Weekdagen nieuweDag= (Weekdagen)extradag;
4 //extraDag heeft de waarde Weekdagen.Woensdag
```

Let er wel op dat je geen extra dag op Zondag probeert bij te tellen. Dat zal niet werken.

5.5.6 Andere interne waarde toekennen

Standaard worden de enum waarden intern dus genummerd beginnende bij 0. Je kan dit ook manueel veranderen door bij het maken van de **enum** expliciet aan te geven wat de interne waarde moet zijn, als volgt:

```
1 enum WeekDagen  
2 {Maandag=1, Dinsdag, Woensdag, Donderdag, Vrijdag, Zaterdag,  
Zondag}
```

De dagen zullen nu vanaf 1 genummerd worden, dus `WeekDagen.Woensdag` zal de waarde 3 hebben.

We kunnen ook nog meer informatie meegeven, bijvoorbeeld:

```
1 enum WeekDagen  
2 {Maandag=1, Dinsdag, Woensdag, Donderdag, Vrijdag, Zaterdag=50,  
Zondag=60}
```

In dit geval zullen Maandag tot Vrijdag intern als 1 tot en met 5 bewaard worden, Zaterdag als 50, en Zondag als 60.



De individuele enum waarden moeten steeds met een hoofdletter starten.

5.5.7 Gebruikersinvoer naar enum

Heel vaak zal je een programma schrijven waarbij de gebruiker een keuze moet maken uit een menu of iets dergelijks. Dit menu kan je voorstellen met een enum. Het probleem is vervolgens vragen wat de keuze van de gebruiker is en deze dan verwerken. Je zou dit kunnen doen met behulp van een reeks if-testen (**if**(*userinput*==**"demo"**)...) Maar waarom niet de kracht van **enum** benutten?!

Volgende code toont hoe je dit kunt doen:

```
1 enum Menu {Demo=1, Start, Einde}
2 static void Main(string[] args)
3 {
4     Console.WriteLine("Wat wil je doen?");
5     Console.WriteLine("1. Demo");
6     Console.WriteLine("2. Start");
7     Console.WriteLine("3. Einde");
8     int userkeuze = int.Parse(Console.ReadLine());
9
10    Menu keuze = (Menu)userkeuze;
11
12    switch (keuze)
13    {
14        //...
```

5.5.8 Parsen van enum

Sinds .NET 5 uitkwam, is er een meer gebruiksvriendelijke manier verschenen om een string te parsen naar een enum variabele. Hierbij wordt gebruikt gemaakt van *generics* (herkenbaar aan < >), een concept dat uit de doeken wordt gedaan in de appendix van dit boek.

Echter, zelfs zonder generics te kennen zou volgende code toch begrijpbaar moeten zijn. We gebruiken terug het eerder gedefinieerde *Menu* type en de nieuw beschikbare *Enum.Parse< >*-methode :

```
1 Menu keuze = Enum.Parse<Menu>(Console.ReadLine());
```

We plaatsen tussen de < > het enum datatype naar waar we willen parsen.

Optioneel kan je via een tweede argument van het type bool aangeven of de parsing hoofdlettergevoelig is (**false**) of niet (**true**):

```
1 Menu keuze = Enum.Parse<Menu>(Console.ReadLine(), true);
```



Ah, de tijden zonder **enum**. Ik weet nog hoe we onze grotten beschilderen zonder ons druk te moeten maken in enumeraties. Om maar te zeggen: je kan perfect leven zonder **enum**. Vele programmeurs voor je hebben dit bewezen. Echter, van zodra ze **enum** ontdekken (en begrepen) zijn nog maar weinig programmeurs er terug van afgestapt.

De eerste kennismaking met enumeraties is wat bevreemdend: je kan plots je eigen datatypes aanmaken?! Van zodra je ze in de vingers hebt zal je ontdekken dat je veel leesbaardere code kunt schrijven én dat Visual Studio je kan helpen met het opsporen van bugs.

Wanneer gebruik je **enum**? Telkens je één of meer variabelen nodig hebt, waarvan je perfect op voorhand weet welke mogelijke waarden ze mogen hebben. Ze worden bijvoorbeeld vaak gebruikt in *finite state machines*.

Bij game development willen we bijhouden in welke staat het programma zich bevindt: Intro, Startmenu, Ingame, Gameover, Optionsscreen, enz. Dit is een typisch **enum** verhaal. We definiëren hiervoor het volgende type:

```
1 enum gamestate {Intro, Startmenu, Ingame, Gameover, Optionsscreen}
```

En vervolgens kunnen we dan met een eenvoudige switch in ons hoofdprogramma snel de relevante code uitvoeren:

```
1 //Bij opstart:
2 gamestate playerGameState= gamestate.Intro;
3 // ...
4 //later
5 switch(playerGameState)
6 {
7     case gamestate.Intro:
8         //show fancy movie
9         break;
10    case gamestate.Startmenu:
11        //show start menu
12        break;
13    //enz.
```

Een ander typisch voorbeeld is schaken. We maken een enum om de speelstukken voor te stellen (Pion, Koning, Toren enz.) en kunnen hen dan laten bewegen en vechten in uiterst leesbare code:

```
1 if(spelstuk == Schaakstuk.Paard)
```

6 Loops

In het vorige hoofdstuk hebben we geleerd hoe we onze code konden vertakken (**branching**) door beslissingen te nemen. Dit stelde ons in staat om verschillende stukken code uit te voeren, afhankelijk van de waarde van bepaalde variabelen of de invoer van de gebruiker.

Wat we nog niet konden was **terug naar een vorige plek in het algoritme gaan**. Soms willen we dat een heel stuk code 2 of meerdere keren moet uitgevoerd worden tot aan een bepaalde conditie wordt voldaan: “*Voer volgende code uit tot dat de gebruiker 666 invoert.*”

Herhalingen (**loops**) creëer je wanneer bepaalde code een aantal keer moet herhaald worden. Hoe vaak de herhaling moet duren is afhankelijk van de conditie die je hebt bepaald. Elke keer dat de code binnen de loop wordt uitgevoerd, noemen we dit een **iteratie**. Dit betekent dat de loop bij elke iteratie zijn codeblok opnieuw doorloopt, totdat de gestelde conditie niet meer voldoet.

Door herhalende code met loops te schrijven maken we onze code korter en bijgevolg ook minder foutgevoelig en beter onderhoudbaar.

6.1 Soorten loops

Er zijn verschillende categorieën loops:

- **Definite of counted loop:** een loop waarbij het aantal herhalingen vooraf bekend is. Bijvoorbeeld: alle getallen van 0 tot en met 100 tonen.
- **Indefinite of sentinel loop:** een loop waarvan op voorhand niet kan gezegd worden hoe vaak deze zal uitgevoerd worden. Invoer van de gebruiker of een interne test bepaalt wanneer de loop stopt. Bijvoorbeeld: “*Voer getallen in, voer -1 in om te stoppen*” of “*Bereken de grootste gemene deler*”.
- **Oneindige loop:** een loop die nooit stopt. Soms gewenst (bv. de game loop) of, vaker, een bug.

6.1.1 Loops in C#

Er zijn 3 standaard manieren om loops te maken in C#:

- **while**: zal 0 of meerdere keren uitgevoerd worden.
- **do while**: zal minimaal 1 keer uitgevoerd worden.
- **for**: een alternatieve, iets compactere manier om loops te beschrijven wanneer je exact weet hoe vaak de loop zal moeten herhalen.

Daarnaast zullen we in hoofdstuk 9 een speciale loopvariant leren kennen wanneer we arrays en objecten bespreken:

- **foreach**: een leesbaardere manier van loopen, die vooral nuttig is wanneer je met objecten werkt.

De 3 categorieën loops die we net bespraken kunnen in principe met eender welk looptype in C# geschreven worden. Toch raad ik je aan om vanaf nu steeds wel dooracht na te denken welk looptype het best bij je probleem past. Samengevat kan je het volgende zeggen:

Looptype	Definite loop	Indefinite loop	Oneindige loop
While en do while	x	x	x
For	x		
Foreach	x		

Deze tabel suggereert dat we met **while** en **do while** meer kunnen, wat ook zo is. Je zal echter gauw ontdekken dat je vaak terugvalt op code met een **for** omdat:

1. Deze compatere code oplevert.
2. Veel problemen met een definite loop kunnen opgelost worden.

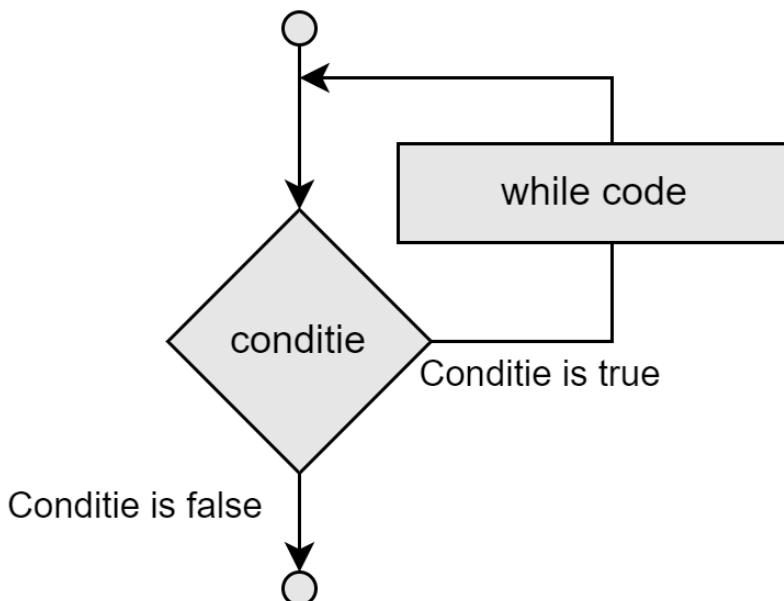
6.2 While

De syntax van een while loop is eenvoudig:

```
1 while (conditie)
2 {
3     // code zal uitgevoerd worden zolang de conditie waar is
4 }
```

Net als bij een **if**-statement wordt de conditie hier uitgedrukt als een booleaanse expressie met één of meerdere relationele operatoren. **Zolang de conditie true is zal de code binnen de accolades uitgevoerd worden.** Indien de conditie reeds vanaf het begin **false** is dan zal de code binnen de loop nooit worden uitgevoerd.

Telkens wanneer het programma aan het einde van het **while** codeblock komt springt het terug naar de conditie bovenaan en zal de test wederom uitgevoerd worden. Is deze weer **true** dan wordt de code weer uitgevoerd. Van zodra de test **false** is zal de code voorbij het codeblock springen en na het **while** codeblok doorgaan. De flowchart is duidelijk:



Figuur 6.1: While flowchart.

Een voorbeeld van een eenvoudige while loop:

```
1 int tellertje = 0;
2 while (tellertje < 100)
3 {
4     tellertje++;
5     Console.WriteLine(tellertje);
6 }
```

Zolang tellertje kleiner is dan 100 (tellertje < 100) zal het met 1 verhoogd worden en op het scherm worden getoond. We krijgen met dit programma dus alle getallen van 1 tot en met 100 op het scherm onder elkaar te zien. Daar de test gebeurt aan het begin van de loop wil dit zeggen dat het getal 100 nog wel getoond zal worden. **Begrijp je waarom?** Test dit zelf!



Zodra je dezelfde lijn(en) code meerdere keren onder elkaar ziet staan, is de kans groot dat je dit korter kunt schrijven met behulp van loops (of methoden, wat we in het volgende hoofdstuk zullen bespreken).

Code onder elkaar kopiëren en plakken is dus vaak een duidelijke indicator dat je loops kan gebruiken.

6.2.1 Complexe condities

Uiteraard mag de conditie waaraan een loop moet voldoen complexer zijn door middel van de relationele operators.

Volgende **while** bijvoorbeeld zal uitgevoerd worden zolang teller groter is dan 5 én de variabele naam van het type **string** niet gelijk is aan "tim":

```
1 while(teller > 5 && naam != "tim")
2 {
3     //Keep repeating
4 }
```

6.2.2 Oneindige loops

Indien de loop-conditie nooit **false** wordt dan heb je een oneindige loop gemaakt. Soms is dit gewenst gedrag. Maar vaker is dit een bug en zal je dit moeten debuggen.

Volgende twee voorbeelden tonen dit:

Een **bewust oneindige loop**:

```
1 while(true)
2 {
3     //"To infinity and beyond!"
4 }
```

Een **bug die een oneindige loop veroorzaakt**:

```
1 int teller = 0;
2 while(teller<10)
3 {
4     Console.WriteLine(teller);
5     teller--; //oops, dit had teller++ moeten zijn
6 }
```



Zorg er altijd voor dat de variabele(n) die je in je testconditie gebruikt, ook effectief in de loop worden aangepast. Als deze in de loop niet verandert dan zal ook de test-conditie dezelfde blijven en heb je dus een oneindige loop gemaakt.

6.2.3 Scope van variabelen in loops

Let er op dat de scope van variabelen bij loops zeer belangrijk is. Indien je een variabele binnen de loop definieert dan zal deze steeds terug “geset” worden wanneer de volgende iteratie van de loop start. Volgende code toont bijvoorbeeld **foutief** hoe je de som van de eerste 10 getallen ($1+2+3+\dots+10$) zou maken:

```
1 int teller = 1;
2 while(teller <= 10)
3 {
4     int som = 0;
5     som = som+teller;
6     teller++;
7 }
8 Console.WriteLine(som); //deze lijn zal een fout genereren
```

Voorgaande code zal volgende VS foutbericht geven: *The name ‘som’ does not exist in the current context.*

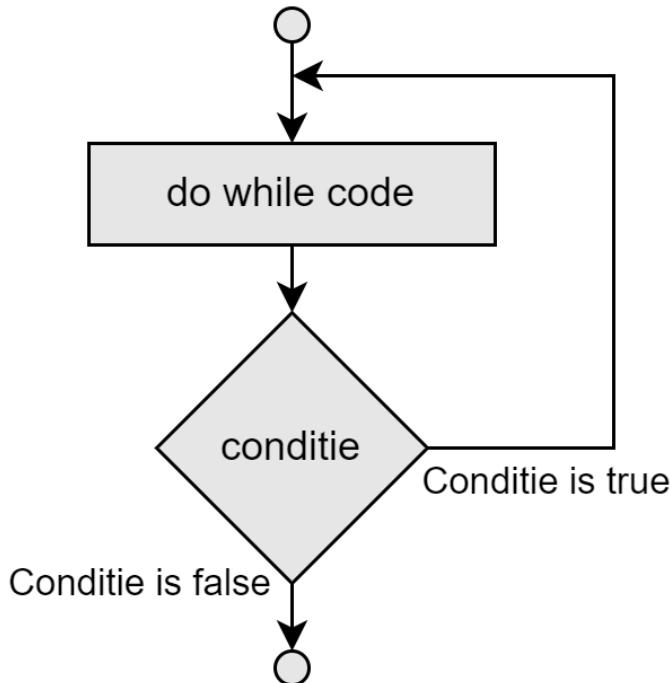
De **correcte** manier om dit op te lossen is te beseffen dat de variabele som enkel binnen de accolades van de while-loop gekend is. Op de koop toe wordt deze steeds terug op 0 gezet en er kan dus geen som van alle teller-waarden bijgehouden worden. Hier de oplossing:

```
1 int teller = 1;
2 int som = 0;
3 while(teller <= 10)
4 {
5     som = som+teller;
6     teller++;
7 }
8 Console.WriteLine(som);
```

6.3 Do while

In tegenstelling tot een **while** loop, zal een **do while** loop sowieso **minstens 1 keer uitgevoerd worden**. De reden is eenvoudig: de stopconditie gecontroleerd wordt na iedere iteratie getest. Bij een **while** gebeurt dit voor dat er een nieuwe iteratie wordt gestart.

Vergelijk volgende flowchart van de **do while** met die van de **while**:



Figuur 6.2: De **do while** flowchart.

De syntax van een **do while** is eveneens eenvoudig:

```

1 do{
2     //code zal uitgevoerd worden zolang de conditie waar is
3 } while (conditie);
  
```



Merk op dat achteraan de testconditie een puntkomma na het ronde haakje staat.
Deze vergeten is een v  l voorkomende fout. Bij een while is dit niet!

Het volgende eenvoudige aftelprogramma toont de werking van de **do while** loop:

```
1 int i = 10;
2 do
3 {
4     i--;
5     Console.WriteLine(i);
6 } while (i > 0);
```

Begrijp je wat dit programma zal doen? Inderdaad, dit zal alle getallen van 9 tot en met 0 onder elkaar op het scherm zetten.

6.3.1 Foute input van gebruiker met loops verwerken

Dankzij loops kunnen we nu ook eenvoudiger omgaan met foutieve input van de gebruiker. Stel dat we volgende vraag hebben gesteld aan de gebruiker:

```
1 Console.WriteLine("Geef uw keuze in: a, b of c");
2 string input = Console.ReadLine();
```

Met een loop kunnen we nu deze vragen blijven stellen tot de gebruiker een geldige input (a,b of c) geeft:

```
1 string input;
2 do
3 {
4     Console.WriteLine("Geef uw keuze in: a, b of c");
5     input = Console.ReadLine();
6 }while(input != "a" && input != "b" && input != "c");
```

Zolang (while) de gebruiker niet "a", "b" of "c" invoert zal de loop zichzelf blijven herhalen.

Merk op dat we de variabele **string input** voor de **do while** moeten aanmaken. Zouden we die in de loop pas aanmaken dan zou de variabele niet als test kunnen gebruikt worden aan het einde van de loop. De reden? Wederom de scope van variabelen. De accolades van de **do while** creëren een duidelijke scope die iedere iteratie verdwijnt en terug wordt aangemaakt, inclusief dus variabelen die binnen deze accolades worden aangemaakt.



Ik herhaal voorgaande nog eens nadrukkelijk omdat hier vaak fouten op gemaakt worden: je ziet dat de test achteraan (**while (input...);**) buiten de accolades van de loop ligt en dus een andere scope heeft.



De booleaanse expressie `input != "a" && input != "b" && input != "c"` kan ook anders geschreven met dezelfde interne logica (en dus werking) als:

```
1 !(input == "a" || input == "b" || input == "c")
```

Sommige mensen prefereren deze tweede vorm. Maar dat is persoonlijke smaak.



Voorgaande logica is een gevolg van de **Wetten van De Morgan** (ook wel *dualiteit van De Morgan* genoemd) die het verband leggen tussen de logische operatoren EN, OF en de negatie.

Deze wetten zeggen dat (uitgedrukt even in C# voor de duidelijkheid):

- $!(A \& B)$ is hetzelfde als $\neg A \vee \neg B$.
- $!(A \vee B)$ is hetzelfde als $\neg A \wedge \neg B$.

Zie je hoe ik de tweede wet gebruikt heb in het voorgaande voorbeeld om de alternatieve logica te vinden?

6.4 For-loops

Een veelvoorkomende manier van while-loops gebruiken is waarbij je een bepaalde teller bijhoudt die je telkens met een bepaalde waarde verhoogt. Wanneer de teller een bepaalde waarde bereikt moet de loop afgesloten worden.

Bijvoorbeeld volgende code om alle even getallen van 0 tot 10 te tonen:

```
1 int i = 0;
2 while(i<11)
3 {
4     Console.WriteLine(i);
5     i = i + 2;
6 }
```

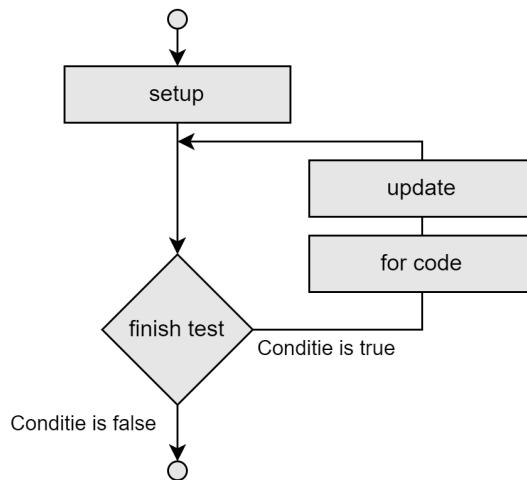
Met een for-loop kunnen we deze veel voorkomende code-constructie verkort schrijven.

6.4.1 For syntax

De syntax van een **for**-loop is de volgende:

```
1 for (setup; finish test; update)
2 {
3     //code die zal uitgevoerd worden zolang de finish test true geeft
4 }
```

- **setup:** Hier zetten we de “wachter-variabele” op de beginwaarde. De wachter-variabele is de variabele die we tijdens de loop in het oog zullen houden en die zal bepalen hoe vaak de loop moet uitgevoerd worden (bv. `int i = 0;`).
- **finish test:** Hier plaatsen we een booleaanse expressie die de wachter-variabele gebruikt om te testen of een volgende iteratie moet worden uitgevoerd (bv. `i<11`).
- **update:** Hier plaatsen we wat er moet gebeuren na iedere iteratie. Meestal zullen we hier de wachter-variabele verhogen of verlagen (bv. `i = i + 2`).



Figuur 6.3: For flowchart.

Voor de *setup*-variabele kiest men meestal *i*, maar dat is niet noodzakelijk.

Gebruiken we deze kennis, dan kunnen we de eerder vermelde code om de even getallen van 0 tot en met 10 tonen als volgt:

```

1 for (int i = 0; i < 11; i += 2)
2 {
3     Console.WriteLine(i);
4 }
```

Deze code zal telkens *i* met 2 verhogen(*update*), startende bij 0 (*setup*). Het blijft dit doorlopen zolang *i* kleiner is dan 11 (*finish test*). Als output krijgen we:

```

1 0
2 2
3 4
4 6
5 8
6 10
```

**for-tab-tab**

Als je in Visual Studio **for** typt en dan tweemaal op [tab] duwt krijg je een kant en klare for-loop:

```
for (int i = 0; i < length; i++)  
{  
}  
}
```

Telkens je vervolgens op [tab] duwt verspringt je cursor tussen **i** en **length**. Op die manier kan je dus snel een for schrijven.

6.4.2 continue en break

Het **continue** keyword laat toe om in een loop de huidige iteratie te eindigen en weer naar de start van de volgende iteratie te gaan. In het volgende voorbeeld gebruiken we **continue** om alle getallen van 1 tot 10 te tonen waarbij we het getal 5 zullen overslaan:

```
1 for (int i = 1; i <= 10; i++)
2 {
3     if (i == 5)
4     {
5         continue;
6     }
7     Console.WriteLine(i);
8 }
```

Met **break** kan je loops altijd vroegtijdig stopzetten. Je springt dan als het ware ogenblikkelijk uit de loop. Je ziet het aankomen zeker? Yups, daar is ie....



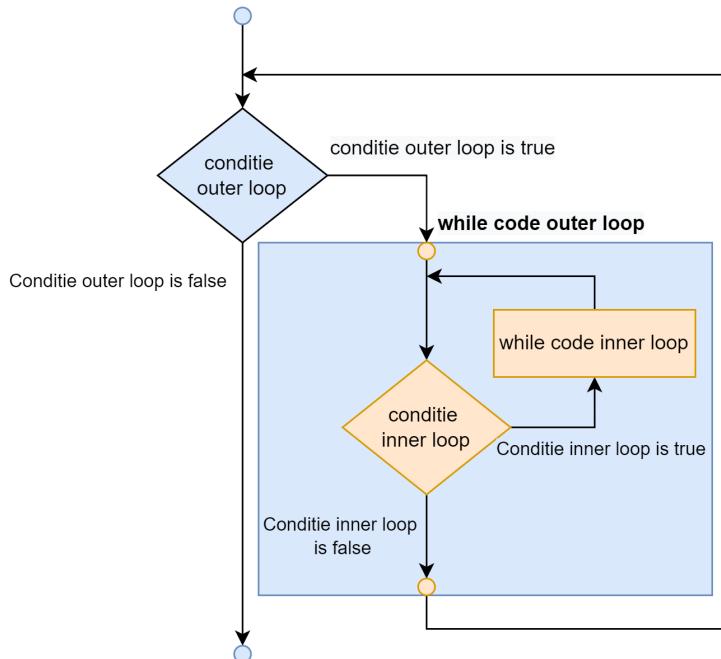
Olla!? Wat denken we dat we aan het doen zijn? Gelieve die keywords ogenblikkelijk terug uit je code te verwijderen. Bedankt.

break en **continue** zijn de subtielere vrienden van **goto**. Ze werken net als **goto** in de schemerzone tussen wat wenselijk is en wat niet. Dit maakt ze extra gevaarlijk. Voordat je **break** als oplossing gebruikt, probeer eerst of je de loop netjes kunt afsluiten door bijvoorbeeld de juiste booleaanse expressie in de testconditie te gebruiken. Hetzelfde geldt voor **continue**, dat ook snel goto-achtige bugs kan veroorzaken.

*Ik heb gemerkt dat beginnende C#-programmeurs vaak te lui zijn om een deftige stopconditie voor hun loop te schrijven. En dan maar **break** als oplossing hanteren.*

6.5 Nested loops

Wanneer we 1 of meerdere loops in een andere loop plaatsen dan spreken we over **geneste loops**. Geneste loops komen vaak voor, maar zijn wel een ander paar mouwen wanneer je deze zaken wilt debuggen en correct schrijven.



Figuur 6.4: Voorbeeld van geneste loops.

We spreken steeds over de **outer loop** als de omhullende of “grootste” loop. Waarbij de binnennste loop de **inner loop** is.

Volgende code toont bijvoorbeeld 2 loops die genest werden:

```

1 int tellerA = 0;
2 int tellerB = 0;
3
4 while(tellerA < 3) //outer loop
{
    tellerA++;
    tellerB = 0;
    while(tellerB < 5)
    {
        tellerB++;
        Console.WriteLine($"A:{tellerA}, B: {tellerB}");
    }
}

```

De uitvoer hiervan zal als volgt zijn:

```
1 A:1, B: 1
2 A:1, B: 2
3 A:1, B: 3
4 A:1, B: 4
5 A:1, B: 5
6 A:2, B: 1
7 A:2, B: 2
8 A:2, B: 3
9 A:2, B: 4
10 A:2, B: 5
11 A:3, B: 1
12 A:3, B: 2
13 A:3, B: 3
14 A:3, B: 4
15 A:3, B: 5
```

Merk het ‘ritme’ op in de uitvoer. De linkse teller gaat een pak trager dan de rechtse.

6.5.1 Geneste loops tellen

Om te tellen hoe vaak de *inner* code zal uitgevoerd worden dien je te weten hoe vaak iedere loop afzonderlijk wordt uitgevoerd. Vervolgens vermenigvuldig je al deze getallen met elkaar.

Een voorbeeld: Hoe vaak zal het woord “Hallo” op het scherm verschijnen bij volgende code?

```
1 for (int i = 0; i < 10; i++)
2 {
3     for (int j = 0; j < 5; j++)
4     {
5         Console.WriteLine("Hallo");
6     }
7 }
```

De outer loop wordt 10 keer uitgevoerd (waarbij *i* de waarden 0 tot en met 9 aanneemt). De inner loop wordt bij elke iteratie van de outer loop 5 keer uitgevoerd (waarbij *j* de waarden 0 tot en met 4 aanneemt). In totaal zal dus 50 keer “Hallo” op het scherm verschijnen (5×10).



Let er op dat **break** je enkel uit de huidige loop zal halen. Indien je dit dus gebruikt in de inner loop dan zal de outer loop nog steeds voortgaan. Nog een reden om zéér voorzichtig om te gaan met **break**. **Of beter nog: gewoon niet gebruiken!**

7 Methoden

“I will always choose a lazy person to do a difficult job. Because, he will find an easy way to do it.” *Bill Gates, oprichter van Microsoft.*

Het is je misschien nog niet opgevallen, maar sinds het vorige hoofdstuk zijn we de jacht begonnen op zo weinig mogelijk code te schrijven met zoveel mogelijk rendement. Loops waren een eerste stap in de goede richting. De volgende zijn methoden! Tijd om nog luier te worden.

Veel code die we hebben geschreven wordt meerdere keren, al dan niet op verschillende plaatsen, gebruikt. Dit verhoogt natuurlijk de foutgevoeligheid. Door het gebruik van methoden kunnen we de foutgevoeligheid van de code verlagen omdat de code maar op 1 plek staat én maar 1 keer dient geschreven te worden. Echter, ook de leesbaarheid en dus onderhoudbaarheid van de code wordt verhoogd.

Beeld je eens dat we geen gebruik konden maken van de vele .NET bibliotheken. Stel je voor dat `Console.WriteLine` niet bestond? Telkens als we dan iets in C# naar het scherm wilden sturen moesten we de volledige interne code van `WriteLine` uitschrijven. Voor de geïnteresseerden, dat zou er (ongeveer) als volgt uitzien:

```
1  fixed (byte* p = bytes)
2  {
3      if (useFileAPIs)
4      {
5          int numBytesWritten;
6          Interop.Kernel32.WriteFile(hFile, p, bytes.Length, out
7              numBytesWritten, IntPtr.Zero));
8      }
9      else
10     {
11         //enz.
```

Dat is aardig wat bizarre code he? En ik toon maar een stuk. Kortom: we mogen blij zijn dat methoden bestaan. Tijd om ze eens van dichtbij te bekijken!

Trouwens. Het is heel normaal dat voorgaande code je zenuwachtig maakt. Negeer ze maar!¹

¹Toch nieuwsgierig hoe wat er allemaal achter de schermen gebeurt? Voorgaande code komt uit [git-hub.com/dotnet/runtime/blob/main/src/libraries/System.Console/src/System/ConsolePal.Windows.cs](https://github.com/dotnet/runtime/blob/main/src/libraries/System.Console/src/System/ConsolePal.Windows.cs), waar je ook alle andere broncode van de *dotnet runtime* zal terugvinden.

7.1 Werking van methoden

Een methode (ook wel *functie* genoemd) is in C# een blok code dat specifieke taken uitvoert. Een methode bestaat uit één of meerdere statements, kan herhaaldelijk worden aangeroepen met of zonder extra parameters, en kan een resultaat teruggeven. Methoden kunnen vanuit elk deel van je code worden aangeroepen.

Je gebruikt al sinds les 1 methoden. Telkens je `Console.WriteLine()` gebruikt, roep je een methode aan (gennaamd `WriteLine`).

Methoden in C# zijn herkenbaar aan de ronde haakjes achteraan, al dan niet met actuele parameters tussen. Alles wat je nu gaat zien heb je al gebruikt. Het grote verschil zal zijn dat we nu ook **zelf methoden** gaan definiëren, en niet enkel bestaande methoden gebruiken.

Methoden hebben als voordeel dat je herbruikbare stukken code kunt gebruiken en dus niet steeds deze code overal moet kopiëren en plakken. Daarnaast zullen methoden je code ook overzichtelijker maken.

7.1.1 Methode syntax

De basis-syntax van een methode ziet er als volgt uit (de werking van het keyword **static** leg ik uit in hoofdstuk 11):

```
1 static returntype MethodeNaam(optioneel_parameters)
2 {
3     //code van methode
4 }
```

De eerste lijn noemen we de **methode-signatuur**. Deze lijn vertelt alles dat je moet weten om met de methode te werken (returntype, naam en eventuele parameters).

Vervolgens kan je deze methode elders oproepen als volgt, indien de methode geen parameters vereist:

```
1 MethodeNaam();
```

Dat is een mondvol. We gaan daarom de methoden even stapsgewijs leren kennen. Let's go!

7.1.2 Een eenvoudige methode

Beeld je in dat je een applicatie moet maken waarin je op verschillende plaatsen de naam van je programma moet tonen. Zonder methoden zou je telkens moeten schrijven `Console.WriteLine ("Timsoft XP");`

Als je later de naam van het programma wilt veranderen naar iets anders (bv. Timsoft 11) dan zal je manueel overal de titel moeten veranderen in je code. Met een methode hebben we dat probleem niet meer. We schrijven daarom een methode `ToonTitel` als volgt:

```
1 static void ToonTitel()
2 {
3     Console.WriteLine("Timsoft XP");
4 }
```

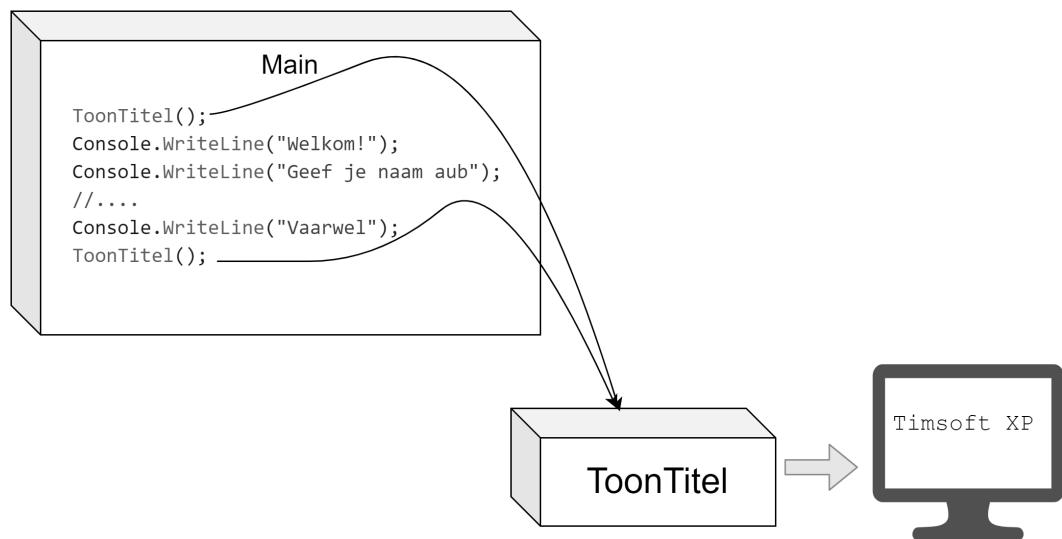
Vanaf nu kan je eender waar in je programma deze methode aanroepen door te schrijven:

```
1 ToonTitel();
```

Volgend programma'tje toont dit:

```
1 namespace Demo1
2 {
3     internal class Program
4     {
5         static void ToonTitel()
6         {
7             Console.WriteLine("Timsoft XP");
8         }
9
10        static void Main(string[] args)
11        {
12            ToonTitel();
13            Console.WriteLine("Welkom!");
14            Console.WriteLine("Geef je naam aub");
15            //...
16            Console.WriteLine("Vaarwel");
17            ToonTitel();
18        }
19    }
20 }
```

Volgende afbeelding toont hoe je programma doorheen de code loopt. De pijlen geven de flow aan:



Figuur 7.1: Visualisatie van bovenstaande code.

7.1.3 Main is ook een methode

Zoals je misschien al begint te vermoeden is dus de `Main` waar we steeds onze code schrijven ook een methode. Een console-applicatie heeft een startpunt nodig en daarom begint ieder programma in deze methode, maar in principe kan je even goed je programma op een andere plek laten starten.

Wat denk je trouwens dat je dit doet?

```
1 static void Main(string[] args)
2 {
3     Console.WriteLine("Ik zit vast!");
4     Main(); //Endless loop incoming!
5 }
```



`string[] args` is een verhaal apart en zullen we in het volgende hoofdstuk bekijken. Ik verklap alvast dat je via deze `args` opstartparameters aan je programma kan meegeven tijdens het opstarten (bijvoorbeeld `explorer.exe google.com`) zodat je code hier iets mee kan doen.

7.2 Returntypes van methoden

Voorgaande methode gaf niets terug. Dat kon je zien aan het keyword **void** (letterlijk: *leegte*).

Vaak willen we echter wel dat de methode iets teruggeeft. Bijvoorbeeld het resultaat van een berekening.

Het returntype van een methode geeft aan wat het type is van de data die de methode als resultaat teruggeeft bij het beëindigen ervan. Eender welk datatype kan hiervoor gebruikt worden (**int**, **string**, **char**, **float**, enz.). Ook **enum** datatypes kunnen als returntype in methoden gebruikt worden (en later ook objecten, wat we in hoofdstuk 10 zullen ontdekken).

7.2.1 return keyword

Het is belangrijk dat in je methode het resultaat ook effectief wordt teruggegeven, dit doe je met het keyword **return** gevolgd door de variabele die moet teruggeven worden.

Denk er dus aan dat deze variabele van het type is dat je hebt opgegeven als zijnde het returntype. Van zodra je **return** gebruikt zal je op die plek uit de methode ‘vliegen’.

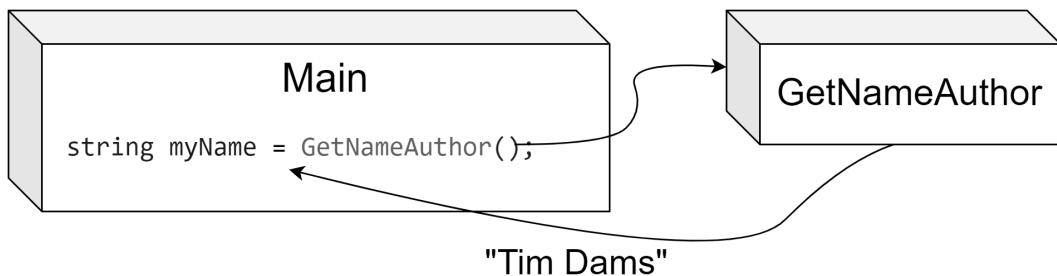
Wanneer je een methode maakt die iets teruggeeft (dus een ander returntype dan **void**) is het ook de bedoeling dat je het resultaat van die methode ontvangt en gebruikt. Je kan bijvoorbeeld het resultaat van de methode in een variabele bewaren. Dit vereist dat die variabele dan van hetzelfde returntype is!

Volgend voorbeeld bestaat uit een methode die de naam van de auteur van je programma teruggeeft:

```
1 static string VerkrijgAuteurNaam()
2 {
3     string name = ;
4     return "Tim Dams";
5 }
```

Een mogelijke manier om deze methode in je programma te gebruiken zou nu kunnen zijn:

```
1 string myName = VerkrijgAuteurNaam();
```



Figuur 7.2: Visualisatie van de flow.

Maar ook dit zal werken:

```
1 Console.WriteLine(VerkrijgAuteurNaam());
```

Of verderop misschien als volgt:

```
1 Console.WriteLine($"Auteur van dit boek: {VerkrijgAuteurNaam()}");
```



Je mag zowel literals als variabelen en zelfs andere methode-aanroepen plaatsen achter het **return** keyword. Zolang het maar om een expressie gaat die een resultaat heeft kan dit. Voorgaande methode kunnen we dus ook schrijven als:

```
1 static string VerkrijgAuteurNaam()
2 {
3     return "Tim Dams";
4 }
```

7.3 Een uitgewerkte methode

De faculteit van een getal n schrijven we als $n!$. Het is het product van alle positieve getallen van 1 tot en met n , waarbij $0!$ gelijk is aan 1. Hier een voorbeeld van een methode die de faculteit van 5 berekent, $5!$. We willen dus $1 \times 2 \times 3 \times 4 \times 5$ berekenen, wat 120 is. De oproep van de methode gebeurt vanuit de Main-methode:

```

1 internal class Program
2 {
3     static int FaculteitVan5()
4     {
5         int resultaat = 1;
6         for (int i = 1; i <= 5; i++)
7         {
8             resultaat *= i;
9         }
10        return resultaat;
11    }
12
13    static void Main(string[] args)
14    {
15        Console.WriteLine($"Faculteit van 5 is {FaculteitVan5()}");
16    }
17 }
```

7.3.1 void

Indien je methode niets teruggeeft wanneer de methode eindigt (bijvoorbeeld indien de methode enkel tekst op het scherm toont) dan dien je dit ook aan te geven. Hiervoor gebruik je het keyword **void**.

Een voorbeeld:

```

1 static void ToonVersie()
2 {
3     Console.WriteLine("Dit is versie 8.31 ");
4 }
```

Deze methode moet je dus als volgt aanroepen:

```
1 ToonVersie();
```

Volgende 2 manieren **werken niet** bij een methode met **void** als returntype:

```

1 string result = ToonVersie(); //MAG NIET!!
2 Console.WriteLine(ToonVersie()); // MAG NIET!
```

7.3.2 return

Je mag het **return** keyword eender waar in je methode gebruiken. Weet wel dat van zodra een statement met **return** wordt bereikt de methode ogenblikkelijk afsluit en het resultaat achter **return** teruggeeft.

Soms is dit handig zoals in volgende voorbeeld:

```
1 static string WindRichting()
2 {
3     Random r = new Random();
4     switch (r.Next(0,4))
5     {
6         case 0:
7             return "noord";
8             break;
9         case 1:
10            return "oost";
11            break;
12        case 2:
13            return "zuid";
14            break;
15        case 3:
16            return "west";
17            break;
18    }
19    return "onbekend";
20 }
```

Merk op dat de onderste lijn (19) nooit zal bereikt worden. Toch vereist C# dit!



Dacht je nu echt dat ik weg was?! Het is me opgevallen dat je niet altijd de foutboodschappen in VS leest. Ik blijf alvast uit jouw buurt als je zo doorgaat. Doe jezelf (en mij) dus een plezier en probeer die foutboodschappen in de toekomst te begrijpen. Er zijn er maar een handvol en bijna altijd komen ze op hetzelfde neer. Neem nou de volgende:**Not all code paths return a value** Die ga je nog vaak tegenkomen!

Bovenstaande foutboodschap zal je vaak krijgen en geeft altijd aan dat er bepaalde delen binnen je methode zijn waar je kan komen zonder dat er een **return** optreedt. Het einde van de methode wordt met andere woorden bereikt zonder dat er iets uit de methoden terug komt (wat enkel bij **void** mag).

Foutboodschappen hebben de neiging om gecompliceerder te klinken dan de effectieve fout die ze beschrijven. Een beetje zoals een lector die lesgeeft over iets waar hij zelf niets van begrijpt.

7.4 Parameters doorgeven

Methoden zijn handig vanwege de herbruikbaarheid. Wanneer je een methode hebt geschreven om de sinus van een hoek te berekenen, dan is het echter ook handig dat je de hoek als parameter kunt meegeven zodat de methode kan gebruikt worden voor eender welke hoekwaarde.

Indien er wel parameters nodig zijn dan geef je die mee als volgt:

```
1 MethodeNaam(parameter1, parameter2, ...);
```

Je hebt dit ook al geregeld gebruikt. Wanneer je tekst op het scherm wilt tonen dan roep je de `WriteLine` methode aan en geef je 1 parameter mee, namelijk hetgeen dat op het scherm moet komen.

7.4.1 Methoden met formele parameters

Om zelf een methode te definiëren die 1 of meerdere parameters aanvaardt, dien je per parameter het datatype en een tijdelijk naam (*identifier*) te definiëren (*formele parameters*) in de methodesignatuur

Als volgt:

```
1 static returntype MethodeNaam(type parameter1, type parameter2)
2 {
3     //code van methode
4 }
```

Deze formele parameters zijn nu beschikbaar binnen de methode om mee te werken naar believen.

Stel bijvoorbeeld dat we onze FaculteitVan5 willen veralgemenen naar een methode die voor alle getallen werkt, dan zou je volgende methode kunnen schrijven:

```
1 static int BerekenFaculteit(int grens)
2 {
3     int resultaat = 1;
4     for (int i = 1; i <= grens; i++)
5     {
6         resultaat *= i;
7     }
8     return resultaat;
9 }
```

De naam *grens* kies je zelf. Maar we geven hier dus aan dat de methode *BerekenFaculteit* enkel kan aangeroepen worden indien er 1 actuele parameter van het type **int** wordt meegegeven.

Aanroepen van de methode gebeurt dan als volgt:

```
1 int getal = 5;
2 int resultaat = BerekenFaculteit(getal);
```

Of sneller:

```
1 int resultaat = BerekenFaculteit(5);
```

Als we even later *resultaat* dan zouden gebruiken zal er de waarde 120 in zitten.



Parameters worden “by value” meegegeven (zie het hoofdstuk over Arrays hierna) wat wil zeggen dat een **kopie** van de waarde wordt meegegeven. Als je dus in de methode de waarde van de parameter aanpast, dan heeft dit géén invloed op de waarde van de originele parameter waar je de methode aanriep.

Je zou nu echter de waarde van getal kunnen aanpassen (door bijvoorbeeld aan de gebruiker te vragen welke faculteit moet berekend worden) en je code zal nog steeds werken.



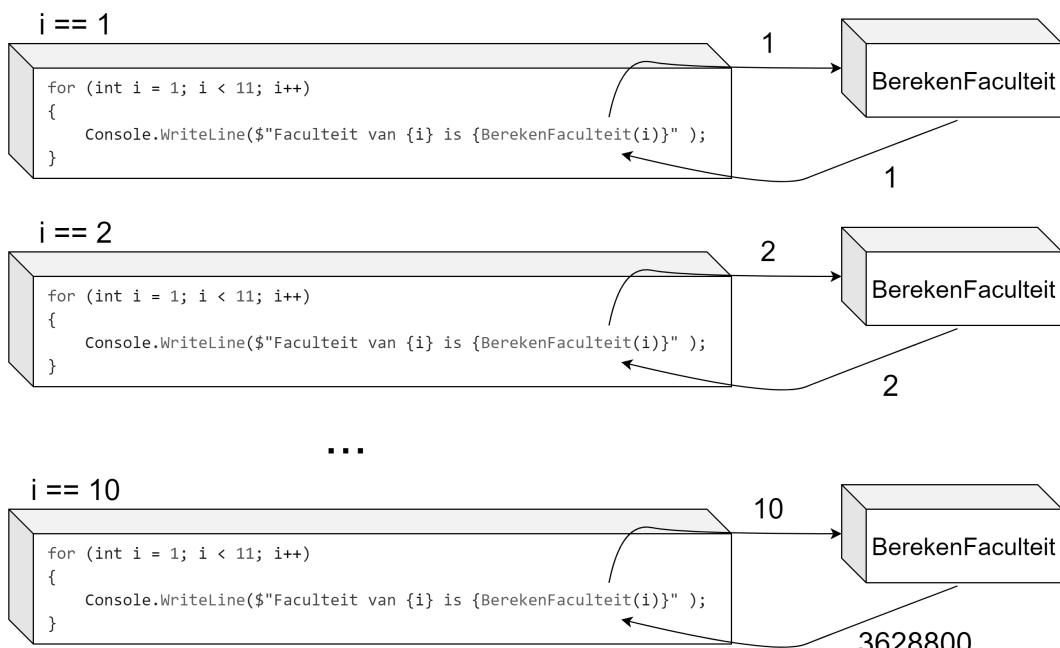
Veel beginnende programmeurs zijn soms verward dat de naam van de parameter in de methode (bv. grens) niet dezelfde moet zijn als de naam van de variabele (of literal) die we bij de aanroep meegeven.

Het is echter logisch dat deze niet noodzakelijk gelijk moeten zijn: het enige dat er gebeurt is dat de methodeparameter de waarde krijgt die je meegeeft, ongeacht van waar de parameter komt.

En wat als je de faculteiten wenst te kennen van alle getallen tussen 1 en 10? Dan zou je schrijven:

```

1  for (int i = 1; i < 11; i++)
2  {
3      Console.WriteLine($"Faculteit van {i} is {BerekenFaculteit(i)}" );
4 }
```



Figuur 7.3: Visualisatie flow.

Dit zal als resultaat geven:

```

1 Faculteit van 1 is 1
2 Faculteit van 2 is 2
3 Faculteit van 3 is 6
4 Faculteit van 4 is 24
5 Faculteit van 5 is 120
6 Faculteit van 6 is 720
7 Faculteit van 7 is 5040
8 Faculteit van 8 is 40320
9 Faculteit van 9 is 362880
10 Faculteit van 10 is 3628800

```



Merk op dat dankzij je methode, je v  l code maar   n keer moet schrijven, wat de kans op fouten verlaagt.

7.4.2 Volgorde van parameters

De volgorde waarin je je parameters meegeeft bij de aanroep van een methode is belangrijk. De eerste variabele wordt aan de eerste parameter toegekend, enz. Het volgende voorbeeld toont dit.

Stel dat je een methode hebt:

```

1 static void ToonDeling(double teller, double noemer)
2 {
3     if(noemer != 0)
4         Console.WriteLine(teller/noemer);
5     else
6         Console.WriteLine("Een zwart gat ontstaat!");
7 }

```

Deze 2 aanroepen zullen dus een andere output geven:

```

1 ToonDeling(3.5 , 2.1 );
2 ToonDeling(2.1 , 3.5 );

```

Zeker wanneer je met verschillende types als formele parameters werkt is de volgorde belangrijk. Het verschil met de vorige methode is hier wel dat VS jou zal helpen wanneer je volgorde niet klopt.

Stel dat we volgende methode hebben gemaakt:

```

1 static void ToonInfo(string name, int age)
2 {
3     Console.WriteLine($"{name} is {age} old");
4 }

```

Deze aanroep is correct:

```
1 ToonInfo("Tim", 37);
```

Maar deze is **FOUT** en zal niet compileren:

```
1 ToonInfo(37, "Tim"); //mag niet!
```

7.4.3 Doorgeven van parameters

Parameters kunnen op 2 manieren worden doorgegeven aan een methode:

1. **By value**: hierbij wordt **een kopie gemaakt van de huidige waarde**. Het is die kopie die wordt meegegeven.
2. **by reference**: het adres (**pointer of reference**) naar de actuele parameter wordt meegegeven. Aanpassingen aan de actuele parameter in de methode zal daardoor ook zichtbaar zijn binnen de scope van de originele variabele. Parameters *by reference* komen pas vanaf hoofdstuk 9 van pas².

Het effect van manier 1 is hopelijk duidelijk: wanneer je in een methode de inhoud van een actuele parameter aanpast, dan heeft dat geen gevolg op de originele variabele die we meegaven bij de methode-aanroep!

Dit zien we in dit programma:

```
1 static void JaartjeOuder(int leeftijd)
2 {
3     leeftijd++;
4     Console.WriteLine($"Hoera. Je bent {leeftijd} jaar geworden.");
5 }
6 static void Main(string[] args)
7 {
8     int mijnLeeftijd = 40;
9     Console.WriteLine($"Je bent {mijnLeeftijd} jaar.");
10    JaartjeOuder(mijnLeeftijd);
11    Console.WriteLine($"Je bent {mijnLeeftijd} jaar.");
12 }
```

In de output zien we dat `mijnLeeftijd` niet werd aangepast in de methode:

```
1 Je bent 40 jaar.
2 Hoera. Je bent 41 jaar geworden.
3 Je bent 40 jaar.
```

²Het tweede punt mag je volledig negeren als je geen flauw benul had wat er net werd gezegd. Ik kom hier later in de volgende hoofdstukken nog uitgebreid op terug!

7.4.4 Methoden nesten

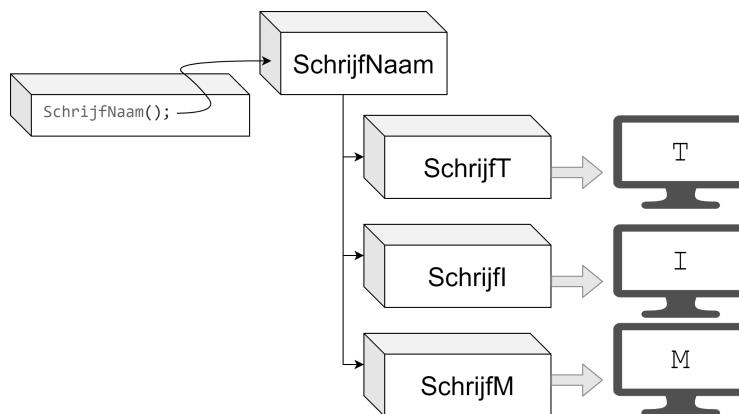
In het begin ga je vooral vanuit je Main methoden aanroepen, maar dat is geen verplichting. Je kan ook vanuit methoden andere methoden aanroepen. Je kan zelfs vanuit die aangeroepen methode weer andere aanroepen, enz.

Volgende (nutteloze) programma'tje toont dit in actie:

```

1  static void SchrijfT()
2  {
3      Console.Write("T");
4  }
5  static void SchrijfI()
6  {
7      Console.Write("I");
8  }
9  static void SchrijfM()
10 {
11     Console.Write("M");
12 }
13 static void SchrijfNaam()
14 {
15     SchrijfT();
16     SchrijfI();
17     SchrijfM();
18 }
19 public static void Main()
20 {
21     SchrijfNaam();
22 }
```

Er verschijnt “Timmi” op het scherm.



Figuur 7.4: Visualisatie van de code zonder terugkerende pijlen.

7.4.5 Bugs met methoden

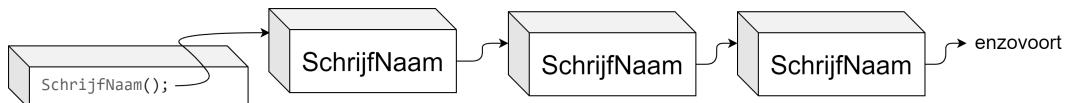
Wanneer je programma's complexer worden moet je zeker opletten dat je geen oneindige *methode-lussen* creëert. Zie je de fout in volgende code?

```

1  public static void Main()
2  {
3      SchrijfNaam();
4  }
5  static void SchrijfNaam()
6  {
7      SchrijfNaam();
8      Console.WriteLine("Klaar?");
9  }

```

Deze code heeft een methode die zichzelf aanroeft, zonder dat deze ooit afsluit. Hierdoor komen we dus in een oneindige aanroep van de methode `SchrijfNaam`. Dit programma zal een leeg scherm tonen (daar er nooit aan de tweede lijn in de methode wordt geraakt) en dan crashen wanneer het werkgeheugen van de computer op is.



Figuur 7.5: Deze keer zijn er bewust geen terugkerende pijlen getekend: ze zijn er niet.

7.4.5.1 Lokale functies...en waarom je ze beter niet gebruikt

Sinds C# 7.0 kan je methoden definiëren binnenin een andere methode. Dit noemt men **lokale functies** (*local functions*). Alhoewel ze zeker hun nut hebben, is het in deze fase van C# leren **geen goed idee om lokale functies te gebruiken**.

Het is veel belangrijker dat je eerst goed leert methoden schrijven. Beginnende programmeurs schrijven soms per ongeluk een lokale functies. Dan ontdekken ze dat ze die methode nergens kunnen aanroepen. Lokale functies zijn alleen oproepbaar binnen de methode waarin ze zijn gedefinieerd.

Kortom, zorg dat je nooit dit schrijft!

```

1  static void Main(string[] args)
2  {
3      TimVindtDitNietLeuk();
4
5      static void TimVindtDitNietLeuk() //NIET DOEN!
6      {
7          Console.WriteLine("Doe dit niet!");
8      }
9  }

```

Maar wel

```

1 static void TimVindtDitNietLeuk() //Beter!
2 {
3     Console.WriteLine("Doe dit niet!");
4 }
5
6 static void Main(string[] args)
7 {
8     TimVindtDitNietLeuk();
9 }
```



Even ingrijpen en je wijzen op recursie zodat je code niet in je gezicht blijft ontploffen.

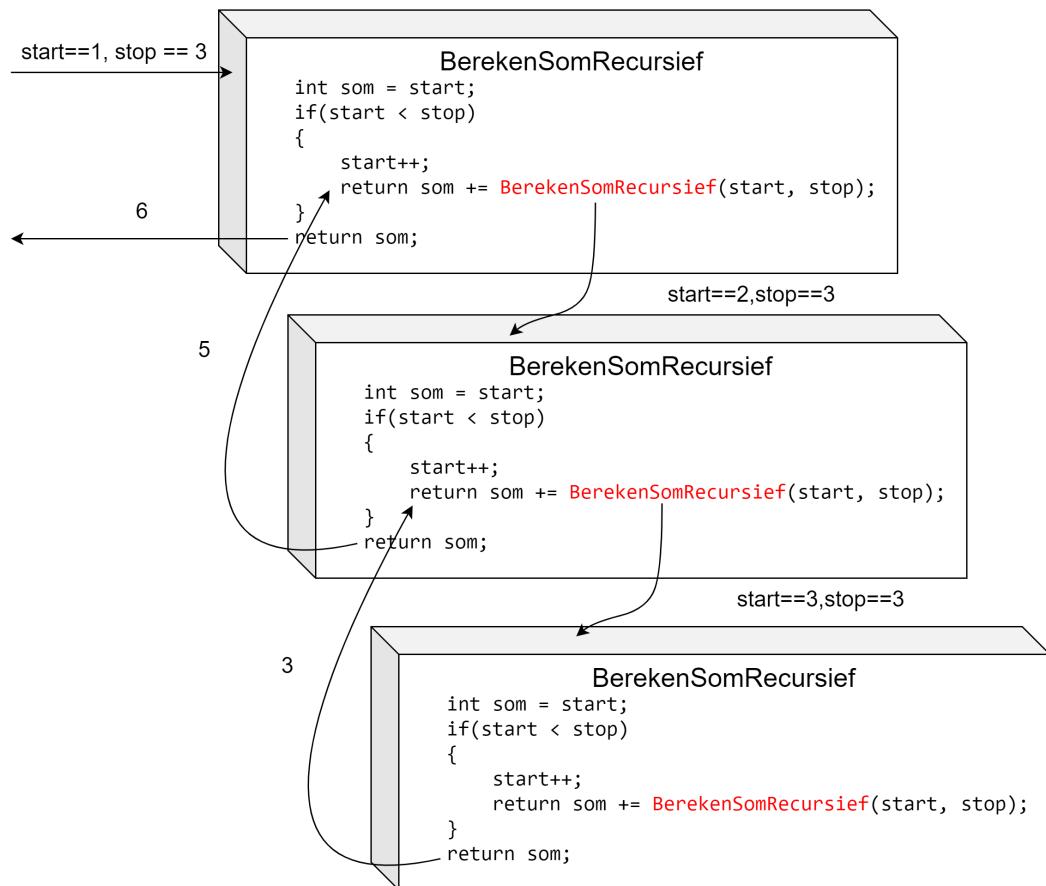
Recursie is een geavanceerd programmeerconcept wat niet in dit boek wordt besproken (enkel in hoofdstuk 18 gaan we recursie nog kort ontmoeten), maar laten we het hier kort toelichten. Recursieve methoden zijn methoden die zichzelf aanroepen maar wél op een gegeven moment stoppen wanneer dat moet gebeuren. Volgend voorbeeld is een recursieve methode om de som van alle getallen tussen start en stop te berekenen:

```

1 static int BerekenSomRecursief(int start, int stop)
2 {
3     int som = start;
4     if(start < stop)
5     {
6         start++;
7         return som += BerekenSomRecursief(start, stop);
8     }
9     return som;
10 }
```

Je herkent recursie aan het feit dat de methode zichzelf aanroept. Maar een controle voorkomt dat die aanroep blijft gebeuren zonder dat er ooit een methode wordt afgesloten. We krijgen 6 terug ($1+2+3$) als we de methode als volgt aanroepen:

```
1 int einde = BerekenSomRecursief(1,3);
```



Figuur 7.6: Flow van de recursie.

7.4.6 Commentaar aan methoden toevoegen

Het is aan te raden om steeds boven een methode een nieuwe vorm van commentaar te plaatsen als volgt (dit werkt enkel bij methoden): `///`

Visual Studio zal dan automatisch de parameters verwerken van je methode zodat je vervolgens enkel nog het doel van iedere parameter moet schrijven.

Stel dat we een methode hebben geschreven die de macht van een getal berekent (wat dom is... er bestaat al zoets als `Math.Pow`). We zouden dan volgende commentaar toevoegen:

```
1  /// <summary>
2  /// Berekent de macht van een getal.
3  /// </summary>
4  /// <param name="grondtal">Het getal dat je tot macht wilt verheffen
5  /// </param>
6  /// <param name="exponent">De exponent van de macht</param>
7  /// <returns></returns>
8  static int Macht(int grondtal, int exponent)
9  {
10     int result = grondtal;
11     for (int i = 1; i < exponent; i++)
12     {
13         result *= grondtal;
14     }
15 }
```

Wanneer we nu elders de methode `Macht` gebruiken dan krijgen we automatische extra informatie:

```
int Program.Macht(int grondtal, int exponent)
Berekent de macht van een getal.
grondtal: Het getal dat je tot macht wilt verheffen
```

Figuur 7.7: Het is aanbevolen om je documentatie in het Engels te doen, niet zoals in dit voorbeeld dus.

7.5 Bestaande methoden en bibliotheken

Laten we eens kijken naar de vele methoden die reeds ingebouwd zitten in .NET en hoe we ze nu beter kunnen gebruiken.

Sommige methoden vereisen dat je een aantal parameters meegeeft. De parameters dien je tussen de ronde haakjes te zetten. Hierbij weten we nu dat het uiterst belangrijk dat je de volgorde respecteert die de ontwikkelaar van de methode heeft gebruikt.

Maar wat als je niet weet in welke volgorde je argumenten moet meegeven? **Intellisense** is dan de oplossing! Typ de methode in je code en stop met typen na het eerste ronde haakje. Vervolgens zal Intellisense alle mogelijke manieren tonen waarop je deze methoden kan oproepen. Met de omhoog- en omlaagpijltjes van het toetsenbord kan je alle mogelijke manieren bekijken.



Figuur 7.8: Dit soort popups bevat een schat aan informatie.

In de voorgaande screenshot zien we dat Intellisense telkens duidelijke de methode-signatuur beschrijft:

- Het return type (in dit geval **void**).
- Gevolgd door de naam van de methode.
- Finaal de formele parameters en hun datatype(s).

Merk trouwens op dat je de `WriteLine`-methode ook mag aanroepen zonder parameters, dit zal resulteren in een lege lijn in de console.

Met behulp van de F1-toets kunnen we meer info over de methode in kwestie tonen. Hiervoor dien je je cursor op de Methode in je code te plaatsen, en vervolgens op F1 te drukken. Je komt dan op de online documentatie van de methode waar erg veel informatie terug te vinden is over het gebruik ervan. Scroll naar de *overload list*, daar zien we de verschillende manieren waarop je de methode in kwestie kan aanroepen (het concept *overloaden* bespreek ik in de volgende sectie). Je kan vervolgens op iedere methode klikken voor meer informatie en een codevoorbeeld.

7.5.1 Intellisense

Wat kan deze .NET bibliotheek eigenlijk? is een veelgestelde vraag. Zeker wanneer je de basis van C# onder de knie hebt en je stilletjes aan met bestaande .NET bibliotheken wilt gaan werken. Wat volgt is een essentieel onderdeel van VS dat veel gevloek en tandengeknars zal voorkomen.

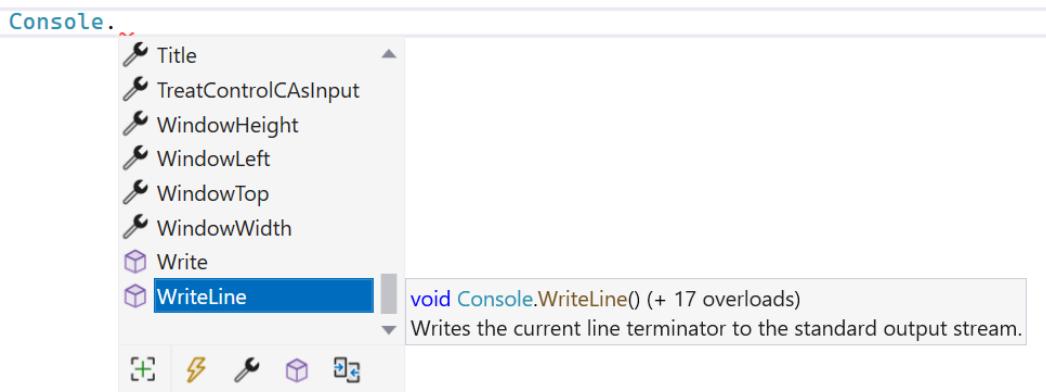
De online documentatie van VS is zeer uitgebreid en dankzij **IntelliSense** krijg je ook aardig wat informatie tijdens het typen van de code zelf. IntelliSense is de achterliggende technologie in VS die ervoor zorgt dat je minder moet typen. Als een soort assistent probeert IntelliSense een beetje te voorspellen wat je gaat typen en zal je daarmee helpen.

Type eens het volgende in:

```
1 System.Console.
```

Wacht nu even en er zal na het punt (.) een lijst komen van methoden en fields die beschikbaar zijn. Dit is IntelliSense in actie. Als er niets verschijnt of iets dat je niet had verwacht, dan is de kans groot dat er je een schrijffout hebt gemaakt.

Je kan door deze lijst met de muis doorheen scrollen en zo zien welke methoden allemaal bij de Console bibliotheek horen. Indien gewenst kan je vervolgens de gewenste methode selecteren en op spatie duwen zodat deze in je code verschijnt.



Figuur 7.9: De icoontjes geven aan of het om een methode (kubus), een eigenschap (Engelse sleutel) of een “event” (bliksem) gaat. Events behandel ik niet in dit handboek.

7.5.1.1 Herbruikbare gebruikersinvoer vragen

Vaak moet je code schrijven waarin je een getal aan de gebruiker vraagt:

```
1 Console.WriteLine("Geef leeftijd");
2 int leeftijd = int.Parse(Console.ReadLine());
```

Als deze constructie op meerdere plekken in een project voorkomt dan is het nuttig om deze twee lijnen naar een methode te verhuizen die er dan zo kan uitzien:

```
1 static int VraagInt(string zin)
2 {
3     Console.WriteLine(zin);
4     return int.Parse(Console.ReadLine());
5 }
```

De code van zonet kan je dan nu herschrijven naar:

```
1 int leeftijd = VraagInt("Geef leeftijd");
```

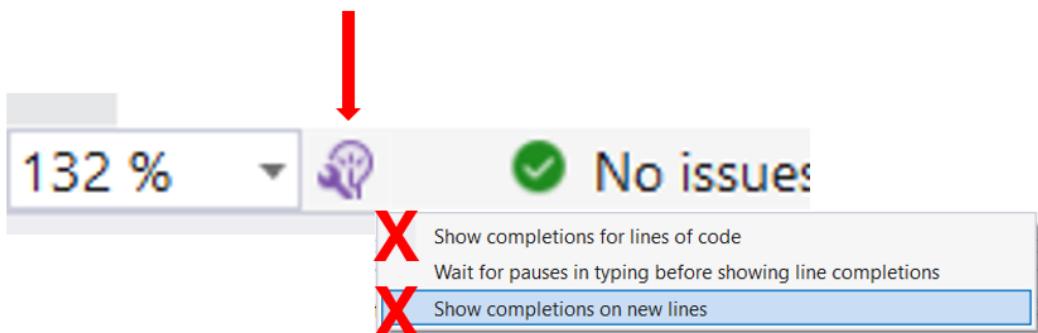
Het voorgaande voorbeeld toont ook ineens aan waarom methoden helpen om je code leesbaarder en onderhoudbaar te maken. Je Main blijft gevrijwaard van veel repeterende lijnen code en heeft aanroepen naar methoden met een duidelijke naam die ieder een specifiek ding doen. Dit maakt het debuggen ook eenvoudiger: je ziet in één oogopslag wat een methode doet. :::

7.5.2 IntelliCode

Sinds Visual Studio 2022 heeft IntelliSense een ongelooflijk krachtig broertje bijgekregen, genaamd IntelliCode. Deze tool zal ervoor zorgen dat je nog betere aanbevelingen krijgt van VS terwijl je aan het typen bent. Het gaat soms zo ver dat het lijkt alsof IntelliCode in je hoofd kan kijken en perfect kan voorspellen wat je wilt typen. **Let hier echter goed voor op:** de aanbevelingen zijn meestal erg accuraat, maar:

1. Ze zorgen ervoor dat je zelf minder moet typen en daardoor ook mogelijk jezelf niet genoeg traint. Zeker als beginnende programmeur. Ik raad je eigenlijk aan om IntelliCode uit te schakelen (via het Tools&Options menu-item). Waarom? Laten we de analogie van het leren fietsen er nog eens bijhalen. Wat IntelliCode eigenlijk doet is je af en toe optillen en enkele meters hoger op de berg plaatsen. Handig, dat wel, maar je traint je fietsbenen natuurlijk niet.
2. De aanbevelingen zijn natuurlijk soms gewoon fout of bevatten bugs die later bijvoorbeeld door hackers kunnen misbruikt worden. Of wat te denken van aanbevelingen die op zich wel zullen werken, maar wel 10x zoveel geheugen vereisen? Kortom, **wees steeds kritisch over de aanbevelingen van IntelliCode**

IntelliCode zal ook IntelliSense verbeteren door de belangrijkste, meest gebruikte methoden bovenaan te zetten. Je zal echter IntelliCode vooral herkennen wanneer er plots een hele lijn code verschijnt in het lichtgrijs. Je kan dit uitschakelen door onderaan op het kleine paarse lampje met Engelse sleutel te klikken en dan beide “Show...”-opties uit te schakelen.



Figuur 7.10: Je hebt een vergrootglas nodig om IntelliCode af te zetten...



Github Copilot project is zelfs nog krachtiger en komt dus met een nog grotere disclaimer: **beginnende programmeurs, laat dit soort tools beter nog even links liggen!** Je leert ook niet hoofdrekenen door vanaf dag 1 met een zakrekenmachine aan de slag te gaan.

7.6 Geavanceerde methode-technieken

Nu we methoden in de vingers krijgen, is het tijd om naar enkele gevorderde aspecten te kijken. Je hebt vermoedelijk al door dat methoden een erg fundamenteel concept zijn van een programmeertaal en dus hoe beter we ermee kunnen werken, hoe beter.³

7.6.1 Named parameters

Wanneer je een methode aanroeft is de volgorde van je actuele parameters belangrijk: deze moeten meegeven worden in de volgorde zoals de methode ze verwacht.

Met behulp van **named parameters** kan je echter expliciet aangeven welke actuele parameters aan welke formele parameter moet meegegeven worden.

Stel dat we een methode hebben met volgende signatuur:

```
1 static void PrintDetails(string seller, int orderNum, string product)
2 {
3     //do stuff
4 }
```

Zonder named parameters zou een aanroep van deze methode als volgt kunnen zijn:

```
1 PrintDetails("Gift Shop", 31, "Red Mug");
```

We kunnen named parameters aangeven door de naam van de parameter gevuld door een dubbel punt en de waarde. Als we dus bovenstaande methode willen aanroepen kan dat ook als volgt met named parameters:

```
1 PrintDetails(orderNum: 31, product: "Red Mug", seller: "Gift Shop");
```

of ook:

```
1 PrintDetails(product: "Red Mug", seller: "Gift Shop", orderNum: 31);
```

Kortom, op deze manier maakt de volgorde van parameter niets uit. **Dit werkt echter enkel als je alle parameters op deze manier gebruikt.**

³Dit hoofdstuk is grotendeels gebaseerd op docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments.

7.6.1.1 Named en unnamed mixen: volgorde wél belangrijk

Je mag echter ook een combinatie gebruiken van named en gewone parameters, maar **dan is de volgorde belangrijk**: je moet je dan houden aan de volgorde van de methode-signatuur. Je verbetert hiermee de leesbaarheid van je code, maar krijgt niet het voordeel van een eigen volgorde te hanteren. Enkele **geldige** voorbeelden:

```
1 PrintDetails("Gift Shop", 31, product: "Red Mug");
2 PrintDetails(seller: "Gift Shop", 31, product: "Red Mug");
```

Enkele **niet geldige** voorbeelden:

```
1 PrintDetails(product: "Red Mug", 31, "Gift Shop");
2 PrintDetails(31, seller: "Gift Shop", "Red Mug");
```

7.6.2 Optionele parameters

Soms wil je dat een methode een standaardwaarde voor een parameter gebruikt indien de programmeur in z'n aanroep geen waarde meegaf. Dat kan met behulp zogenaamde van **optionele of default parameters**. Je geeft aan dat een parameter optioneel is door deze een default waarde te geven in de methode-signatuur. Deze waarde zal dan gebruikt worden indien de parameter geen waarde van de aanroeper heeft gekregen.

Let op: **Optionele parameters worden steeds achteraan de parameterlijst van de methode geplaatst.**

In het volgende voorbeeld maken we een nieuwe methode aan en geven aan dat de laatste twee parameters (optName en age) optioneel zijn door er met de toekenningsoperator een default waarde aan te geven:

```
1 static void BookFile(int required, string optName = "unknown", int
    age = 10)
```

Wanneer nu een parameter niet wordt meegegeven, dan zal deze default waarde in de plaats gebruikt worden:

```
1 BookFile(15, "tim", 25);
2 BookFile(20, "dams"); //age zal 10 zijn, optName "dams"
3 BookFile(35); //optName zal "unknown" en age zal 10 zijn
```

De inhoud van argumenten wordt bij iedere aanroep:

	required	optName	age
Lijn 1	15	"tim"	25
Lijn 2	20	"dams"	10
Lijn 3	35	"unknown"	10

Je mag enkel de optionele parameters van achter naar voor weglaten. Volgende aanroep is dus niet geldig:

```
1 BookFile(3, 4); //daar de tweede param een string moet zijn
```

Met optionele parameters kunnen we dit omzeilen. Volgende aanroep is wel geldig:

```
1 BookFile(3, age: 4);
```

7.6.3 Method overloading

Method overloading wil zeggen dat je een **methode met dezelfde naam en returntype** meerdere keren definiert *maar met andere formele parameters qua datatype en/of aantal*. De compiler zal dan zelf bepalen welke versie moet aangeroepen worden, gebaseerd op het aantal en type actuele parameters dat je meegeeft.

Volgende methoden zijn overloaded:

```

1 static int BerekenOpp(int lengte, int breedte)
2 {
3     int opp = lengte*breedte;
4     return opp;
5 }
6
7 static int BerekenOpp(int straal)
8 {
9     int opp = (int)(Math.PI*straal*straal);
10    return opp;
11 }
```

Afhankelijk van de aanroep zal dus de ene of andere methode uitgevoerd worden. Volgende code zal dus werken:

```

1 Console.WriteLine($"Rechthoek: {BerekenOpp(5, 6)}");
2 Console.WriteLine($"Cirkel: {BerekenOpp(7)}");
```

7.6.3.1 Betterness rule

Indien de compiler twijfelt tijdens de **overload resolution** zal de **betterness rule** worden gehanteerd: de best *passende* methode zal aangeroepen worden.

Stel dat we volgende overloaded methoden hebben:

```

1 static int BerekenOpp(int straal) //versie A
2 {
3     int opp = (int)(Math.PI*straal*straal);
4     return opp;
5 }
6 static int BerekenOpp(double straal) //versie B
7 {
8     int opp = (int)(Math.PI * straal * straal);
9     return opp;
10 }
```

Volgende aanroepen zullen dus als volgt uitgevoerd worden, gebaseerd op de betterness rule:

```

1 Console.WriteLine($"Cirkel 1: {BerekenOpp(7)}"); //versie A
2 Console.WriteLine($"Cirkel 2: {BerekenOpp(7.5)}"); //versie B
3 Console.WriteLine($"Cirkel 3: {BerekenOpp(7.3f)}"); //versie B
```

Volgende tabel geeft de betternes rule weer. In de linkse kolom staat het datatype van de parameter die wordt meegegeven. De rechtse kolom toont welk datatype het argument in de methodesignatuur meer voorkeur heeft van links naar rechts indien dus het originele type niet beschikbaar is.

Parameter	Van meeste voorkeur naar minste
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Als je bijvoorbeeld een parameter van het type **int** meegeeft bij een methode-aanroep (eerste kolom), dan zal een methode waar het argument een **long** verwacht geprefereerd worden boven een methode die voor datzelfde argument een **float** verwacht, enz.

Indien de betterness rule niet werkt, dan zal de eerste parameter bepalen wat er gebruikt wordt. Dat zien we in volgende voorbeeld:

```

1 static void Main(string[] args)
2 {
3     Toonverhouding(5, 3.4); //versie A
4     Toonverhouding(6.2, 3); //versie B
5 }
6
7 static void Toonverhouding(int a, double b) //versie A
8 {
9     Console.WriteLine($"{a}/{b}");
10}
11
12 static void Toonverhouding(double a, int b) //versie B
13 {
14     Console.WriteLine($"{a}/{b}");
15 }
```

Indien ook die regel niet werkt dan zal volgende foutmelding verschijnen:



CS1503 Argument 1: cannot convert from 'double' to 'int'

Figuur 7.11: We zien aan de foutbericht duidelijk dat er eerst naar de eerste parameter wordt gekeken bij twijfel.

```
1 static void Main(string[] args)
2 {
3     Toonverhouding(5.6, 3.4);
4 }
5
6 static void Toonverhouding(int a, double b)
7 {
8     Console.WriteLine($"{a}/{b}");
9 }
10
11 static void Toonverhouding(double a, int b)
12 {
13     Console.WriteLine($"{a}/{b}");
14 }
```

7.6.4 Methoden debuggen met step-in

Herinner je je dat ik in hoofdstuk 4 debuggen uitlegde en zei dat we één knopje later gingen bekijken? Wel die tijd is nu gekomen. Tijd om de **step in** knop toe te lichten.



Figuur 7.12: Je vindt deze knop bovenaan in je menu wanneer je in debug-modus bent

Wanneer je een breakpoint zet in je code en in debugmodus komt dan kan je doorheen je code *stappen*, wat je hopelijk al geregeld hebt gedaan. Het nadeel was dat je niet **in** een methode ging wanneer je daar *over stapte*. Wel, met de “step in” knop kan je dat nu wel. Wanneer je aan een lijn met een **eigen geschreven** methode komt dan zorgt deze knop ervoor dat je in de methode gaat en vervolgens daar verder kunt stappen over de verschillende lijnen code.

Het klinkt simpel, maar oefen het toch best een paar keer!

8 Arrays

Arrays zijn een veelgebruikt principe in vele programmeertalen. Het grote voordeel van arrays is dat je één enkele variabele kunt hebben die een grote groep waarden voorstelt van eenzelfde type. Hierdoor wordt je code leesbaarder en eenvoudiger in onderhoud. Arrays zijn een zeer krachtig hulpmiddel, maar er zitten wel enkele venijnige addertjes onder het gras.

Op papier zijn arrays eenvoudig... helaas programmeren we zelden nog op papier. Eigenlijk is een array niets meer dan **een verzameling waarden van hetzelfde datatype**. Deze aparte waarden kunnen benaderd worden via 1 enkele variabele, de array zelf. Door middel van een **index** kan ieder afzonderlijk element uit de array aangepast of uitgelezen worden.

Een nadeel van arrays is dat, eens we de lengte van een array hebben ingesteld, deze lengte niet meer kan veranderd worden. In het hoofdstuk 12 zullen we leren werken met lists en andere collections die dit nadeel niet meer hebben.

De nadelen zullen we echter met plezier erbij nemen wanneer we programma's beginnen schrijven die werken met vél data van dezelfde soort: eenvoudigweg kan je stellen dat van zodra je 3 of meer variabelen hebt die dezelfde soort data bevatten (en dus van hetzelfde datatype zijn), een array bijna altijd de oplossing zal zijn.

8.1 Nut van arrays

Stel dat je de dagelijkse neerslag wenst te bewaren om zo later de gemiddelde regen te berekenen. Dit kan je zonder arrays eenvoudig:

```
1 int dag1 = 34;  
2 int dag2 = 45;  
3 int dag3 = 0;  
4 int dag4 = 34;  
5 int dag5 = 12;  
6 int dag6 = 0;  
7 int dag7 = 23;
```

Als we je nu vragen om de gemiddelde neerslag te berekenen dan krijg je al een redelijk lang statement:

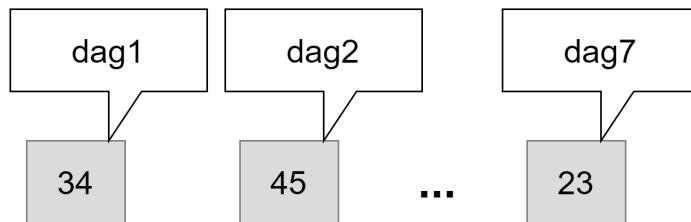
```
1 double gemiddelde = (dag1+dag2+dag3+dag4+dag5+dag6+dag7)/7.0;
```

Maar wat als je plots de neerslag van een heel jaar wenst te bewaren. Of een hele eeuw? Of een millennium?! Van zodra je een bepaalde soort informatie hebt die je veelvuldig wenst te bewaren dan zijn arrays dus de oplossing.

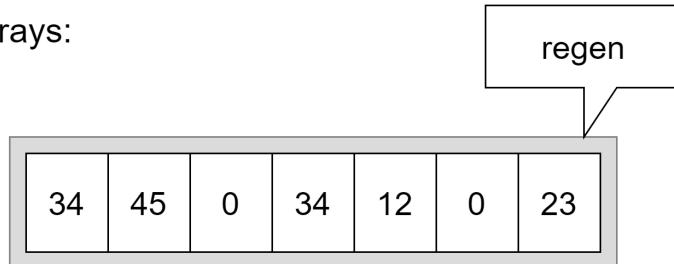
Voorgaande lijst van 7 aparte variabelen kunnen we eenvoudiger definiëren met 1 array (we bespreken de details verderop), genaamd regen:

```
1 int[] regen = {34, 45, 0, 34, 12, 0, 23};
```

Zonder arrays:



Met arrays:



Figuur 8.1: Een schematische voorstelling van een lijst van aparte variabelen en het equivalent met een array.

Het gemiddelde berekenen kan dan als volgt:

```
1 double gemiddelde = (regen[0]+regen[1]+regen[2]+regen[3]+regen[4]+  
    regen[5]+regen[6])/7.0;
```

Dat lijkt niet veel beter...Integendeel. We zitten nu ook nog met een hoop vierkante haakjes ([]) in onze code.

De kracht van arrays komt nu: het getal tussen die vierkante haakjes (de index) kan je als een variabele beschouwen en dus ook dynamisch genereren in een loop. Volgend voorbeeld toont hoe we bijvoorbeeld een langere array van elementen met een for-loop overlopen om de som van alle elementen te berekenen:

```
1 int[] regen = {34, 45, 0, 34, 12, 0, 23, 7, 20, 34, 7, 42}; //  
    aanmaken array  
2 double som = 0;  
3 for(int i = 0; i<regen.Length;i++)  
4 {  
5     som += regen[i]; //element per element uit array optellen  
6 }  
7 double gemiddelde = som/regen.Length;
```



Sorry dat we weer even in het diepe water zijn gedoken. Het leek ons nuttig om even het totaalplaatje van arrays alvast uit de doeken te doen, zodat je snapt waarom er hier zo enthousiast over arrays wordt gedaan.

A propos, kijk eens achterom! Schrik je van hé. Je hebt al een aardige weg afgelegd als we vergelijken met de eerste keer toen ik je in het zwembad gooide.

Alles wordt kinderspel, als je maar lang genoeg met iets bezig bent. Zelfs de code die we net toonden met die arrays zou je niet meer zo erg mogen afschrikken als die eerste keer. Ok, er staan wat nieuwe termen tussen, maar al bij al zouden de grote lijnen van het algoritme en de werking ervan duidelijk moeten zijn.

Blijf dus maar hier lekker in het diep dobberen en ontdek verder waarom arrays zo'n krachtig concept zijn.

8.2 Werken met arrays

8.2.1 Arrays declareren

Een array creëren (declareren) kan op 3 manieren.

8.2.1.1 Manier 1

De eenvoudigste variant is deze waarbij je een array variabele aanmaakt, maar deze nog niet initialiseert. Je maakt enkel een identifier aan, maar zet er nog niets in. De syntax is als volgt:

```
1 type[] arraynaam;
```

Type kan eender welk bestaand datatype zijn dat je reeds kent. De [] (vierkante haken of *square brackets*) duiden aan dat het om een array gaat.

Voorbeelden van array declaraties kunnen dus bijvoorbeeld zijn:

```
1 int[] verkoopCijfers;
2 double[] gewichtHuisdieren;
3 bool[] examenAntwoorden;
4 ConsoleColor[] mijnKleuren;
```

Op dit punt bestaan de arrays nog niet. **Hun lengte ligt nog niet vast.** In het geheugen is enkel een klein stukje geheugen gereserveerd voor een toekomstige referentie (of pointer) naar een array (wat we zo meteen gaan uitleggen).

Stel dat je een array van strings wenst waarin je verschillende kleuren zal plaatsen dan schrijf je:

```
1 string[] myColors;
```

Vervolgens kunnen we later waarden toekennen aan de array:

```
1 string[] myColors;
2 myColors = {"red", "green", "yellow", "orange", "blue"};
```

Je array zal na lijn 2 **een lengte van 5 hebben en kan niet meer groeien of krimpen.**

8.2.1.2 Manier 2

Indien je ogenblikkelijk waarden wilt toekennen (*initialiseren*) tijdens het aanmaken van de array zelf dan mag dit ook als volgt:

```
1 string[] myColors = {"red", "green", "yellow", "orange", "blue"};
```

Ook hier zal na lijn 1 je array een vaste lengte van 5 elementen hebben.

Merk op dat deze manier dus enkel werkt indien je reeds weet welke waarden in de array moeten. In manier 1 kunnen we perfect een array aanmaken en pas veel later in het programma ook effectief waarden toekennen (bijvoorbeeld door ze stuk per stuk door een gebruiker te laten invoeren).

8.2.1.3 Manier 3

Nog een andere manier om arrays aan te maken is diegene waarbij je aangeeft hoe groot de array moet zijn. We gaan echter nog niet effectief waarden in de array plaatsen.

```
1 string[] myColors;  
2 myColors = new string[5];
```

Uiteraard kan dit ook in 1 stap:

```
1 string[] myColors = new string[5];
```

We geven hier aan dat de array vanaf z'n prille bestaan 5 elementen kan bevatten. Deze elementen zullen allemaal de defaultwaarde van hun datatype krijgen. In het geval van **string** hier zal de array dus 5 lege string-elementen bevatten (" " of **string**.Empty).



Ook hier geldt dat de lengte vanaf dan vastligt en niet meer kan veranderen.



Er is een essentieel verschil tussen manier 1 en 3. Wanneer je bij de sectie “Geheugengebruik bij arrays” zal je dit ontdekken. Spoiler: in manier 1 wordt er nooit een array aangemaakt. In manier 2 wél.

8.2.2 Elementen van een array aanpassen en uitlezen

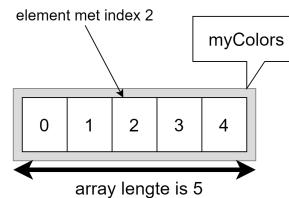
Van zodra er waarden in een array staan of moeten bijgeplaatst worden kan je deze benaderen met de zogenaamde **array accessor** notatie. Deze notatie is heel eenvoudigweg de volgende:

```
1 myColors[2]; //element met index 2
```

We plaatsen de naam van de array, gevolgd door vierkante haakjes waarbinnen een getal, 2 in dit voorbeeld, aangeeft het hoeveelste element we wensen te benaderen (lezen en/of schrijven). Deze nummering start vanaf 0.



De index van een C#-array start steeds bij 0. Indien je dus een array aanmaakt met lengte 5 dan heb je de indices 0 tot en met 4.



Figuur 8.2: Lengte is 5, index laatste element is 4, eerste element is 0.



Veelgemaakte fouten bij arrays gebeuren op de lengte en indexering ervan.

Het gebeurt vaak dat beginnende programmeurs verward geraken omtrent het aanmaken van een array aan de hand van de lengte en het indexeren erna. Maar niet getreurd, ik zal je hier extra tips geven.

De regels zijn duidelijk:

- Bij het maken van een array is de lengte van een array gelijk aan het aantal elementen dat er in aanwezig is. Dus een array met 5 elementen heeft als lengte 5.*
- Bij het schrijven en lezen van individuele elementen uit de array (zie hierna) gebruiken we een indexering die start bij 0. Bijgevolg is 4 de index van het laatste element in een array met lengte 5.*

8.2.2.1 Lezen

Je weet nu hoe je individuele waarden in een array kan benaderen. Ze gebruiken is exact hetzelfde zoals we in het verleden al met eender welke andere variabele hebben gedaan. Het enige verschil is dat de identifier vierkante haken met een index in bevat om aan te geven welke element we nodig hebben van de array.

Wanneer je dus het tweede element van een array wenst te gebruiken kan dit bijvoorbeeld als volgt:

```
1 Console.WriteLine(myColors[1]);
```

of ook

```
1 string kleurkeuze = myColors[1];
```

of zelfs

```
1 if(myColors[1] == "pink")
```

Kortom, alles wat je al kon, kan ook met arrays. Je kan ze zelfs als parameters aan methoden meegeven of terugkrijgen (zie verder). **De individuele elementen in een array zijn gewone variabelen** (enkel hun naamgeving is gekoppeld aan die van de array en de index van het element in de array).



Een array proberen te tonen als volgt gaat niet:

```
1 Console.WriteLine(myColors);
```

De enige manier alle elementen van een array te tonen is door manueel ieder element individueel naar het scherm te sturen. Bijvoorbeeld:

```
1 for(int i = 0 ; i<myColors.Length;i++)
2 {
3     Console.WriteLine($"{myColors[i]}");
4 }
```

Stel dat we een array van getallen hebben, dan kunnen we bijvoorbeeld 2 waarden uit die array optellen en opslaan in een andere variabele als volgt:

```
1 int[] numbers = {5, 10, 30, 45};
2 int som = numbers[0] + numbers[1];
```

De variabele som zal vervolgens de waarde 15 bevatten (5+10).

Stel dat we *alle* elementen uit de array `numbers` met 5 willen verhogen, dan kunnen we schrijven:

```
1 int[] numbers = {5, 10, 30, 45};
2 numbers[0] += 5;
3 numbers[1] += 5;
4 numbers[2] += 5;
5 numbers[3] += 5;
```

Maar eigenlijk zijn we dan het voordeel van arrays niet aan het gebruiken. Met loops maken we bovenstaande oplossing beter zodat deze zal werken, ongeacht het aantal elementen in de array¹:

```
1 for(int teller = 0; teller < numbers.Length; teller++)
2 {
3     numbers[teller] += 5;
4 }
```

8.2.2.2 Schrijven

Ook schrijven van waarden naar een array gebruikt dezelfde notatie. Enkel moet je dus deze keer de array accessor-notatie links van de toekenningsoperator plaatsen. Stel dat we bijvoorbeeld de waarde van het eerste element uit de `myColors` array willen veranderen van `red` naar `indigo`, dan gebruiken we volgende notatie:

```
1 myColors[0] = "indigo";
```

Als we bij aanvang nog niet weten welke waarden de individuele elementen moeten hebben in een array, dan kunnen we deze eerst definiëren, en vervolgens individueel toekennen:

```
1 string[] myColors;
2 myColors = new string[5];
3 // ...
4 myColors[0] = "red";
5 myColors[1] = "green";
6 myColors[2] = "yellow";
7 myColors[3] = "orange";
8 myColors[4] = "blue";
```

¹Zoals je merkt zijn loops en arrays dikke vrienden.



Een veel gestelde vraag wanneer een programmeur het nut van arrays nog niet 100% ziet is het volgende. Stel dat je deze code hebt;

```

1 int dag1 = 34;
2 int dag2 = 45;
3 int dag3 = 0;
4 int dag4 = 34;
5 int dag5 = 12;
6 int dag6 = 0;
7 int dag7 = 23;
```

“Kan ik die namen (dag1, dag2, enz.) met een loop genereren/bereiken zodat ik iets kan doen als volgt?” **OPGELET! Hier komt een zeer fout voorbeeld aan...**

```

1 for(int i=1; i<=7; i++)
2     dagi = ...
```

Dat gaat niet! Van zodra je van plan bent om variabele-namen “dynamisch” in je code te proberen aan te roepen, moeten er tal van alarmbelletjes afgaan. De kans is dan héél groot dat je probleem beter met een array wordt opgelost dan met een boel variabelen met soortgelijke namen.

8.2.3 De lengte van de array te weten komen

Soms kan het nodig zijn dat je in een later stadium van je programma de lengte van je array nodig hebt. De Length-eigenschap van iedere array geeft dit weer. Volgend voorbeeld toont dit:

```

1 string[] myColors = {"red", "green", "yellow", "orange", "blue"};
2 Console.WriteLine($"Length of array = {myColors.Length}" );
```

De Length-eigenschap wordt vaak gebruikt in for/while loops waarmee je de hele array wenst te doorlopen. Door de Length-eigenschap te gebruiken als grenscontrole verzekeren we er ons van dat we nooit buiten de grenzen van de array zullen lezen of schrijven:

```

1 //Alle elementen van een array tonen
2 for (int i = 0; i < getallen.Length; i++)
3 {
4     Console.WriteLine(getallen[i]);
5 }
```



Elementen benaderen buiten de range van een array geeft erg dikke errors. Het jammer is dat VS dit soort subtile ‘out of range’ bugs niet kan detecteren tijdens het compileren. Je zal ze pas ontdekken bij de uitvoer. Volgende code zal perfect gecompileerd worden, maar bij de uitvoer zal er op lijn 2 een foutbericht verschijnen en het programma zal stoppen:

```
1 int[] getallen = { 1,2,3 };  
2 Console.WriteLine(getallen[5]);
```

Dit zal resulteren in een “Out of Range exception”.

Hackers misbruiken dit soort fouten in code om toegang tot delen van het geheugen te krijgen waar ze eigenlijk niet mochten zijn. Dit zijn zogenaamde *buffer overflow attacks*.



*Sorry dat ik je al weer lastig val. Maar ik wil je nog eens extra goed naar bovenstaande fout (exception) laten kijken. Prent die **Out of Range fout** goed in je hoofd.*

*Deze fout zegt exact wat er mis is: **je probeert elementen in een array te benaderen die niet bestaan omdat je buiten het bereik (range) van de array bent gegaan.***

Momenteel werken we aan een gebouw met 3 verdiepingen (.Length is dus 3). Het is hetzelfde als wanneer ik tegen mijn personeel zeg: “ga jij de muur alvast metsen op de zesde verdieping (gebouw[5])”. Hij zal dan vermoedelijk van het gebouw vallen en nog net kunnen roepen: “Out of Range exception!!!!”.

8.2.4 Volledig voorbeeldprogramma met arrays

Met al de voorgaande informatie is het nu mogelijk om vlot complexere programma's te schrijven, die veel data moeten kunnen verwerken. Meestal gebruikt men een for-loop om een bepaalde operatie over de hele array toe te passen.

Het volgende programma zal een array van integers aanmaken die alle gehele getallen van 0 tot 99 bevat. Vervolgens zal ieder getal met 3 vermenigvuldigd worden. Finaal tonen we enkel die getallen die een veelvoud van 4 zijn na de bewerking.

```

1 //Array aanmaken
2 int[] getallen = new int[100];
3 //Array vullen
4 for (int i = 0; i < getallen.Length; i++)
5 {
6     getallen[i] = i;
7 }
8 //Alle elementen met 3 vermenigvuldigen
9 for (int i = 0; i < getallen.Length; i++)
10 {
11     getallen[i] = getallen[i] * 3;
12 }
13 //Enkel veelvouden van 4 op het scherm tonen
14 for (int i = 0; i < getallen.Length; i++)
15 {
16     if(getallen[i] % 4 == 0)
17         Console.WriteLine(getallen[i]);
18 }
```

8.2.5 Opstartparameters via args

Begrijp je nu wat `string[] args` wil zeggen in je Main? Iedere Main heeft volgende methodesignatuur:

```
1 static void Main(string[] args)
```

De args arrays kunnen we in ons programma uitlezen om eventuele opstartparameters te verwerken die de gebruiker meegaf bij de opstart van het programma. Ik heb dit nog nooit *in-depth* uitgelegd, maar laten we eens kijken hoe je dit doet.

Volg daarom volgende stappenplan:

1. Maak een nieuw console-project aan genaamd `argstest`.
2. Voeg volgende code toe in je `Main`:

```

1 for (int i = 0; i < args.Length; i++)
2 {
3     Console.WriteLine(args[i]);
4 }
5
6 if (args.Length>=2 && args[2]=="cool")
7 {
8     Console.WriteLine("Ik ga akkoord!");
9 }
```

3. Compileer je programma. Run het gerust al eens, je zal zien dat het programma nog niet veel doet. Waarom? Omdat we geen opstartparameters hebben meegegeven. Laten we dat oplossen!
4. Ga via je verkenner naar je project-folder (vanuit VS kan dit snel door in de solution Explorer te rechterklikken op je project en dan de optie “Open folder in Explorer” te kiezen).
5. Open de `bin` folder, en open daarin dan de `debug` folder, gevuld door de `net8.0` folder (die laatste kan mogelijk anders zijn, afhankelijk van welke .NET versie je gebruikt). Hier staat je gecompileerde programma. In principe kan je hier dubbelklikken op je applicatie, maar dat zal niet veel doen, daar we nog steeds geen opstartparameters hebben meegegeven.
6. Nu goed opletten: klik in je verkenner bovenaan in de adresbalk, rechts van de tekst (niet er op). Je kan nu zelf iets intypen. Typ nu `cmd` in en druk enter.
7. Cool he. Je zit nu in een shell in de juiste folder.
8. Nu kan je je programma runnen mét opstartparameters. Kijk maar eens wat er gebeurt als je typt: `argstest ziescherp is cool`

```
C:\Users\damst\source\repos\argstest\argstest\bin\Debug\net8.0>argstest ziescherp is cool
ziescherp
is
cool
Ik ga akkoord!
```

Inderdaad. De spaties gelden als “splittings” tussen ieder argument. En dus ieder woord zal een apart element in de `args` array worden. Je zou nu bijvoorbeeld code kunnen schrijven die iets doet afhankelijk van de parameter, enz.

Let er zeker op dat je steeds met `args.Length` test of er wel genoeg opstartargumenten werden meegegeven. Daarom dat we `args.Length>=2 && args[2]=="cool"` schreven.



De volgorde van operanden bij een `&&` operator zijn belangrijk. Kijk wat er gebeurt als we de operanden omwisselen in de vorige `if`:

```
1 if ( args[2]=="cool" && args.Length>=2)
```

Als je deze versie uitvoeren met minder dan 3 opstartparameters, dan zal de applicatie crashen. Waarom? De `&&`-operator werkt van links naar rechts en zal stoppen met testen indien de linkse operand reeds `false` teruggeeft. Door dus eerst te testen of de lengte klopt, komen we enkel bij de `args[2]`-code als die ook effectie kan aangeroepen worden.

Kortom: denk ook steeds goed na in welke volgorde je je conditionele testen beschrijft.

8.3 Geheugengebruik bij arrays

Met arrays komen we voor het eerst iets dichter tot één van de sterktes van C#, namelijk het aspect **referenties**. Vanaf het volgende hoofdstuk zullen we hier ongelooflijk veel mee doen, maar laten we nu alvast eens kijken waarom arrays met referenties werken.

8.3.1 Reference types en value types

We zagen reeds bij methoden dat variabelen eigenlijk op 2 manier kunnen doorgegeven worden, *by reference of by value*. We herhalen dat hier nog eens:

- **Value** types: deze variabelen bevatten effectief de waarde die de variabele moet hebben. Als we schrijven `int age = 5`, dan bewaren we de binaire voorstelling voor het geheel getal 5 in het geheugen.
- **Reference** types: deze variabelen bewaren een geheugenadres naar een andere plek in het geheugen waar de effectieve waarde(n) van de variabele te vinden is. Reference types zijn als het ware een wegwijzer en worden ook soms **pointers** genoemd.



Alle datatypes die we tot nog toe zagen - `string` is een speciaal geval en negeren we om nachtmerries te vermijden- werken stevast *by value*. Momenteel zijn het enkel arrays die we kennen die *by reference* werken in C#. In het volgende hoofdstuk zullen we zien dat er echter nog een hele hoop andere mysterieuze dingen (gennaamd *objecten*) zijn die ook *by reference* werken.

Arrays worden steeds by reference in een variabele bewaard! Dat wil dus zeggen dat we niet de array zelf toekennen aan een variabele, maar wel het geheugenadres naar de array. Dit heeft natuurlijk gevolgen op de manier dat bijvoorbeeld de toekenning-operator (=) werkt bij arrays.

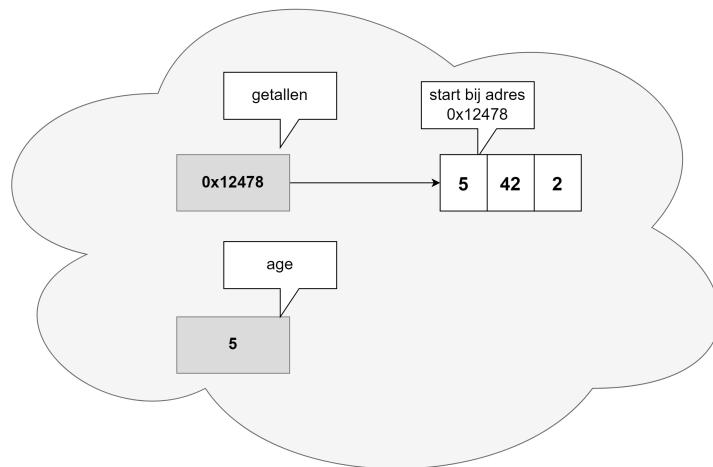
8.3.2 Arrays kopiëren

8.3.2.1 Het probleem als je arrays wilt kopiëren

Kijk wat er gebeurt bij volgende code:

```
1 int[] getallen = {5,42,2};  
2 int age = 5
```

In `getallen` bewaren enkel een geheugenadres bewaren dat wijst naar de plek waar de effectieve waarden staan elders in het geheugen. Terwijl in `age` effectief de waarde "5" zal bewaard worden. De afbeelding op volgende pagina geeft dit weer.



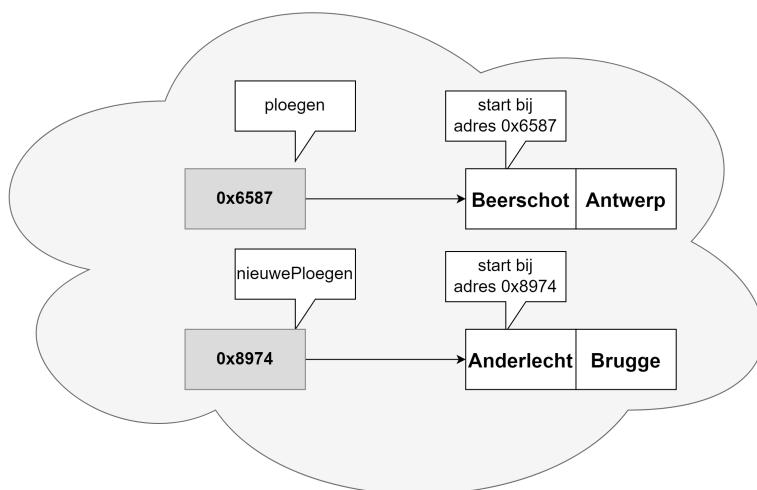
Figuur 8.3: De wolk stelt het werkgeheugen voor. De geheugenadressen zijn willekeurig.

Het gevolg van voorgaande is dat volgende code niet zal doen wat je vermoedelijk wenst:

```

1 string[] ploegen = {"Beerschot", "Antwerp"};
2 string[] nieuwePloegen = {"Anderlecht", "Brugge"};
3 nieuwePloegen = ploegen;
  
```

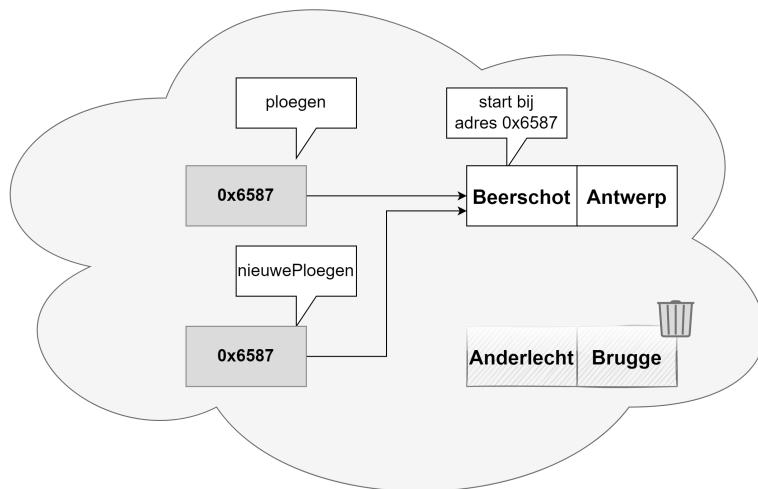
De situatie wanneer lijn 2 werd uitgevoerd is de volgende:



Figuur 8.4: Beerschot is de ploeg van't stad ;)

Zonder het bestaan van *references* zou je verwachten dat op lijn 3 nieuwePloegen een kopie krijgt van de inhoud van ploegen. Dat is dus niet zo.

Lijn 3 zal perfect werken. Wat er echter is gebeurd, is dat we de referentie naar ploegen ook in nieuwePloegen hebben geplaatst. **Bijgevolg verwijzen beide variabelen naar dezelfde array, namelijk die waar ploegen al naar verwees.** We hebben een soort *alias* gemaakt en kunnen nu op twee manieren de array met de Antwerpse voetbalploegen benaderen. De nieuwe situatie na lijn 3 is dus de volgende geworden:



Figuur 8.5: Beerschot is nog steeds de ploeg van't stad!

Als je vervolgens schrijft:

```
1 nieuwePloegen[1] = "Beerschot";
```

Dan is dat hetzelfde als onderstaande schrijven daar beide variabele naar dezelfde array-inhoud verwijzen. Het effect zal dus hetzelfde zijn.

```
1 ploegen[1] = "Beerschot";
```

En waar staan de ploegen in de nieuwePloegen array ("Anderlecht" en "Brugge")? **Die array in het geheugen is niet meer bereikbaar** (de garbage collector zal deze ten gepaste verwijderen, wat in hoofdstuk 10 zal toegelicht worden).

8.3.2.2 De oplossing als je arrays wilt kopiëren

Wil je arrays kopiëren dan kan dat **niet** als volgt:

```
1 string[] ploegen = {"Beerschot", "Antwerp"};
2 string[] nieuwePloegen = {"Anderlecht", "Brugge"};
3 nieuwePloegen = ploegen; //FAIL!!!
```

Je moet manueel ieder individueel element van de ene naar de andere array kopiëren als volgt:

```
1 for(int i = 0; i < ploegen.Length; i++)
2 {
3     nieuwePloegen[i] = ploegen[i];
4 }
```



Er is een ingebouwde methode in de `Array`-bibliotheek (deze bibliotheek zien we in de volgende sectie) die ook toelaat om arrays te kopiëren genaamd `Copy`.



Wanneer je met arrays van objecten (zie hoofdstuk 12) werkt dan zal bovenstaande mogelijk niet het gewenste resultaat geven daar we nu ook de individuele referenties van een object kopiëren!

8.4 System.Array

Je kan de `System.Array` bibliotheek gebruiken om je array-code te vereenvoudigen. Deze bibliotheek bevat naast de `.Length` eigenschap, ook enkele nuttige methoden zoals `BinarySearch()`, `Sort()`, `Copy` en `Reverse()`. Het gebruik hiervan is bijna steeds hetzelfde zoals volgende voorbeelden tonen.

8.4.1 Sort: Arrays sorteren

Om arrays te sorteren roep je de `Sort()`-methode op en geef je als parameter de array mee die gesorteerd moet worden. Volgend voorbeeld toont hier het gebruik van:

```
1 string[] myColors = {"red", "green", "yellow", "orange", "blue"};
2 Array.Sort(myColors); //Sorteren maar
3 //Toon resultaat van sorteren
4 for (int i = 0; i < myColors.Length; i++)
5 {
6     Console.WriteLine(myColors[i]);
7 }
```

Wanneer je de `Sort`-methode toepast op een array van strings dan zullen de elementen alfabetisch gerangschikt worden. Uiteraard werkt dit ook op arrays van andere datatypes. Zolang C# maar weet hoe dit type gesorteerd moet worden, zal dit werken. Het zal getallen van klein naar groot sorteren, tekst volgens de regels van het alfabet, enums volgens hun interne voorstelling, enz.

8.4.2 Reverse: Arrays omkeren

Met de `Array.Reverse()`-methode kunnen we dan weer de volgorde van de elementen van de array omkeren (dus het laatste element vooraan zetten en zo verder):

```
1 Array.Reverse(myColors);
```

8.4.3 Clear: Arrays leegmaken

Een array volledig leegmaken waarbij alle elementen op hun standaard waarde zetten (bv. 0 bij `int`, enz.) doe je met de `Array.Clear()`-methode, als volgt:

```
1 Array.Clear(myColors, 0, myColors.Length);
```

Hierbij geeft de tweede parameter aan vanaf welke index moet leeggemaakt worden, en de derde hoeveel elementen vanaf die index.

8.4.4 Copy : array by value kopiëren

De .Copy() behelst iets meer werk, daar deze methode:

- een reeds aangemaakte, nieuwe array nodig heeft, waar naar gekopieerd moet worden.
- moet meekrijgen hoe lang de bronarray (*source*) is, of hoeveel elementen uit de bronarray moeten gekopieerd worden.

Volgend voorbeeld toont hoe we alle elementen uit *myColors* kunnen kopiëren naar een nieuwe array *copyColors*. De eerste parameter is de bron-array, dan de doel-array en final het aantal elementen dat moet gekopieerd worden:

```
1 string[] myColors = { "red", "green", "yellow", "orange", "blue" };  
2 string[] copyColors = new string[myColors.Length];  
3 Array.Copy(myColors, copyColors, myColors.Length);
```

Willen we enkel de eerste twee elementen kopiëren dan zou dat er als volgt uitzien:

```
1 Array.Copy(myColors, copyColors, 2);
```



Bekijk zeker ook de overloaded versies die de .Copy() methode heeft. Zo kan je ook een bepaald stuk van een array kopiëren en ook bepalen waar in de doel-array dit stuk moet komen.

8.4.5 BinarySearch: Zoeken in arrays

De `BinarySearch`-methode maakt het mogelijk om te zoeken naar de index van een gegeven element in een array.



De `BinarySearch`-methode werkt enkel indien de elementen in de array gesorteerd staan!

Je geeft aan de methode 2 parameters mee: enerzijds de array in kwestie en anderzijds het element dat we zoeken. Als resultaat wordt de index van het gevonden element teruggegeven. Indien niets wordt gevonden zal het resultaat **negatief** zijn.

Volgende code zal bijvoorbeeld de index teruggeven van de kleur “red” indien deze in de array `myColors` staat:

```
1 int indexRed = Array.BinarySearch(myColors, "red");
```

Volgend voorbeeld toont het gebruik van deze methode:

```
1 int[] metingen = {224, 34, 156, 1023, -6};
2 Array.Sort(metingen); //anders zal BinarySearch niet werken
3
4 Console.WriteLine("Welke meting zoekt u?");
5 int keuze = int.Parse(Console.ReadLine());
6
7 int index = Array.BinarySearch(metingen, keuze);
8 if(index >= 0)
9     Console.WriteLine($"{keuze} gevonden op {index}");
10 else
11     Console.WriteLine("Niet gevonden");
```

8.5 Algoritmes en arrays

Omdat arrays ongelooflijk groot kunnen worden, is het nuttig dat je algoritmes kunt schrijven die vlot met arrays kunnen werken. Je wilt niet dat je programma er 3 minuten over doet om gewoon te ontdekken of een bepaalde waarde in een array voorkomt of niet². We zullen nu 2 typische algoritmen bespreken die vaak voorkomen als je met loops en arrays aan de slag gaat gaan.



Twee maar!? Er zijn tal van andere algoritmes. Denk maar aan de verschillende manieren om arrays te sorteren (bijvoorbeeld de fameuze bubblesort en quicksort algoritmes). Al deze algoritmes hier bespreken zou een boek apart vereisen. Ik toon er daarom enkele ter illustratie.

8.5.0.1 Manueel zoeken in arrays

Het nadeel van BinarySearch is dat deze vereist dat je array-elementen gesorteerd staan. Uiteraard is dit niet altijd gewenst. Stel je voor dat je een simulatie maakt voor een fietswedstrijd en wilt weten of een bepaalde wielrenner in de top 5 staat.

Het zoeken in arrays kan met behulp van loops tamelijk snel. Volgende applicatie gaat zoeken of het getal 12 aanwezig is in de array (de wielrenners werken met rugnummers). Indien ja dan wordt de index bewaard van de positie in de array waar het getal staat:

```

1 int teZoekenGetal = 12;
2 int[] top5 = { 5, 10, 12, 25, 16 };
3 bool gevonden = false;
4 int index = 0;
5 do
6 {
7     if (top5[index] == teZoekenGetal)
8     {
9         gevonden = true;
10    }
11    index++;
12 } while ( !gevonden && index < top5.Length );
13
14 if (gevonden)
15 {
16     Console.WriteLine($"Nr. {teZoekenGetal} op plek {index}");
17 }
18 }
```

²Bij jobs sollicitaties voor programmeurs word je soms gevraagd om dergelijke algoritmes zonder hulp uit te schrijven.

8.5.0.2 Manueel zoeken met while

Ik toon nu een voorbeeld van hoe je kan zoeken in een array wanneer we bijvoorbeeld 2 arrays hebben die 'synchroon' zijn. Daarmee bedoel ik: de eerste array bevat bijvoorbeeld producten, de tweede array bevat de prijs van ieder product. De prijs van de producten staat steeds op dezelfde index in de andere array (de prijs van peren is dus 6.2, meloenen 2.9, enz.):

```
1 string[] producten = {"appelen", "peren", "meloenen"};
2 double[] prijzen = {3.3, 6.2, 2.9};
```

We vragen nu aan de gebruiker van welk product de prijs getoond moet worden:

```
1 Console.WriteLine("Welke productprijs wenst u?");
2 string keuzeGebruiker = Console.ReadLine();
```

Ik toon vervolgens hoe je met **while** eerst het juiste product zoekt en dan vervolgens die index bewaart en gebruikt om de prijs te tonen:

```
1 bool gevonden = false;
2 int productIndex = -1;
3 int teller = 0;
4 while (teller < producten.Length && keuzeGebruiker != producten[
    teller])
5 {
6     teller++;
7 }
8 if (teller != producten.Length) //product gevonden!
9 {
10     gevonden = true;
11     productIndex = teller;
12 }
13 if (gevonden)
14 {
15     Console.Write($"Prijs van {keuzeGebruiker}");
16     Console.WriteLine($"is {prijzen[productIndex]}.");
17 }
18 else
19 {
20     Console.WriteLine("Niet gevonden");
21 }
```

8.6 String en arrays

Het type **string** is niet meer dan een arrays van karakters, **char** []. Het is dan ook logisch dat we dit erg belangrijke datatype even apart toelichten en enkele nuttige methoden tonen om strings te manipuleren.

8.6.1 String naar char array

Om de aparte karakters van een **string** te bewerken zet je deze best om naar een *char-array*. Dit kan gebruikt worden van `.ToCharArray()` als volgt:

```
1 string origineleZin = "Ik ben Tom";
2 char[] karakters = origineleZin.ToCharArray();
3 karakters[8] = 'i';
```

De array zal nu het volgende bevatten: `Ik ben Tim`. Willen je dit nu terug als **string**, dan lees je snel verder!

8.6.2 Char array naar string

Ook de omgekeerde weg is mogelijk. De werking is iets anders en maakt gebruik van `new string()`. Let vooral op hoe we de char array doorgeven als argument bij het aanmaken van een nieuwe **string** in lijn 3:

```
1 char[] alleKarakters = {'h', 'a', 'l', 'l', 'o'};
2 alleKarakters[2] = 'x';
3 string woord = new string(alleKarakters);
4 Console.WriteLine(woord);
```

De uitvoer van deze code zal zijn: `haxlo`.

8.6.3 Andere nuttige methoden met strings

Volgende methoden kan je rechtstreeks op string-variabelen oproepen:

8.6.3.1 Length

Geeft het totaal aantal karakters in de string wat logisch is, daar het om een array gaat:

```
1 string myName = "Tim";
2 Console.WriteLine(myName.Length); //er verschijnt 3 op het scherm
```

8.6.3.2 IndexOf

Deze methode geeft een **int** terug die de index bevat waar de string die je als parameter meegaf begint. Je kan deze index gebruiken om te ontdekken of een bepaald woord bijvoorbeeld in een grote lap tekst voorkomt zoals volgend voorbeeld toont:

```
1 string boek = "Ik ben Reinhardt";
2 int index = boek.IndexOf("ben");
3 Console.WriteLine(index);
```

Er zal 3 verschijnen op scherm. De substring “ben” start op positie 3. “ik” staat op positie 0 en 1, gevolgd door een spatie op positie 2. Indien de string niet gevonden werd, zal **index** de waarde -1 krijgen.

8.6.3.3 Trim

Trim() verwijdert alle onnodige spaties en andere onzichtbare tekens vooraan en achteraan de string. Deze methode geeft de opgekuiste string terug als resultaat. Dit resultaat moet je dus bewaren als je er nog iets mee wilt doen. In het volgende voorbeeld overschrijven we de originele string met z’n opgekuiste versie:

```
1 string boek = " Ik ben Reinhardt ";
2 Console.WriteLine(boek);
3 boek = boek.Trim();
4 Console.WriteLine(boek);
```

Dit zal de output op het scherm zijn (de spaties achteraan op lijn 1 zie je niet, maar zijn er dus wel):

```
1 Ik ben Reinhardt
2 Ik ben Reinhardt
```

8.6.3.4 ToUpper en ToLower

ToUpper zal de meegegeven string naar ALLCAPS omzetten en geeft de nieuwe string als resultaat terug. **ToLower ()** doet het omgekeerde.

```
1 string boek = "Ik ben Reinhardt";
2 Console.WriteLine(boek.ToUpper());
3 Console.WriteLine(boek.ToLower());
```

Output op het scherm:

```
1 IK BEN REINHARDT
2 ik ben reinhardt
```

8.6.3.5 Replace

Replace(**string** old, **string** news) zal in de string alle substrings die gelijk zijn aan old vervangen door de meegegeven news string. Vervolgens zal de nieuwe string als resultaat worden teruggegeven.

Volgende voorbeeld toont dit en zal “Mercy” vervangen door “Reinhardt”:

```
1 string boek = "Ik ben Mercy";
2 boek = boek.Replace("Mercy", "Reinhardt");
3 Console.WriteLine(boek);
```



Replace kan je ook misbruiken om bijvoorbeeld alle woorden uit een stuk tekst te verwijderen door deze te vervangen door een lege **string** met de waarde "".

Volgende code zal alle "e"'s uit de tekst verwijderen:

```
1 string boek = "Ik ben Mercy";
2 boek = boek.Replace("e", "");
3 Console.WriteLine(boek);
```

Waardoor we Ik bn Mrcy op het scherm krijgen.

8.6.3.6 Remove

Remove(**int** start, **int** lengte) zal op de index start alle lengte volgende karakters in de **string** verwijderen en een nieuwe, kortere **string** als resultaat geven.

Volgend voorbeeld zal het stukje “ben” uit de **string** weghalen:

```
1 string boek = "Ik ben Mercy";
2 boek = boek.Remove(3,4);
3 Console.WriteLine(boek);
```

Output op het scherm:

```
1 Ik Mercy
```

In voorgaande voorbeeld gaven we de methode Remove de opdracht: “verwijder alles vanaf het element met index 3 (de b) en dit gedurende 4 tekens (dus tot en mét de spatie na ben)”.

8.6.3.7 Split



Volgende twee methoden zijn **static** en moet je via de klasse **String** doen en niet via de objecten zelf. Ik leg in hoofdstuk 11 uit waarom dat is.

De **Split** methode laat toe een string te splitsen op een bepaald teken. Het resultaat is steeds een **array van strings**.

```
1 string data = "12,13,20";
2 string[] gesplitst = data.Split(',');
3
4 for(int i = 0; i<gesplitst.Length;i++)
5 {
6     Console.WriteLine(gesplitst[i]);
7 }
```

Uiteraard kan je dit gebruiken om op eender welk **char** te splitsen.

8.6.3.8 Join

Via **Join** kunnen we een array van strings terug samenvoegen. Het resultaat is een nieuwe string.

Volgende voorbeeld zal de eerder gesplitste array van het vorige voorbeeld opnieuw samenvoegen maar nu met telkens een ; tussen iedere string:

```
1 string joined = String.Join(";", gesplitst);
```

8.7 Methoden en arrays

Zoals alle datatypes kan je ook arrays van eender welk datatype als parameter gebruiken bij het schrijven van een methode. **Lees nu volgende waarschuwing extra aandachtig, a.u.b:**“



Herinner je dat arrays *by reference* werken. Je werkt dus steeds met de origineel meegegeven array (of beter, de referentie er naar), ook in de methode. Als je dus aanpassingen aan de array aanbrengt in de methode, dan zal dit ook gevolgen hebben op de array van waaruit we de methode aanriepen.

Stel dat je bijvoorbeeld een methode hebt die als parameter 1 array van ints meekrijgt. De methode zou er dan als volgt uitzien:

```
1 static void LeesData(int[] inArray)
2 {
3
4 }
```

Om deze methode aan te roepen volstaat het om een bestaande array als parameter mee te geven:

```
1 int[] getallen = {1, 2, 3};
2 LeesData(getallen);
```

8.7.1 Array grootte in de methode

Een array als parameter meegeven kan dus, maar een ander aspect waar rekening mee gehouden moet worden is dat je niet kan ingeven in de parameterlijst hoe groot de array is. Je zal dus in je methode steeds de grootte van de array moeten uitlezen met de `.Length`-eigenschap.

Volgende methodesignatuur is dus **FOUT!**

```
1 static void LeesData(int[6] inArray)
2 {
3
4 }
```

En zal volgende foutbericht genereren:

CS0270 Array size cannot be specified in a variable declaration (try initializing with a 'new' expression)

Figuur 8.6: Duidelijk toch!

8.7.2 Arraymethode voorbeeld

Volgend voorbeeld toont een methode die alle getallen van de meegegeven array op het scherm zal tonen:

```
1 static void ToonArray(int[] getalArray)
2 {
3     Console.WriteLine("Array output:");
4     for (int i = 0; i < getalArray.Length; i++)
5     {
6         Console.WriteLine(getalArray[i]);
7     }
8 }
```

De ToonArray methode aanroepen kan dan als volgt:

```
1 int[] leeftijden = {2, 5, 1, 6};
2 ToonArray(leeftijden);
```

En de output zal dan zijn:

```
1 Array output:
2 2
3 5
4 1
5 6
```

8.7.3 Voorbeeldprogramma met methoden

Volgend programma toont hoe we verschillende onderdelen van de code in methoden hebben geplaatst zodat:

1. de lezer van de code sneller kan zien wat het programma juist doet.
2. code herbruikbaar is.

Begrijp je wat dit programma doet? En kan je voorspellen wat er op het scherm zal komen?

```
1 static void VulArray(int[] getalArray)
2 {
3     for (int i = 0; i < getalArray.Length; i++)
4     {
5         getalArray[i] = i;
6     }
7 }
8
9 static void VermenigvuldigArray(int[] getalArray, int factor)
10 {
11     for (int i = 0; i < getalArray.Length; i++)
12     {
13         getalArray[i] = getalArray[i] * factor;
14     }
15 }
16
17 static void ToonVeelvouden(int[] getalArray, int veelvoudenVan)
18 {
19     for (int i = 0; i < getalArray.Length; i++)
20     {
21         if (getalArray[i] % veelvoudenVan == 0)
22             Console.WriteLine(getalArray[i]);
23     }
24 }
25
26 static void Main(string[] args)
27 {
28     int[] getallen = new int[100];
29     VulArray(getallen);
30     VermenigvuldigArray(getallen, 3);
31     ToonVeelvouden(getallen, 4);
32 }
```

8.7.4 Array als return-type bij een methode

Een array kan ook gebruikt worden als het returntype van een methode. Hiervoor zet je gewoon het type array als returntype in de methodesignatuur. Ook hier mag geen grootte aangeven.

Stel bijvoorbeeld dat je een methode hebt die een int-array aanmaakt van een gegeven grootte waarbij ieder element van de array reeds een beginwaarde heeft die je ook als parameter meegeeft:

```
1 static int[] MaakArray(int lengte, int beginwaarde)
2 {
3     int[] resultArray = new int[lengte];
4     for (int i = 0; i < lengte; i++)
5     {
6         resultArray[i] = beginwaarde;
7     }
8     return resultArray;
9 }
```

De aanroep van deze methode vereist dan dat je het resultaat opvangt in een nieuwe variabele, als volgt:

```
1 int[] mijnNieuweArray = MaakArray(4,666);
```



Snel, zet je helm op, voor er ongelukken gebeuren! Ik had al enkele keren gezegd dat arrays by reference worden meegegeven, maar wat is daar nu het gevolg van? Wel, laten we eens naar volgende programmaatje kijken dat ik heb geschreven om de nummering van de appartementen in een flatgebouw aan te passen.

Zoals je weet is het gelijkvloers in sommige landen 0, terwijl in andere dit 1 is. Volgende programma past het nummer van het gelijkvloers aan:

```
1 static void PasAan(int[] inarr)
2 {
3     inarr[0] = 0;
4 }
5
6 public static void Main()
7 {
8     int[] verdiepnummers = {1,2,3};
9     Console.WriteLine($"VOOR:{verdiepnummers[0]}"); // VOOR:1
10    PasAan(verdiepnummers);
11    Console.WriteLine($"NA:{verdiepnummers[0]}"); // NA:0
12 }
```

Dankzij het feit dat we aan PasAan een array meegeven *by reference* zal de methode werken op de originele array en is deze code dus mogelijk.

Vergelijk dit met volgende voorbeeld waar we een **int** als parameter meegeven die *by value* en niet *by reference* wordt meegegeven:

```
1 static void PasAan(int inArray)
2 {
3     inArray = 0; //inArray wordt 0
4 }
5
6 public static void Main()
7 {
8     int[] getallen = {1,2,3};
9     PasAan(getallen[0]);
10    Console.WriteLine(getallen[0]); // NA:1
11 }
```

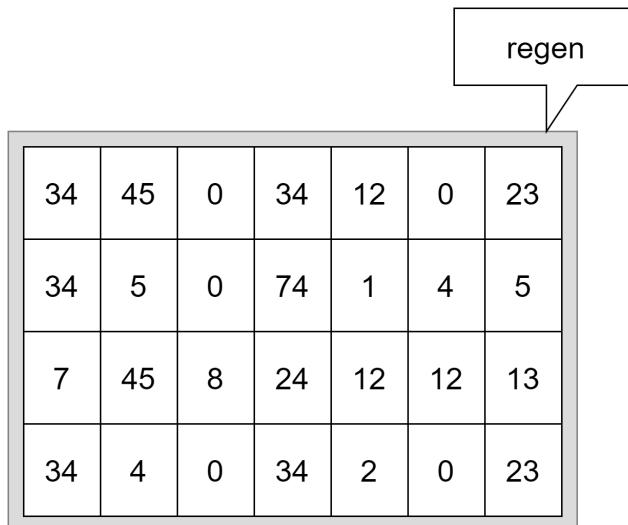
Daar de methode nu werkt met een kopie, zal de aanpassing in de methode dus geen invloed hebben op de origineel meegegeven **int** (ongeacht dat die deel uitmaakt van een array).

8.8 Meer-dimensionale arrays

Voorlopig hebben we enkel met zogenaamde 1-dimensionale arrays gewerkt. Je kan echter ook meerdimensionale arrays maken. Denk maar aan een n-bij-m array om een matrix voor te stellen. Ik bespreek meerdimensionale arrays maar kort om de eenvoudige reden dat je die in de praktijk minder vaak zal nodig hebben.

Stel je het voorbeeld aan het begin van dit hoofdstuk voor, waarin we de regenval gedurende 7 dagen wilden meten. Wat als we dit gedurende 4 weken wensen te doen, maar wel niet alle data in één lange array willen plaatsen? We zouden dan een 2-dimensionale array kunnen maken als volgt:

```
1 int[,] regen =
2 {
3     {34,45,0,34,12,0,23},
4     {34,5,0,74,1,4,5},
5     {7,45,8,24,12,12,13},
6     {34,4,0,34,2,0,23}
7 }
```



Figuur 8.7: Een tweedimensionale array.

De arrays die we nu behandelen zullen steeds “rechthoekig” zijn. Daarmee bedoelen we dat ze steeds per rij of kolom evenveel elementen zullen bevatten als in de andere rijen of kolommen.



Arrays die per rij of kolom een andere hoeveelheid elementen hebben zijn zogenaamde **jagged arrays**, welke we verderop kort zullen bespreken.

8.8.1 n-dimensionale arrays aanmaken

Door een komma tussen rechte haakjes te plaatsen tijdens de declaratie van een array, kunnen we meer-dimensionale arrays maken.

Bijvoorbeeld om een 2D array te maken schrijven we:

```
1 string[,] boeken;
```

Een 3D-array:

```
1 short[,,,] temperaturen;
```

(enz.)



Ja, dit kan dus ook een 10-dimensionale array aanmaken. Kan handig zijn als je een fysicus bent die rond de supersnaartheorie onderzoekt doet.

```
1 int[,,,,,,,,,,] jeBentGek;
```

Ja, 11 kan ook als je meer in de M-theorie gelooft. En zelfs 26 moet de bosonische snaartheorie meer je ding zijn:

```
1 int[,,,,,,,,,,,,,,,,,,] jeBentNogGekker;
```

8.8.2 Initialisatie

Ook om nu effectief een array aan te maken gebruiken we de komma-notatie, alleen moeten we nu ook de effectieve groottes aangeven. Voor een 5 bij 10 array bijvoorbeeld schrijven we (merk op dat dit dus een 2D-array is):

```
1 int[,] matrix = new int[5,10];
```

Om een array ook onmiddellijk te initialiseren met waarden gebruiken we de volgende uitdrukking :

```
1 string[,] boeken =
2 {
3     {"Macbeth", "Shakespeare", "ID12341"}, 
4     {"Before I Get Old", "Dave Marsh", "ID234234"}, 
5     {"Security+", "Mike Pastore", "ID3422134"}, 
6     {"Zie scherp", "Tim Dams", "ID007"} 
7 };
```

Merk op dat we dus nu een 3 bij 4 array maken maar dat dit dus nog steeds een 2D-array is. Iedere rij bestaat uit 3 elementen. **We maken letterlijk een array van arrays.**

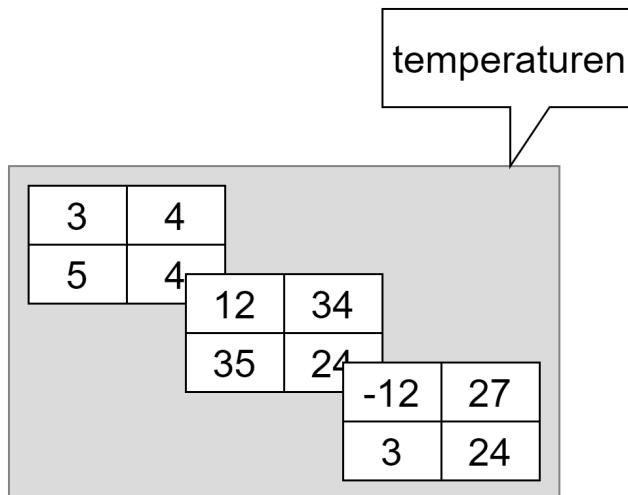
Of bij een 3D-array:

```

1 int[,,] temperaturen =
2 {
3     {
4         {3,4}, {5,4}
5     },
6     {
7         {12,34}, {35,24}
8     },
9     {
10        {-12,27}, {3,24}
11    },
12 };

```

Die we als volgt kunnen visualiseren:



Figuur 8.8: De derde dimensie bestaat uit drie 2-dimensionale 2 bij 2 arrays...



Zoals je ziet worden meerdimensionale arrays snel een kluwen van komma's, accolades en haakjes. Probeer dus je dimensies te beperken. Je zal zelden een 3 -of meer dimensionale array nodig hebben.

De regel is eenvoudig: als je een 7-dimensionale array nodig hebt, is de kans groot dat je een volledig verkeerd algoritme hebt verzonden ... of dat je nog niet aan hoofdstuk 9 bent geraakt ... of dat je een topwetenschapper in CERN bent. *Choose your reason!*

Stel dat we uit de boeken-array de auteur van het derde boek wensen te tonen dan kunnen we schrijven:

```
1 Console.WriteLine(boeken[2, 1]);
```

Dit zal Mike Pastore op het scherm zetten.

En bij de temperaturen:

```
1 Console.WriteLine(temperaturen[2, 0, 1]);
```

Dit zal 27 teruggeven. We vragen van de laatste array ([2]), daarbinnenin de eerste array (rij [0]) en daarvan het tweede element(kolom [1]).

8.8.3 Lengte van iedere dimensie in een n-dimensionale matrix

Indien je de lengte opvraagt van een meer-dimensionale array dan krijg je de som van iedere lengte van iedere dimensie. Dit is logisch: in het geheugen van een computer worden arrays altijd als 1 dimensionale arrays voorgesteld. De boeken array zal lengte 12 hebben (3×4) en temperaturen toevallig ook ($3 \times 2 \times 2$).

Je kan echter de lengte van iedere aparte dimensie te weten komen met de `.GetLength()` methode die iedere array heeft. Als parameter geef je de dimensie mee waarvan je de lengte wenst:

```
1 int arrayRijen = boeken.GetLength(0); //geeft 4
2 int arrayKolommen = boeken.GetLength(1); //geeft 3
```

Het aantal dimensies van een array wordt trouwens weergegeven door de `.Rank` eigenschap die ook iedere array heeft. Bijvoorbeeld:

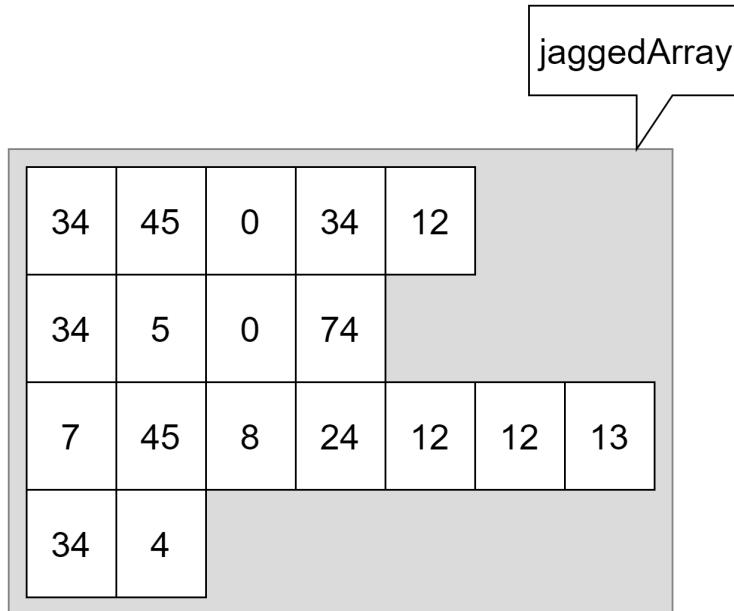
```
1 Console.WriteLine(boeken.Rank); //geeft 2
2 Console.WriteLine(temperaturen.Rank); //geeft 3
```

Willen we dus de lengte van iedere dimensie van bijvoorbeeld de temperaturen array op het scherm krijgen dan kan dat als volgt:

```
1 for (int i = 0; i < temperaturen.Rank; i++)
2 {
3     Console.WriteLine(temperaturen.GetLength(i));
4 }
```

8.8.4 Jagged arrays

Jagged arrays (letterlijk *gekartelde arrays*) zijn *arrays van arrays* maar van verschillende lengte. De arrays die we totnogtoe zagen moesten steeds rechthoekig zijn. Jagged arrays, zoals de naam doet vermoeden, hoeven dat niet te zijn:



Figuur 8.9: Voorbeeld van een jagged array.

Ik ga niet al te diep in deze arrays ingaan omdat deze, alhoewel erg nuttig, vaak omslachtige, meer foutgevoelige code zullen creëren. Meestal zijn er gezondere alternatieven te gebruiken zoals de verschillende collectie-klassen uit hoofdstuk 12.

8.8.4.1 Jagged arrays aanmaken

Het grote verschil bij het aanmaken van bijvoorbeeld een 2D jagged array is het gebruik van de vierkante haken (en dus niet bijvoorbeeld `tickets[,]`):

```

1 double[][] tickets=
2 {
3     new double[] {3.0, 40, 24},
4     new double[] {123, 31.3 },
5     new double[] {2.1}
6 };

```

8.8.4.2 Indexering bij jagged arrays

De indexering blijft dezelfde, maar ook hier dus niet met komma's, maar met vierkante haken (bijvoorbeeld `tickets[0][1]`).

Uiteraard moet je er wel rekening mee houden dat niet eerder welke index binnen een bepaalde sub-array zal werken, het is dan ook aangeraden om zeker de `Length`-methode te gebruiken om de sub-arrays op hun lengte te bevragen. Wanneer je `.Length` vraagt van de `tickets` array dan zal je 3 als antwoord krijgen, daar deze 2D array uit 3 sub-arrays bestaat.

Wil je vervolgens de lengte kennen van de middelste sub-array (met dus index 1) dan gebruik je `tickets[1].Length`.

9 Object georiënteerd programmeren

Tot nu toe heb je vooral geleerd om *gestructureerd* te programmeren - soms ook wel *procedureel* programmeren genoemd - een programmeerconcept uit de jaren zestig. Hierbij schrijf je code gebruik makend van methoden, loops en beslissingsstructuren.

Hoewel gestructureerd programmeren erg nuttig blijft, wordt het bij complexere applicaties vaak minder intuïtief en soms nodeloos complex. Een middelgroot project verzandt al gauw in moeilijk te onderhouden spaghetti code.

Er is echter een alternatief: **object georiënteerd Programmeren (OOP, Object Oriented Programming)**. OOP bouwt voort op gestructureerd programmeren, maar maakt het mogelijk om veel krachtigere applicaties te ontwikkelen.

Bij OOP draait alles rond **klassen en objecten** die intern nog steeds gestructureerde code bevatten. Alle bekende concepten zoals loops, methoden en beslissingsstructuren blijven bestaan. We gaan ze echter verpakken in handige, kleinere aparte klassen. Met OOP wordt onze code **modulair, leesbaarder en onderhoudbaar**. Bovendien wordt de code krachtiger en kunnen we complexere problemen eenvoudiger oplossen.



Ik zet “oplossen” tussen aanhalingstekens. Net zoals alles binnen dit domein ben jij als programmeur uiteindelijk degene die het boeltje moet oplossen. Code, programmeerparadigma’s en bibliotheken zijn niet meer dan nuttig gereedschap in jouw arsenaal van programmeertools. Als jij beslist om een hamer als zaag te gebruiken... Tja, dan houd ik m’n hart vast voor het resultaat.

Dit geldt ook voor de technieken die je nog in dit boek gaat leren: ze zijn “een tool”, niets meer. Jij zal ze nog steeds zo optimaal mogelijk moeten leren gebruiken. Uiteraard is het doel van dit boek je zo duidelijk mogelijk het verschil én de bruikbaarheid van de verschillende nieuwe technieken aan te leren.

9.1 C# is OO in hart en nieren

Toen C# werd ontwikkeld in 2001 was één van de hoofddoelen van de programmeertaal om “een eenvoudige, moderne, objectgeoriënteerde programmeertaal voor algemene doeleinden” te worden. **C# is van de grond af opgebouwd met het OOP paradigma als primaire drijfveer.** Een paradigma is een algemeen geaccepteerde manier van denken en doen binnen een bepaald vakgebied, in dit geval binnen de programmeerwereld.

Wanneer we nieuwe programma's in C# ontwikkelden dan zagen we hier reeds bewijzen van. Zo zagen we steeds het keyword **class** bovenaan staan, telkens we een nieuw project aanmaakten:

```
1  namespace WorldDominationTool  
2  {  
3      internal class Program  
4  {
```

De klasse Program zorgt ervoor dat ons programma voldoet aan de C# afspraken die zeggen dat alle C# code in klassen moet staan.



Duizend mammoeten en sabeltandtijgers! Ik dacht dat ik nu wel mee zou zijn met alles wat C# me zou voorschotelen. Helaas, wolharige neushoorn-kaas, niet dus. Ik ga een voorspelling doen: van alle hoofdstukken in dit boek, wordt dit hoofdstuk hetgene waar je het meest je tanden op gaat stuk bijten. Hou dus vol, geef niet te snel op en kom geregeld hier terug. Succes gewenst!

9.1.1 Een wereld zonder OOP: Pong

Om de kracht van OOP te demonstreren gaan we een applicatie van lang geleden (deels) herschrijven gebruik makende van de kennis van de vorige 8 hoofdstukken. We gaan de arcadehal klassieker "Pong" deels namaken, waarbij we als doel hebben om een balletje alvast op het scherm te laten botsen. Een rudimentaire oplossing zou de volgende kunnen zijn:

```
1 Console.CursorVisible = false;
2 int balX = 20;
3 int balY = 20;
4 int vX = 2;
5 int vY = 1;
6 while (true)
7 {
8     //vX van richting veranderen aan de randen
9     if (balX + vX >= Console.WindowWidth || balX+vX < 0)
10    {
11        vX = -vX;
12    }
13    balX = balX + vX; //X positie updaten
14    //vY van richting veranderen aan de randen
15    if (balY + vY >= Console.WindowHeight || balY+vY < 0)
16    {
17        vY = -vY;
18    }
19    balY = balY + vY; //Y positie updaten
20    //Output naar scherm sturen
21    Console.SetCursorPosition(balX, balY);
22    Console.Write("0");
23    System.Threading.Thread.Sleep(50); //50 ms wachten
24    Console.Clear();
25 }
```

Hopelijk begrijp je deze code. Test ze maar eens in een programma. Zoals je zal zien krijgen we een balletje ("0") dat over het scherm vliegt en telkens van richting verandert wanneer het aan de randen van het applicatievenster komt. De belangrijkste informatie zit in de variabelen `balX`, `balY` die de huidige positie van het balletje bevatten. Voorts zijn ook `vX` en `vY` belangrijk: hierin houden we bij in welke richting (en met welke snelheid) het balletje beweegt (een zogenaamde bewegingsvector).

9.1.1.1 Extra balletjes?

Dit soort applicatie in C# schrijven met behulp van gestructureerde programmeer-concepten is redelijk eenvoudig. Maar wat als we nu 2 balletjes nodig hebben? Laten we arrays even links laten liggen en het gewoon eens naïef oplossen. Al na enkele lijnen kopiëren merken we dat onze code ongelooflijk rommelachtig gaat worden en we bijna iedere lijn moeten dupliveren:

```
1 Console.CursorVisible = false;
2 int balX = 20;
3 int balY = 20;
4 int vX = 2;
5 int vY = 1;
6
7 int bal2X = 10;
8 int bal2Y = 8;
9 int v2X = 2;
10 int v2Y = -1;
11
12 while (true)
13 {
14     if (balX + vX >= Console.WindowWidth || balX + vX < 0)
15     {
16         vX = -vX;
17     }
18     if (bal2X + v2X >= Console.WindowWidth || bal2X + v2X < 0)
19     {
20         v2X = -v2X;
21     }
22
23     balX = balX + vX;
24     bal2X = bal2X + v2X;
25     //enzovoort
```

Bijna iedere lijn code moeten we verdubbelen. Arrays zouden dit probleem deels kunnen oplossen, maar we krijgen dan in de plaats de complexiteit van werken met arrays op ons bord. Dat is voor 2 balletjes misschien wat overdreven. Het zal op de koop toe onze terug minder leesbaar maakt.

9.1.2 Een wereld met OOP: Pong

Uiteraard zijn we nu eventjes gestructureerd programmeren aan het demoniseren, dit is echter een bekend 21e eeuws trucje om je punt te maken.

Wanneer we Pong met het OOP paradigma willen aanpakken dan is het de bedoeling dat we werken met klassen en objecten.

Net zoals aan de start van dit boek ga ik je ook nu even in het diepe gedeelte van het bad gooien. Wees niet bang, ik zal je er tijdig uithalen! Je zal versteld staan hoeveel code je eigenlijk zult herkennen.

Om Pong in OOP te maken hebben we eerst een klasse nodig waarin we ons balletje gaan beschrijven, zonder dat we al een balletje hebben. En dat ziet er zo uit:

```
1 internal class Balletje
2 {
3     //Eigenschappen
4     public int X { get; set; }
5     public int Y { get; set; }
6     public int VX { get; set; }
7     public int VY { get; set; }
8
9     //Methoden
10    public void Update()
11    {
12        if (X + VX >= Console.WindowWidth || X + VX < 0)
13        {
14            VX = -VX;
15        }
16        X = X + VX;
17
18        if (Y + VY >= Console.WindowHeight || Y + VY < 0)
19        {
20            VY = -VY;
21        }
22        Y = Y + VY;
23    }
24
25    public void TekenOpScherm()
26    {
27        Console.SetCursorPosition(X, Y);
28        Console.Write("O");
29    }
30}
31 }
```



De code voor een nieuwe klasse schrijf je best in een apart bestand in je project. Klik bovenaan in de menu balk op “Project” en kies dan “Add class...”. Geef het bestand de naam “Balletje.cs”.

Bijna alle code van zonet hebben we hier geïntegreerd in een **class** `Balletje`, maar er zit duidelijk een nieuw sausje over. Vooral aan het begin zien we onze 4 variabelen terugkomen in een nieuw kleedje: namelijk als eigenschappen oftewel *properties* (herkenbaar aan de **get** en **set** keywords).

Maar al bij al lijkt de code grotendeels op wat we al kenden. En dat is goed nieuws. OOP gooit de vorige hoofdstukken niet in de vuilbak, het gaat als het ware een extra laag over het geheel leggen. Let ook op het essentiële woordje **class** bovenaan, daar draait alles natuurlijk om: **klassen en objecten**.



Een klasse is een blauwdruk van een bepaalde soort ‘dingen’ of objecten. Objecten zijn de “echte” dingen die werken volgens de beschrijving van de klasse. Ja ik heb zonet 2x hetzelfde verteld, maar het is essentieel dat je het verschil tussen de termen **klasse** en **object** goed begrijpt.

Laten we eens een **balletje-object** in het leven roepen. In de main schrijven we daarom dit:

```
1 Console.CursorVisible = false;
2 Balletje bal1 = new Balletje();
3 bal1.X = 20;
4 bal1.Y = 20;
5 bal1.VX = 2;
6 bal1.VY = 1;
```

Ok, interessant. Die **new** heb je al gezien wanneer je met Random ging werken en de code erna is ook nog begrijpbaar: we stellen eigenschappen van het nieuwe `bal1` object in. En nu komt het! Kijk hoe eenvoudig onze volledig `main` nu is geworden:

```

1 static void Main(string[] args)
2 {
3     Console.CursorVisible = false;
4     Balletje bal1 = new Balletje();
5     bal1.X = 20;
6     bal1.Y = 20;
7     bal1.VX = 2;
8     bal1.VY = 1;
9
10    while (true)
11    {
12        bal1.Update();
13        bal1.TekenOpScherm();
14
15        System.Threading.Thread.Sleep(50);
16        Console.Clear();
17    }
18 }
```

De loopcode is herleid tot 2 aanroepen van **methoden op het bal1 object**: .Update() en .TekenOpScherm.

Run deze code maar eens. Inderdaad, deze code doet exact hetzelfde als hiervoor. Ook nu krijgen we 1 balletje dat op het scherm over en weer botst.

En nu - abracadabra - kijk goed hoe eenvoudig onze code blijft als we 2 balletjes nodig hebben:

```

1 Console.CursorVisible = false;
2 Balletje bal1 = new Balletje();
3 bal1.X = 20;
4 bal1.Y = 20;
5 bal1.VX = 2;
6 bal1.VY = 1;
7
8 Balletje bal2 = new Balletje();
9 bal2.X = 10;
10 bal2.Y = 8;
11 bal2.VX = 2;
12 bal2.VY = -1;
13
14 while (true)
15 {
16     bal1.Update();
17     bal2.Update(); //zo simpel!
18     bal1.TekenOpScherm();
19     bal2.TekenOpScherm(); //wow, zooo simpel :)
20     System.Threading.Thread.Sleep(50);
21     Console.Clear();
22 }
```

Dit is de volledige code om 2 balletjes te hebben. Hoe mooi is dat?!

De kracht van OOP zit hem in het feit dat we de logica IN DE OBJECTEN ZELF plaatsen. De

objecten zijn met andere woorden verantwoordelijk om hun eigen gedrag uit te voeren gebaseerd op externe impulsen en hun eigen interne toestand. In onze main zeggen we aan beide balletjes “update je zelf eens”, gevuld door “teken je zelf eens”.

Wanneer we 3 of meer balletjes zouden nodig hebben dan zullen we best arrays in de mix moeten gooien. Onze code blijft echter v  l eenvoudiger  n krachtiger dan wanneer we in het voorgaande enkel de kennis gebruikten die we tot nog toe hadden. Omdat we toch al in het diepe eind zitten, zal ik hier toch al eens tonen hoe we 100 balletjes op het scherm kunnen laten botsen (we gaan Random gebruiken zodat er wat willekeurigheid in de balletjes zit):

```

1 const int AANTAL_BALLETJES = 100;
2 Random r = new Random();
3 Balletje[] veelBalletjes = new Balletje[AANTAL_BALLETJES];
4 for (int i = 0; i < veelBalletjes.Length; i++) //balletjes aanmaken
5 {
6     veelBalletjes[i] = new Balletje();
7     veelBalletjes[i].X = r.Next(10, 20);
8     veelBalletjes[i].Y = r.Next(10, 20);
9     veelBalletjes[i].VX = r.Next(-2, 3);
10    veelBalletjes[i].VY = r.Next(-2, 3);
11 }
12
13 while (true)
14 {
15     for (int i = 0; i < veelBalletjes.Length; i++)
16     {
17         veelBalletjes[i].Update(); //update alle balletjes
18     }
19     for (int i = 0; i < veelBalletjes.Length; i++)
20     {
21         veelBalletjes[i].TekenOpScherm(); //teken alle balletjes
22     }
23     System.Threading.Thread.Sleep(50);
24     Console.Clear();
25 }
```

De reden dat we nu twee loops gebruiken, is omdat we in de updatefase eerst alle objecten willen bijwerken (soms in relatie tot andere objecten) voordat we alles opnieuw op het scherm tekenen. Anders kan het zijn dat je vreemde effecten te zien krijgt als je bijvoorbeeld balletjes tegen elkaar wil laten wegbotsen.

Ok, zwem maar snel naar de kant. We gaan al het voorgaande van begin tot einde uit de doeken doen! Leg die handdoek niet te ver weg, we gaan hem nog nodig hebben.

9.2 Klassen en objecten

Een elementair aspect binnen OOP is het verschil begrijpen tussen een klasse en een object.

Wanneer we meerdere objecten gebruiken van dezelfde soort dan kunnen we zeggen dat deze objecten allemaal deel uitmaken van eenzelfde klasse. **Het OOP paradigma houdt ook in dat we de echte wereld gaan proberen te modeleren in code.** OOP laat namelijk toe om onze code zo te structureren zoals we dat ook in het echte leven doen. Alles om ons heen behoort tot een bepaalde klasse die alle objecten van dat type beschrijven.

Neem eens een kijkje aan een druk kruispunt waar fietsers, voetgangers, auto's en verkeerslichten samenkommen¹. Het is een erg hectisch geheel, toch kan je alles dat je daar ziet *classificeren*. We zien bijvoorbeeld allemaal mens-objecten die tot de klasse van de Mens behoren, maar ook:

- Alle mensen hebben gemeenschappelijke eigenschappen (binnen deze beperkte context van een kruispunt): ze bewegen of staan stil (gedrag), ze hebben een bepaalde kleur van jas (eigenschap).
- Alle auto's behoren tot een klasse Auto. Ze hebben gemeenschappelijke zaken zoals: een bouwjaar (eigenschap), ze werken op een bepaalde vorm van energie (eigenschap) en ze staan stil of bewegen (gedrag).
- Ieder verkeerslicht behoort tot de klasse VerkeersLicht.
- Fietsers behoren tot de klasse Fietser.

9.2.1 Definitie klasse en object

Volgende 2 definities druk je best af op een grote poster die je boven je bed hangt:

- **Een klasse** is als een **blauwdruk** (of prototype) dat het gedrag en toestand beschrijft van alle objecten van deze klasse.
- Een individueel **object** is een **instantie** van een klasse en heeft een eigen *toestand, gedrag* en *identiteit*.

Objecten zijn instanties met een eigen levenscyclus die wordt gekenmerkt door:

- **Gedrag**: deze wordt beschreven door de **methoden** in de klasse.
- **Toestand**: deze kan wijzigen door zijn eigen gedrag, of door externe impulsen en wordt bepaald door **datavelden** die beschreven staan in de klasse (properties en instantievariabelen).
- **Identiteit**: een unieke naam van object zodat andere objecten ermee kunnen interageren.

¹Dit voorbeeld is gebaseerd op de inleiding van het inzichtvolle boek "Handboek objectgeoriënteerd programmeren" door Jan Beurghs (EAN: 9789059406476).



Je zou dit kunnen vergelijken met het grondplan voor een huis dat tien keer in een straat zal gebouwd worden. Het plan is de *klasse*. De effectieve huizen die we bouwen aan de hand van dit plan zijn de instanties of objecten van deze klasse. Ieder huis heeft een eigen toestand (ander type bakstenen, wel of geen zonnepannelen) en gedrag (rolluiken gaan open als de zon opkomt).

De klasse beschrijft het algemene **gedrag** van de individuele objecten. Dit gedrag wordt meestal bepaald door de interne staat van ieder object op zichzelf, de zogenaamde **eigenschappen**. Nemen we het voorbeeld van de klasse Auto: de huidige snelheid van een individueel auto-object is mogelijks gebaseerd op het merk (eigenschap) van die auto, alsook welke energiebron (eigenschap) die auto heeft.

Voorts kunnen objecten ook beïnvloed worden door ‘de buitenwereld’: naast de interne staat van ieder object, leven de objecten natuurlijk in een bepaalde context, zoals een druk kruispunt. Andere objecten op dat kruispunt kunnen invloed hebben op wat een auto-object doet.

Met andere woorden: we kunnen ‘van buiten uit’ vaak ook het gedrag en de interne staat van een object aanpassen. We hebben dit reeds zien gebeuren in het Pong-voorbeeld: de interne staat van ieder individueel balletjes-object is z’n positie alsook z’n richtingsvector. De buitenwereld, in dit geval onze Main methode kon echter de objecten manipuleren:

- Het gedrag van een balletje konden we aanpassen met behulp van de Update en TekenOpScherm methode.
- De interne staat via de eigenschappen die zichtbaar zijn aan de buitenwereld (dankzij het **public** keyword).



Wanneer je later de specificaties voor een opdracht krijgt en snel wilt ontdekken wat potentiële klassen zijn, dan is het een goede tip om op zoek te gaan naar de zelfstandige naamwoorden (*substantieven*) in de tekst. Dit zijn meestal de objecten en/of klassen die jouw applicatie zal nodig hebben.



95% van de tijd zullen we in dit boek de voorgaande definitie van een klasse beschrijven, namelijk de blauwdruk voor de objecten die er op gebaseerd zijn. Je zou kunnen zeggen dat de klasse een fabriekje is dat objecten kan maken. Echter, wanneer we het **static** keyword zullen bespreken gaan we ontdekken dat heel af en toe een klasse ook als een soort object door het leven kan gaan. Heel vreemd allemaal!

9.2.2 Abstractie en encapsulatie

Een belangrijk concept bij OOP is het **Black-box** principe waarbij we de afzonderlijke objecten en hun werking als zwarte dozen gaan beschouwen.

Neem het voorbeeld van de auto: deze is in de echte wereld ontwikkeld volgens het blackbox-principe. De werking van de auto kennen tot in het kleinste detail is niet nodig om met een auto te kunnen rijden. De auto biedt een aantal zaken aan de buitenwereld aan (het stuur, pedalen, het dashboard), wat we de “**interface**” noemen. De interface kan je gebruiken om de interne staat van de auto uit te lezen of te manipuleren. Stel je voor dat je moet weten hoe een auto volledig werkte voor je ermee op de baan kon...

Binnen OOP wordt dit blackbox-concept **abstractie** en **encapsulatie** genoemd. Het doel van OOP is programmeurs zoveel mogelijk af te schermen van de interne werking van je klasse code. Vergelijk het met de methoden uit hoofdstuk 7: “if it works, it works” en dan hoeft je niet in de code van de methode te gaan zien wat er juist gebeurt telkens je de methode wil gebruiken.

Bij encapsulatie bedoelen we dat we alle zaken die samen horen bij een auto samenvoegen tot één geheel. De motor, de banden, ieder componentje in de motor, enz. Al deze zaken *encapsuleren* we in één geheel. Vervolgens gaan we met de hulp van abstractie de complexiteit naar de buitenwereld toe kunnen vereenvoudigen.

Kortom: *hoe minder de buitenwereld moet weten om met een object te werken, hoe beter.*

Beeld je in dat je 10 lijnen code nodig had om een random getal te genereren. Niemand zou de klasse Random nog gebruiken. Dankzij de ontwikkelaar van deze klasse hoeven we maar 2 zaken te kunnen:

- Een Random-object aanmaken met **new**/
- De Next-methode aanroepen om een getal uit het object te krijgen

Wat er nu juist in die methode gebeurt boeit ons niet. *It just works!* Met dank aan abstractie en de kracht van OOP.

9.2.3 Tijd voor taart

Een truukje om de belangrijkste concepten van OOP te onthouden is het acroniem **A PIE**, dat staat voor:

- **A**bstractie
- **P**olymorfisme
- **I**nheritance, oftewel overerving
- **E**ncapsulatie

Abstractie (het vereenvoudigen van de complexe, interne structuur van een klasse) en encapsulatie (alle aspecten die in de klasse horen) hebben we net besproken. Polymorfisme zullen we pas in hoofdstuk 16 aanpakken. Inheritance komt al in hoofdstuk 13 in actie. Nog even geduld dus.

9.2.4 Objecten in de woorden van Steve Jobs

Steve Jobs, de oprichter van Apple, was een fervent fan van OOP. In een interview in 1994 voor het Rolling Stone magazine gaf hij volgende uitleg:

"Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here."

Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."

You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet, I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level."

Vooral die laatste zin verdient het om nog eens in vet herhaald te worden: **"They encapsulate complexity, and the interfaces to that complexity are high level."**

9.2.5 Objecten in de woorden van Bill Gates

En, omdat het vloeken in de kerk is om Steve Jobs in een C# boek aan het woord te laten, hier wat Microsoft-oprichter Bill Gates over OOP te zeggen had:

"Another trick in software is to avoid rewriting the software by using a piece that's already been written, so called component approach which the latest term for this in the most advanced form is what's called Object Oriented Programming."



Ik zie dat je gereedsschapkist al aardig gevuld is. Zoals je misschien al gemerkt hebt aan deze sectie, zullen we vanaf nu ook geregeld minder “praktische” en eerder “filosofische” zaken tegenkomen. Maar wees gerust, je zal toch een grotere gereedschapkist nodig hebben. Echter, net zoals een voorman niet alleen moet kunnen metsen en timmeren, maar ook stabiliteitsplannen begrijpen, zal ook jij moeten begrijpen wat de grotere ideeën achter bepaalde concepten zijn.

Zet nu je helm maar op, want in de volgende sectie gaan we wel degelijk onze handen lekker vuil maken!

9.3 OOP in C#

In C# kunnen we geen objecten aanmaken zonder eerst een klasse te definiëren. Een klasse beschrijft de algemene eigenschappen (properties en instantievariabelen) en het gedrag (methoden) van die objecten.

9.3.1 Klasse maken

Een klasse heeft minimaal de volgende vorm:

```
1 class ClassName  
2 {  
3  
4 }
```



De naam die je een klasse geeft moet voldoen aan de identifier regels uit hoofdstuk 2. Het is echter een goede gewoonte om **klassenamen altijd met een hoofdletter te laten beginnen**.

Volgende code beschrijft de klasse Auto in C#

```
1 class Auto  
2 {  
3  
4 }
```

Binnen het codeblock dat bij deze klasse hoort zullen we verderop dan de werking via properties en methoden beschrijven.

9.3.2 Klassen in Visual Studio toevoegen

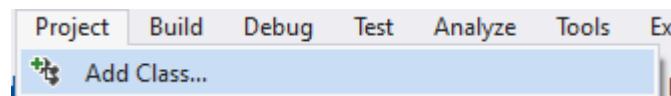
Je kan “eender waar” een klasse aanmaken in een project, maar het is een goede gewoonte om per klasse **een apart bestand** te gebruiken. Dit kan op 2 manieren.

Manier 1:

- In de solution Explorer, rechtersleutel op je project.
- Kies “Add”.
- Kies “Class..”.
- Geef een goede naam voor je klasse.

Manier 2:

- Klik in de menubalk bovenaan op “Project”.
- Kies “Add class...” .



Figuur 9.1: Manier 2 is de snelste. Tip: of maak een eigen toetsenbord shortcut, dat is nog sneller natuurlijk.

Je zal zien dat nieuw toegevoegde klassen in Visual Studio ook nog het keyword **internal** voor **class** krijgen. Dit is een zogenaamde **access modifier** en leg ik zo meteen uit.

9.3.3 Objecten aanmaken

Je kan nu objecten aanmaken van de klasse die je hebt gedefinieerd. Dit kan op alle plaatsen in je code waar je in het verleden ook al variabelen kon declareren, bijvoorbeeld in een methode of je Main-methode.

Je doet dit door eerst een variabele te definiëren en vervolgens een object te **instantiëren** met behulp van het **new** keyword. De variabele heeft als datatype Auto:

```
1 Auto mijnEersteAuto = new Auto();  
2 Auto mijnAndereAuto = new Auto();
```

We hebben nu **twee objecten aangemaakt van het type Auto** die we verderop zouden kunnen gebruiken.

Let goed op dat je dus op de juiste plekken dit alles doet:

- Klassen maak je aan als aparte bestanden in je project.
- Objecten creëer je in je code op de plekken waar je deze nodig hebt, bijvoorbeeld in je Main methode bij een Console-applicatie.

9.3.4 De new operator

In het volgende hoofdstuk leg ik uit wat er allemaal gebeurt in het geheugen wanneer we een object met **new** aanmaken. Het is echter nu al belangrijk te beseffen dat objecten niet kunnen gemaakt worden zonder **new**.

De **new** operator vereist dat je aangeeft van welke klasse (het type) je een object wilt aanmaken, gevuld door ronde haakjes. Bijvoorbeeld:

```
1 new Student();
```

Deze lijn code doet niets nuttig. We roepen hier weliswaar een constructor aan (zie verder) die het object in het geheugen zal aanmaken. Vervolgens geeft **new** een adres terug waar het object zich bevindt. We doen nog niets met dit adres.

Het is dit adres dat we vervolgens kunnen bewaren in een variabele die links van de toekenningsoperator (=) staat:

```
1 Student eersteStudentObject = new Student();
```

Test eens wat er gebeurt als je volgende code probeert te compileren:

```
1 Auto mijnEersteAuto = new Auto();
2 Auto mijnAndereAuto;
3 Console.WriteLine(mijnEersteAuto);
4 Console.WriteLine(mijnAndereAuto);
```

Je zal een **"Use of unassigned local variable mijnAndereAuto"** foutbericht krijgen. Indraad, je hebt nog geen object aangemaakt met **new** en **mijnAndereAuto** is dus voorlopig een lege doos (**het heeft de waarde null**).



Dit concept is dus fundamenteel verschillend van de klassieke *valuetypes* die we al kenden (**int, double**, enz.). Daar zal volgende code wél werken:

```
1 int balans;
2 Console.WriteLine(balans);
```

9.3.5 Klassen zijn gewoon nieuwe datatypes

In hoofdstuk 2 leerden we dat er allerlei datatypes bestaan. We maakten vervolgens variabelen aan van een bepaald datatype zodat deze variabele als inhoud enkel zaken kon bevatten van dat ene datatype.

Zo leerden we toen volgende categorieën van datatypes:

- **Valuetypes** zoals `int`, `char` en `bool`.
- Het `enum` keyword liet ons toe om een nieuw datatype te maken dat maar een eindig aantal mogelijke waarden (values) kon hebben. Intern bewaarden variabelen van zo'n enum-datatype hun waarde als een `int`.
- **Arrays** waren het laatste soort datatypes. Je ontdekte dat je arrays kon maken van eender welk datatype (valuetypes en enums).

Wel nu, klassen zijn niet meer dan een nieuw soort datatypes. Kortom: telkens je een klasse aanmaakt, kunnen we in dat project variabelen en arrays aanmaken met dat datatype. We noemen variabelen die een klasse als datatype hebben **objecten**.

Het grote verschil dat deze objecten zullen hebben is dat ze vaak veel complexer zijn dan de eerdere datatypes die we kennen:

- Ze zullen meerdere “waarden” tegelijk kunnen bewaren (een `int` variabele kan maar één waarde tegelijkertijd in zich hebben).
- Ze zullen methoden hebben die we kunnen aanroepen om het object voor ons te laten werken.



Het blijft ingewikkeld hoor. Heel boeiend om de theorie van een speer te leren, maar ik denk dat ik toch beter een paar keer met een speer naar een mammoetwerp om echt te voelen wat OOP is.

Ik onthoud nu alvast “klassen zijn gewoon een nieuwe vorm van complexere datatypes” dan diegene die ik tot nog toe heb geleerd? Ok?

Correct. Er verandert dus niet veel. Enkel je variabelen worden krachtiger!

9.3.6 De anatomie van een klasse

Ik zal nu enkele basisconcepten van klassen en objecten toelichten aan de hand van praktische voorbeelden.

9.3.6.1 Object methoden

Stel dat we een klasse willen maken die ons toelaat om objecten te maken die verschillende mensen voorstellen. We willen aan iedere mens kunnen zeggen “Praat eens”.

We maken een nieuwe klasse Mens en plaatsen in de klasse een methode Praat:

```
1 internal class Mens
2 {
3     public void Praat()
4     {
5         Console.WriteLine("Ik ben een mens!");
6     }
7 }
```

We zien twee nieuwe aspecten:

- Het keyword **static** mag je **niet** voor een methode signatuur zetten (later ontdekken we wanneer dat soms wel moet).
- Voor de methode plaatsen we **public** : dit is een *access modifier* die aangeeft dat de buitenwereld deze methode op het object kan aanroepen.

Je kan nu elders objecten aanmaken en ieder object z'n methode Praat aanroepen:

```
1 Mens joske = new Mens();
2 Mens alfons = new Mens();
3 joske.Praat();
4 alfons.Praat();
```

Er zal twee maal `Ik ben een mens!` op het scherm verschijnen. Waarbij joske en alfons zelf verantwoordelijk hiervoor waren dat dit gebeurde.

9.3.6.2 Access modifiers

De **access modifier** geeft aan hoe zichtbaar een bepaald deel van de klasse en de klasse zelf is. Wanneer je niet wilt dat “van buiten” een bepaalde methode kan aangeroepen worden, dan dien je deze als **private** in te stellen. Wil je dit net wel dat moet je er explicet **public** of **internal** voor zetten.

Test in de voorgaande klasse eens wat gebeurt wanneer je **public** vervangt door **private**. Inderdaad, je zal de methode Praat niet meer op de objecten kunnen aanroepen.



Wanneer je geen access modifier voor een methode of klasse zet in C# dan zal deze als **private beschouwd worden.** Dit geldt voor alle zaken waar je access modifiers voor kan zetten: niets ervoor zetten wil zeggen **private**.

Volgende twee methoden-signaturen zijn dus identiek:

```
1 private void NiemandMagDitGebruiken()
2 {
3     //...
4 }
5
6 void NiemandMagDitGebruiken()
7 {
8     //...
9 }
```

Het is een héél slechte gewoonte om géén access modifiers voor iedere methode te zetten. Maak er dus een gewoonte van dit steeds ogenblikkelijk te doen.

Test volgende klasse eens, kan je de methode VertelGeheim vanuit de Main op joske aanroepen?

```
1 internal class Mens
2 {
3     public void Praat()
4     {
5         Console.WriteLine("Ik ben een mens!");
6     }
7
8     private void VertelGeheim()
9     {
10        Console.WriteLine("Ik ben verliefd op Anneke");
11    }
12 }
```



Access modifiers hebben een volgorde van de mate waarin ze beschermen. Aan het ene uiterste heb je **private** en aan de andere kant **public**. Daartussen zitten er nog enkele andere, allemaal met hun specifieke bestaansreden.

Klassen zijn ofwel **public** oftewel **internal**. Indien je niets voor de klasse zet dan is deze **internal**. Concreet wil dit zeggen dat een **internal** klasse enkel binnen het huidige project (de *assembly*) kunt gebruiken.

Onderdelen (hoofdzakelijk methoden en datavelden) in een klasse kunnen volgende modifiers hebben:

- **private**: het meest beschermend. Enkel zichtbaar in de klasse zelf. Dit is de standaardwaarde als je geen access modifier expliciet schrijft.
- **protected**: enkel zichtbaar voor overgeërfd klassen (zie hoofdstuk 16).
- **public** : het meest open. Iedereen kan hier aan...en dat raden we ten stelligste af.

Er zijn er nog enkele andere (**protected internal**, **private internal** en **file**), maar die bespreken we niet in dit boek.

9.3.6.3 Reden van private

Waarom zou je bepaalde zaken **private** maken?

De code binnenin een klasse kan overal aan binnen de klasse zelf. Stel dat je dus een erg complexe publieke methode hebt, en je wil deze opsplitsen in meerdere delen, dan ga je die andere delen **private** maken. Dit voorkomt dat programmeurs die je klasse later gebruiken, stukken code aanroepen die helemaal niet bedoeld zijn om rechtstreeks aan te roepen.

Volgende voorbeeld toont hoe je binnenin een klasse andere zaken van de klasse kunt aanroepen: we roepen in de methode Praat de methode VertelGeheim aan. Dit kan want **private** geldt enkel voor de buitenwereld van de klasse, maar dus niet voor de code binnen de Praat-methode zelf.

```
1 internal class Mens
2 {
3     public void Praat()
4     {
5         Console.WriteLine("Ik ben een mens!");
6         VertelGeheim();
7     }
8
9     private void VertelGeheim()
10    {
11        Console.WriteLine("Ik ben verliefd op Anneke");
12    }
13 }
```

Als we nu elders een object laten praten als volgt:

```
1 Mens rachid = new Mens();
2 rachid.Praat();
```

Dan zal de uitvoer worden:

```
1 Ik ben een mens!
2 Ik ben verliefd op Anneke
```



Met behulp van de **dot-operator** (.) kunnen we aan alle informatie die ons object aanbiedt aan de buitenwereld. Ook dit zag je reeds toen je een Random-object hadden: we konden maar een handvol zaken aanroepen op zo'n object, waaronder de Next methode.

Het is natuurlijk een beetje vreemd dat nu al onze objecten zeggen dat ze verliefd zijn op Anneke. Dit is niet het smurfendorp met maar 1 meisje! Dit gaan we verderop oplossen. *Stay tuned!*

9.3.6.4 Instantievariabelen

Voorlopig doen alle objecten van het type Mens hetzelfde. Ze kunnen praten en zeggen hetzelfde.

We weten echter dat objecten ook een interne staat hebben die per object individueel is (we zagen dit reeds toen we balletjes over het scherm lieten botsen: ieder balletje onthield z'n eigen richtingsvector en positie). Dit kunnen we dankzij **instantievariabelen** (ook wel **datavelden** of **datafields** genoemd) oplossen. Dit zullen variabelen zijn waarin zaken kunnen bewaard worden die verschillen per object.

Stel je voor dat we onze mensen een geboortejaar willen geven. Ieder object zal zelf in een instantievariabele bijhouden wanneer ze geboren zijn (het vertellen van geheimen zullen we verderop behandelen):

```
1 internal class Mens
2 {
3     private int geboorteJaar = 1970; //instantievariabele
4
5     public void Praat()
6     {
7         Console.WriteLine("Ik ben een mens! ");
8         Console.WriteLine($"Ik ben geboren in {geboorteJaar}.");
9     }
10 }
```

Enkele belangrijke concepten:

- De instantievariabele `geboorteJaar` zetten we `private`: we willen niet dat de buitenwereld het geboortejaar van een object kan aanpassen. Beeld je in dat dat in de echte wereld ook kon. Dan zou je naar je kameraad kunnen roepen “Hey Adil, jouw geboortejaar is nu 1899! Ha!” Waarop Adil vloekend verandert in een steenoud mannetje.
- We geven de variabele een beginwaarde 1970. Alle objecten zullen dus standaard in het jaar 1970 geboren zijn wanneer we deze met `new` aanmaken.
- We kunnen de inhoud van de instantievariabelen lezen en veranderen vanuit andere delen in de code. Zo gebruiken we `geboorteJaar` in de tweede lijn van de `Praat` methode. Als je die methode nu zou aanroepen dan zou het geboortejaar van het object dat je aanroeft mee op het scherm verschijnen.



Ik moet ook dringend enkele extra *niet-officiële* identifier regels in het leven roepen:

- Klassenamen en methoden in klassen beginnen altijd met een hoofdletter.
- Alles dat `public` is in een klasse begint ook met een hoofdletter.
- Alles dat `private` is begint met een kleine letter (of liggend streepje), tenzij het om een methode gaat, die begint altijd met een hoofdletter.

Dit zijn geen officiële regels, maar afspraken die veel programmeurs onderling hebben gemaakt. Het maakt de code leesbaarder.



Wat?! Ik ben hier niet voor jou? Omdat je geen `goto` hebt gebruikt?! Flink hoor. Maar daarvoor ben ik hier niet. Ik zag je wel denken: “Als ik nu die instantievariabele ook eens `public` maak.” Niet doen. Simpel! Instantievariabele mogen NOOIT public gezet worden.

De C# standaard laat dit weliswaar toe, maar dit is één van de slechtste programmeerdingen die je kan doen. Wil je toch de interne staat van een object kunnen aanpassen dan gaan we dat via `properties` en `methoden` kunnen doen, wat we zo meteen gaan uitleggen. Zie dat ik hier niet te vaak tussenbeide moet komen. Dank!

Ok, we zullen maar luisteren naar meneer de agent. Stel nu dat we een verjongingsstraal hebben. Hiermee kunnen we het geboortejaar van de mensen steeds met 1 jaar kunnen verhogen. We maken ze met andere woorden telkens een jaartje jonger!

```
1 internal class Mens
2 {
3     private int geboorteJaar = 1970;
4     public void Praat()
5     {
6         Console.WriteLine("Ik ben een mens!");
7         Console.WriteLine($"Ik ben geboren in {geboorteJaar}.");
8     }
9     public void StartVerjongingskuur()
10    {
11        Console.WriteLine("Jeuj. Ik word jonger!");
12        geboorteJaar++;
13    }
14 }
```

Zoals al gezegd: **Ieder object zal z'n eigen geboortejaar hebben.**

Die laatste opmerking is een kernconcept van OOP: ieder object heeft z'n eigen interne staat die kan aangepast worden individueel van de andere objecten van hetzelfde type. We zullen dit testen in volgende voorbeeld waarin we 2 objecten maken en enkel 1 ervan verjengen. Kijk wat er gebeurt:

```
1 Mens elvis = new Mens();
2 Mens bono = new Mens();
3 elvis.StartVerjongingskuur();
4 elvis.Praat();
5 bono.Praat();
```

Als je voorgaande code zou uitvoeren zal je zien dat het geboortejaar van Elvis verhoogd en niet die van Bono wanneer we StartVerjongingskuur aanroepen. Zoals het hoort!

De uitvoer zal zijn:

```
1 Jeuj. Ik word jonger!
2 Ik ben een mens!
3 Ik ben geboren in 1971.
4 Ik ben een mens!
5 Ik ben geboren in 1970.
```



“Ja maar, nu pas je toch het geboortejaar van buiten aan via een methode, ook al gaf je aan dat dit niet de bedoeling was want dan zou je Adil ogenblikkelijk erg jong kunnen maken.”

Correct. Maar dat was dus maar een voorbeeld. De hoofdreden dat we instantievariabelen niet zomaar **public** mogen maken is om te voorkomen dat de buitenwereld instantievariabelen waarden geeft die de werking van de klasse zouden stuk maken. Stel je voor dat je dit kon doen: `adil.geboortejaar = -12000;`

Dit kan nefaste gevolgen hebben voor de klasse.

Daarom gaan we de toegang tot instantievariabelen als het ware controleren door deze enkel via properties en methoden toe te laten. We zouden dan bijvoorbeeld het volgende kunnen doen:

```
1 internal class Mens
2 {
3     private int geboorteJaar = 1970;
4
5     public void VeranderGeboortejaar(int geboorteJaarIn)
6     {
7         if(geboorteJaarIn >= 1900)
8             geboorteJaar = geboorteJaarIn;
9     }
10    //...
```

Mooi he. Zo voorkomen we dus dat de buitenwereld illegale waarden aan een variabele kan geven. In dit voorbeeld kunnen mensen dus niet voor het jaar 1900 geboren zijn. **Objecten zijn verantwoordelijk voor zichzelf** en moeten zichzelf dus ook beschermen zodat de buitenwereld niets met hen doet dat hun eigen werking om zeep helpt.

9.3.6.5 Andere lieven

We kunnen nu het probleem oplossen dat al onze mensen verliefd zijn op Anneke. Volgende code toont dit:

```
1 internal class Mens
2 {
3     private string lief = "niemand";
4
5     public void VeranderLief(string nieuwLief)
6     {
7         lief = nieuwLief;
8     }
9     public void Praat()
10    {
11        Console.WriteLine("Ik ben een mens!");
12        VertelGeheim();
13    }
14
15     private void VertelGeheim()
16    {
17        if( lief != "niemand")
18            Console.WriteLine($"Ik ben verliefd op {lief}.");
19        else
20            Console.WriteLine("Ik ben op niemand verliefd.");
21    }
22 }
```

Nu kunnen we dus “*Temptation Island - de OOP editie*” beginnen:

```
1 Mens deelnemer1 = new Mens();
2 Mens deelnemer2 = new Mens();
3 deelnemer1.Praat();
4 deelnemer2.Praat();
5
6 deelnemer2.VeranderLief("Phoebe");
7 deelnemer1.Praat();
8 deelnemer2.Praat();
9
10 deelnemer1.VeranderLief("Camilla");
11 deelnemer1.Praat();
12 deelnemer2.Praat();
```

De uitvoer van voorgaande code zal zijn:

```
1 Ik ben een mens!
2 Ik ben op niemand verliefd.
3 Ik ben een mens!
4 Ik ben op niemand verliefd.
5 Ik ben een mens!
6 Ik ben op niemand verliefd.
7 Ik ben een mens!
8 Ik ben verliefd op Phoebe.
9 Ik ben een mens!
10 Ik ben verliefd op Camilla.
11 Ik ben een mens!
12 Ik ben verliefd op Phoebe.
```



Veel beginnende programmeurs maken fouten op het correct kunnen onderscheiden wat de klassen en wat de objecten in hun opgave juist zijn. Het is altijd belangrijk te begrijpen dat een klasse weliswaar beschrijft hoe alle objecten van dat type werken, maar op zich gaat die beschrijving steeds over 1 object uit de verzameling. Say what now?!

9.3.7 Klasse Studenten of Student?

Als je een klasse Student hebt, dan zal deze eigenschappen hebben zoals Punten, Naam en Geboortejaar. Als je een klasse Studenten daarentegen hebt, dan is dit vermoedelijk een klasse die beschrijft hoe een groep studenten moet werken in je applicatie. Mogelijk zal je dan properties hebben zoals KlasNaam, AantalAfwezigen, enz. Kortom, eigenschappen over de groep, niet over 1 student.

9.3.7.1 Level of Level1?

Een andere veelgemaakte fout is klassen te schrijven die maar exact één object kan en moet creëren. Dit soort klasse noemt een *singleton*. Stel je voor dat je een spel maakt waarin verschillende levels zijn. Een logische keuze zou dan zijn om een klasse Level te maken (niet Levels) die properties heeft zoals MoeilijkheidsGraad, HeeftGeheimeGrotten, AantalVijanden, enz.

Vervolgens kunnen we dan instanties maken: *1 object stelt 1 level in het spel voor*. De speler kan dan van level naar level gaan en de code start dan bijvoorbeeld telkens de BeginLevel methode:

```
1 Level level1 = new Level();  
2 level1.BeginLevel();
```

Wat dus niet mag zijn **klassen** met namen zoals `level1`, `level2`, enz. Vermoedelijk hebben deze klasse 90% gelijkaardige code en is er dus een probleem met wat we de *architectuur* van je code noemen. Of duidelijker: je snapt niet wat het verschil is tussen klassen en objecten!

Objecten met namen zoals `level1` en `level2` zijn wél dus toegestaan, daar ze dan vermoedelijk allemaal van het type `Level` zijn. **Maar opgelet: als je variabelen hebt die genummerd zijn (bv. `bal1`, `bal2`, enz.) dan is de kans groot dat je eigenlijk een array van objecten nodig hebt** (wat ik in hoofdstuk 8 zal uitleggen).

9.4 Properties

We zagen zonet dat instantievariabelen nooit **public** mogen zijn om te voorkomen dat de buitenwereld onze objecten ‘vult’ met slechte zaken. Het voorbeeld waarbij we vervolgens een methode **StartVerjongingskuur** gebruikten om op gecontroleerde manier toch aan de interne staat van objecten te komen is één oplossing, maar een nogal *oldschool* oplossing.

Deze manier van werken - methoden gebruiken om instantievariabelen aan te passen of uit te lezen - is wat voorbij gestreefd binnen C#. Onze programmeertaal heeft namelijk het concept **properties** (*eigenschappen*) in het leven geroepen die toelaten op een eenvoudigere manier aan de interne staat van objecten te geraken.



Properties (*eigenschappen*) zijn de C# manier om objecten hun interne staat in en uit te lezen. Ze zorgen voor een gecontroleerde toegang tot de interne structuur van je objecten.

9.4.1 Star Wars en de nood aan properties

In het Star Wars universum heb je goede oude “Darth Vader”. Hij behoort tot de mysterieuze klasse van de Sith Lords. Deze Lords lopen met een geheim rond: ze hebben een *Sithnaam*. Deze naam mogen de Lords enkel bekend maken aan andere Sith Lords. Voorts heeft een Sith Lord ook een hoeveelheid energie (*The Force*) waarmee hij kattekwaad kan uithalen. Deze energie kan nooit onder nul gezet worden.

We kunnen voorgaande als volgt schrijven:

```
1 internal class SithLord
2 {
3     private int energie;
4     private string sithName;
5 }
```

Het is uit den boze dat we eenvoudige instantievariabelen (energie en name) **public maken.** Zouden we dat wel doen dan kunnen externe objecten deze geheime informatie uitlezen!

```
1 SithLord palpatine = new SithLord();
2 Console.WriteLine(palpatine.sithName); //zal niet compileren!
```

We willen echter wel van buiten uit het energie-level van een sithLord kunnen instellen. Maar ook hier hetzelfde probleem: wat als we de energie-level op -1000 instellen? Terwijl energie nooit onder 0 mag gaan.

Properties lossen dit probleem op.

9.4.2 2 soorten properties

Er zijn 2 soorten properties² in C#:

- **Full properties:** deze stijl van properties verplicht ons véél code te schrijven, maar we hebben ook volledige controle over wat er gebeurt.
 - **Auto-properties:** deze zijn exact het omgekeerde van full properties. Je moet niet veel code schrijven, maar je hebt ook weinig (eigenlijk géén) controle over wat er gebeurt.

Ik behandel eerst full properties, omdat auto-properties een soort afgeleide van full properties zijn. Bepaalde aspecten van full properties worden bij auto-properties achter de scherm verstopt zodat jij als programmeur er geen last van hebt.

9.4.3 Full properties

Properties herken je aan de **get** en **set** keywords in een klasse. Een property is een beschrijving van wat er moet gebeuren indien je informatie uit (**get**) een object wilt halen of informatie in (**set**) een object wilt plaatsen.

In volgende voorbeeld maken we een property, genaamd Energie aan. Deze doet niets anders dan rechtstreeks toegang tot de instantievariabele energie te geven:

```
1 internal class SithLord
2 {
3     private int energie;
4
5     public int Energie
6     {
7         get
8         {
9             return energie;
10        }
11        set
12        {
13            energie = value;
14        }
15    }
16 }
```

Dankzij voorgaande code kunnen we nu buiten het object de property Energie gebruiken als volgt:

```
1 SithLord Vader = new SithLord();
2 Vader.Energie = 20; //set
3 Console.WriteLine($"Vaders energie is {Vader.Energie}"); //get
```

²In één van de volgende versies van C# (normaal versie 11) zal er nog een derde type verschijnen: *semi-auto properties*. Een propertytype dat zich tussen beide bestaande types zal bevinden. De details en exacte gebruik ervan worden nog besproken op github.com door de ontwikkelaars, dus het is nog te vroeg om deze al op te nemen in dit boek.

Laten we eens inzoomen op de full property code.

9.4.3.1 Full property: identifier en datatype

De eerste lijn van een full property beschrijft de naam (identifier) en datatype van de property:

public int Energie

Een property is altijd public daar dit de essentie van een property net is “de buitenwereld gecontroleerde toegang tot de interne staat van een object geven”.

Vervolgens zeggen we wat voor **datatype** de property moet zijn en geven we het een naam die moet voldoen aan de identifier regels van weleer. Voor de buitenwereld zal een property zich gedragen als een gewone variabele, met de naam **Energie** van het type **int**.

Indien je de property gaat gebruiken om een instantievariabele naar buiten beschikbaar te stellen, dan is het een goede gewoonte om dezelfde naam als dat veld te nemen maar nu met een hoofdletter (dus **Energie** i.p.v. **energie**).

9.4.3.2 Full property: get gedeelte

Indien je wenst dat de property data **naar buiten** kan sturen, dan schrijven we de get-code. Binnen de accolades van de **get** schrijven we wat er naar buiten moet gestuurd worden.

```
1 get
2 {
3     return energie;
4 }
```

Dit werkt dus identiek aan een methode met een returntype. **Het element dat je met return teruggeeft in de get code moet uiteraard van hetzelfde type zijn als waarmee je de property hebt gedefinieerd (int in dit geval).**

We kunnen nu van buitenaf toch de waarde van **energie** uitlezen via de property en het get-gedeelte, bijvoorbeeld **int** uitgelezen = **palpatine.Energie**;



We mogen eender wat doen in het get-gedeelte (net zoals bij methoden) zolang er finaal maar iets uitgestuurd wordt m.b.v. **return**. Ik zal hier verderop meer over vertellen, want soms is het handig om *getters* te schrijven die de data transformeren voor ze uitgestuurd wordt.

9.4.3.3 Full property: set gedeelte

In het set-gedeelte schrijven we de code die we moeten hanteren indien men van buiten een waarde aan de property wenst te geven om zo een instantievariabele aan te passen.

```
1  set
2  {
3      energie = value;
4 }
```

De waarde die we van buiten krijgen (als een parameter zeg maar) zal altijd in een lokale variabele **value** worden bewaard binnendoor de set-code. Deze zal van het type van de property zijn.



Deze **value** parameter is een geserveerd keyword van de **set** syntax en kan je niet hernoemen of voor iets anders gebruiken.

Vervolgens kunnen we **value** toewijzen aan de interne variabele indien gewenst: **energie = value;**. Uiteraard kunnen we die toewijzing dus ook gecontroleerd laten gebeuren, wat ik zo meteen uitleg.

We kunnen vanaf nu van buitenaf waarden toewijzen aan de property en zo **energie** toch bereiken: **palpatine.Energie = 50;**.



Je bent niet verplicht om een property te maken wiens naam overeen komt met een bestaande instantievariabele (**maar dit wordt wel aangeraden**).

Dit mag dus ook:

```
1  internal class Auto
2  {
3      private int benzinePeil;
4
5      public int FuelLevel
6      {
7          get { return benzinePeil; }
8          set { benzinePeil = value; }
9      }
10 }
```



Visual Studio heeft een ingebouwde *snippet* om snel een full property, inclusief een bijhorende private instantievariabele, te schrijven. **Typ “propfull” gevuld door twee maal op de tab-toets te duwen.**

9.4.4 Full property met toegangscontrole

De full property Energie heeft nog steeds het probleem dat we negatieve waarden kunnen toewijzen (via de **set**) die dan vervolgens zal toegewezen worden aan energie.

Properties hebben echter de mogelijkheid om op te treden als wachters van en naar de interne staat van objecten.

We kunnen in de **set** code extra controles inbouwen. Aangezien de variabele `value` de waarde krijgt die we extern aan de property toewijzen, kunnen we deze controleren en zo nodig de toewijzing voorkomen. Volgende voorbeeld zal enkel de waarde toewijzen indien deze groter of gelijk aan 0 is:

```
1 public int Energie
2 {
3     get
4     {
5         return energie;
6     }
7     set
8     {
9         if(value >= 0)
10            energie = value;
11    }
12 }
```

Volgende lijn zal dus geen effect hebben:

```
1 palpatine.Energie = -1;
```

We mogen de code binnen **set** en **get** zo complex maken als we zelf willen.

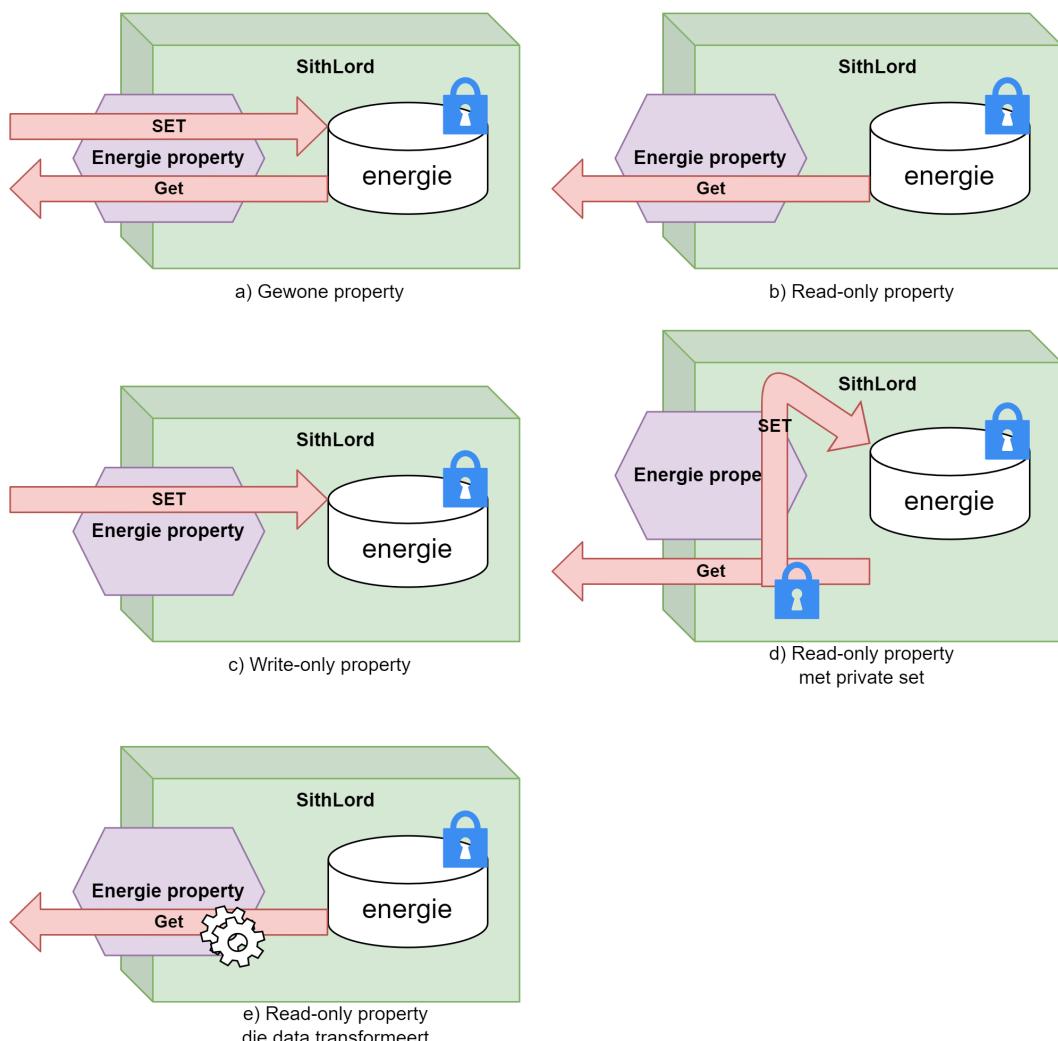


Probeer wel steeds de OOP-principes te hanteren wanneer je met properties werkt: in de **get** en **set** van een property mogen enkel die dingen gebeuren die de verantwoordelijkheid van de property zelf zijn. Je gaat dus bijvoorbeeld niet controleren of een andere property geen illegale waarden krijgt, daar is die andere property voor verantwoordelijk.

9.4.5 Property variaties

We zijn niet verplicht om zowel de **get** en de **set** code van een property te schrijven. Dit laat ons toe om een aantal variaties te schrijven:

- **Write-only property:** heeft geen **get**.
- **Read-only property:** heeft geen **set**.
- **Read-only property met private set :** het omgekeerde, een private **get**, zal je zelden tegenkomen.
- **Read-only property die data transformeert:** om interne data in een andere vorm uit je object te krijgen.



Figuur 9.2: De verschillende full properties mooi opgelist.

9.4.5.1 Write-only property

Dit soort properties zijn handig indien je informatie naar een object wenst te sturen dat niet mag of moet uitgelezen kunnen worden. Het meest typische voorbeeld is een property Pincode van een klasse BankRekening.

```
1 public int Energie
2 {
3     set
4     {
5         if(value >= 0)
6             energie = value;
7     }
8 }
```

We kunnen dus enkel energie een waarde geven, maar niet van buiten uitlezen.

9.4.5.2 Read-only property

Letterlijk het omgekeerde van een write-only property. Deze gebruik je vaak wanneer je informatie uit een object wil kunnen uitlezen uit een instantievariabele dat NIET door de buitenwereld mag aangepast worden.

```
1 public int Energie
2 {
3     get
4     {
5         return energie;
6     }
7 }
```

We kunnen enkel energie van buiten uitlezen, maar niet aanpassen.



Het **readonly** keyword heeft andere doelen en wordt NIET gebruikt in C# om een readonly property te maken.

9.4.5.3 Read-only property met private set

Soms gebeurt het dat we van enkel voor de buitenwereld de property read-only willen maken. We willen in de klasse zelf nog steeds controleren dat er geen illegale waarden aan private instantievariabelen worden gegeven. Op dat moment definiëren we een read-only property met een private setter:

```
1 public int Energie
2 {
3     get
4     {
5         return energie;
6     }
7     private set
8     {
9         if(value >= 0)
10            energie = value;
11    }
12 }
```

Van buiten zal enkel code werken die de **get** van deze property aanroeft, bijvoorbeeld:

```
1 Console.WriteLine(palpatine.Energie);
```

Code die de **set** van buiten nodig heeft (bv. `palpatine.Energie = 65;`) zal een fout geven ongeacht of deze geldig is of niet.



Het is een goede gewoonte om **altijd** via de properties je interne variabele aan te passen en niet rechtstreeks via de instantievariabele zelf. Dit is zo'n nuttige tip dat we op de volgende pagina de voorman hier ook nog even over aan het woord gaan laten.



Lukt het een beetje? Properties zijn in het begin wat overweldigend, maar geloof me: ze zijn zowat dé belangrijkste bewoners in de .NET/C# wereld.

Nu even goed opletten: indien we **in** het object de instantievariabelen willen aanpassen dan is het een goede gewoonte om ook dat **via de property** te doen (ook al zit je in het object zelf en heb dus eigenlijk de property niet nodig). Zo zorgen we ervoor dat de bestaande controle in de property niet wordt omzeilt. Kijk zelf naar volgende **slechte** code voorbeeld:

```

1 internal class SithLord
2 {
3     private int energie;
4     private string sithName;
5     public void ResetLord(int resetWaarde)
6     {
7         energie = resetWaarde;
8     }
9     public int Energie
10    {
11        get
12        {
13            return energie;
14        }
15        private set
16        {
17            if(value >= 0)
18                energie = value;
19        }
20    }
21 }
```

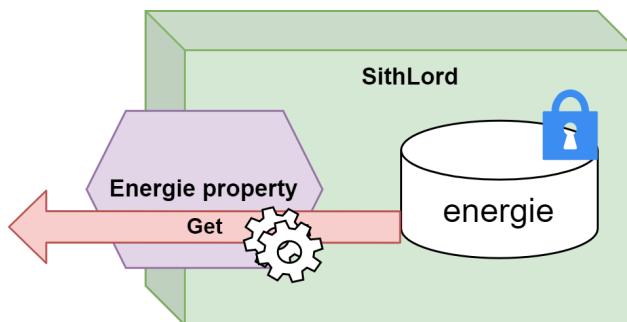
De nieuw toegevoegde methode `ResetLord` willen we gebruiken om de lord z'n energie terug te verlagen. Als we deze methode met een negatieve waarden aanroepen zullen we alsnog energie op een verkeerde waarde instellen. Nochtans is dit een illegale waarde volgens de set-code van de property.

We moeten dus in de methode ook expliciet via de property gaan om bugs te voorkomen en dus gaan we in `ResetLord` schrijven naar de property `Energie` én niet rechtstreeks naar de instantievariabele `energie`:

```

1 public void ResetLord(int resetWaarde)
2 {
3     Energie = resetWaarde; // Energie i.p.v. energie
4 }
```

9.4.5.4 Read-only properties die transformeren



Figuur 9.3: Transformerende properties: Erg nuttig, maar vaak wat stiefmoederlijk behandeld.

Je bent uiteraard niet verplicht om voor iedere instantievariabele een bijhorende property te schrijven. Omgekeerd ook: mogelijk wil je extra properties hebben voor data die je 'on-the-fly' kan genereren dat niet noodzakelijk uit een instantievariabele komt. Stel dat we volgende klasse hebben:

```

1 internal class Persoon
2 {
3     public string Voornaam {get; set;}
4     public string Achternaam {get; set;}
5 }
```

We willen echter ook soms de volledige naam of emailadres krijgen, beide gebaseerd op de inhoud van de instantievariabelen voornaam en achternaam. Via een read-only property die transformeert kan dit:

```

1 internal class Persoon
2 {
3     public string Voornaam {get; set;}
4     public string Achternaam {get; set;}
5     public string VolledigeNaam
6     {
7         get
8         {
9             return $"{Voornaam} {Achternaam}";
10        }
11    }
12    public string Email
13    {
14        get
15        {
16            return $"{Voornaam}@ziescherp.be";
17        }
18    }
19 }
```



Methode of property?

Een veel gestelde vraag bij beginnende OOP-ontwikkelaars is: “*Moet dit in een property of in een methode geplaatst worden?*”

De regels zijn niet in steen gebeiteld, maar ruwweg kan je stellen dat:

- Betreft het een actie of gedrag: iets dat het object moet doen (tekst tonen, iets berekenen of aanpassen, enz.) dan plaats je het in een **methode**.
- Betreft het een eigenschap van het object, dan gebruik je een **property** indien het om data gaat die snel verkregen of berekend kan worden. Gaat het om data die zwaardere en/of langere berekeningen vereist dan is een methode nog steeds aangeraden.

9.4.6 Auto-properties

Automatische eigenschappen (**automatic properties**, *auto-implemented properties*, soms ook *autoprops* genoemd) laten toe om snel properties te schrijven zonder dat we de achterliggende instantievariabele moeten beschrijven.

Een auto-property herken je aan het feit dat ze een pak koper zijn qua code, omdat er veel meer (onzichtbaar) achter de schermen wordt opgelost:

```
1 public string Voornaam { get; set; }
```

Heel vaak wil je heel eenvoudige variabelen aan de buitenwereld van je klasse beschikbaar stellen. Omdat je instantievariabelen echter niet **public** mag maken, moeten we dus properties gebruiken die niets anders doen dan als doorgewassen fungeren. auto-properties doen dit voor ons: het zijn vereenvoudigde full properties waarbij de achterliggende instantievariabele onzichtbaar voor ons is. Je kan echter bij auto-properties ook geen verdere controle op de in-of uitvoer doen.

Zo kan je eenvoudig de volgende klasse Persoon herschrijven met behulp van auto-properties. De originele klasse mét full properties:

```
1 internal class Person
2 {
3     private string voornaam;
4     public string Voornaam
5     {
6         get { return voornaam; }
7         set { voornaam = value; }
8     }
9
10    private int geboorteJaar;
11    public int Geboortejaar
12    {
13        get { return geboorteJaar; }
14        set { geboorteJaar = value; }
15    }
16 }
```

De herschreven klasse met auto-properties wordt:

```
1 internal class Person
2 {
3     public string Voornaam { get; set; }
4     public int Geboortejaar { get; set; }
5 }
```

Beide klassen hebben exact dezelfde functionaliteit, echter is de laatste klasse aanzienlijk koper en dus eenvoudiger om te lezen. **De private instantievariabelen zijn niet meer aanwezig.** C# gaat die voor z'n rekening nemen. Alle code zal dus via de properties moeten gaan.

Het is belangrijk te benadrukken dat de achterliggende instantievariabele onzichtbaar is in auto-properties en onmogelijk kan gebruikt worden. Alles gebeurt via de auto-property, altijd. Je hebt dus enkel een soort publieke variabele. Maar wel eentje die conform de afspraken is (“maak geen instantievariabelen publiek!”). Gebruik dit dus enkel wanneer je 100% zeker bent dat de auto-property geen waarden kan krijgen die de interne werking van je klasse kan verstoren.



Vaak zal je nieuwe klassen eerst met auto-properties beschrijven. Naarmate de specificaties dan vereisen dat er bepaalde controles of transformaties moeten gebeuren, zal je stelselmatig auto-properties vervangen door full properties.

Dit kan trouwens automatisch in VS: selecteer de autoprop in kwestie en klik dan vooraan op de schroevendraaier en kies “Convert to full property”.

Opgelet: Merk op dat de syntax die VS gebruikt om een full property te schrijven anders is dan wat ik hier uitleg. Wanneer je VS laat doen krijg je een oplossing met allerlei => tekens. Dit is heet **Expression Bodied Member syntax (EBM)**. Ik behandel deze nieuwere C# syntax in de appendix.

9.4.7 Nut auto-properties?

Merk op dat je auto-properties dus enkel kan gebruiken indien er geen extra logica in de property (bij de **set** of **get**) aanwezig moet zijn.

Stel dat je bij de setter van geboorteJaar wil controleren op een negatieve waarde, dan zal je dit zoals voorheen moeten schrijven en kan dit niet met een automatic property:

```
1 set
2 {
3     if( value > 0)
4         geboorteJaar = value;
5 }
```

Voorgaande property kan dus NIET herschreven worden met een automatic property. auto-properties zijn vooral handig om snel klassen in elkaar te knutselen, zonder je zorgen te moeten maken om andere vereisten. Vaak zal een klasse in het begin met auto-properties gevuld worden. Naarmate je project vordert zullen die auto-properties meer en meer omgezet worden in full properties.

9.4.8 Beginwaarden van auto-properties

Je mag auto-properties beginwaarden geven door de waarde achter de property te schrijven, als volgt:

```
1 public int Geboortejaar {get;set;} = 2002;
```

Al je objecten zullen nu als geboortejaar 2002 hebben wanneer ze geïnstantieerd worden.

9.4.9 Alleen-lezen auto-properties

Je kan auto-properties ook gebruiken om bijvoorbeeld een read-only property met private setter te definiëren. Als volgt:

```
1 public string Voornaam { get; private set; }
```

Een andere manier die ook kan wanneer we enkel een read-only property nodig hebben, is als volgt:

```
1 public string Voornaam { get; } = "Tim";
```

Hierbij zijn we dan wel verplicht om ogenblikkelijk deze property een beginwaarde te geven, daar we deze op geen enkele andere manier nog kunnen aanpassen.



Als je in Visual Studio in je code prop typt en vervolgens twee keer de tabtoets indrukt dan verschijnt al de nodige code voor een automatic property. Via propg gevuld door twee maal de tabtoets krijg je een auto-property met private setter.

9.5 OOP in de praktijk : DateTime



Doe die zwembroek maar weer aan! We gaan nog eens zwemmen.

Zoals je vermoedelijk al doorhebt hebben we met properties en methoden nog maar een tipje van de *klasse-ijsberg* besproken. Vreemde dingen zoals *constructors*, *static methoden*, *overerving* en arrays van objecten staan ons nog allemaal te wachten.

Om je toch al een voorsmaakje van de kracht van klassen en objecten te geven, gaan we eens kijken naar één van de vele klassen die je tot je beschikking hebt in C#. Je hebt al leren werken met de Random klasse. Maar ook al met enkele speciale *static klassen* zoals de Math- en Console-bibliotheek. Waarom zijn dit *static klassen*? Wel je kan gebruiken ze **zonder** dat je er objecten van moet aanmaken.

Nog zo'n handige ingebouwde klasse is de DateTime klasse³. Je raadt het nooit ... een klasse die toelaat om de tijd en datum als een object voor te stellen. De DateTime klasse is de ideale manier om te leren werken met objecten.

³Technisch gezien is DateTime een **struct**, niet een **class**. Dit onderscheid is in dit handboek niet relevant. Meer informatie over **struct** vind je in de appendix terug.

9.5.1 DateTime objecten aanmaken

Er zijn 2 manieren om DateTime objecten aan te maken:

```
1 DateTime huidigeTijd = DateTime.Now;  
2 DateTime specialeDag = new DateTime(2017,4,21);
```

- Lijn 1: Door aan de klasse de huidige datum en tijd te vragen via `DateTime .Now`.
- Lijn 2: Door manueel de datum en tijd in te stellen met het `new` keyword en de **klasse-constructor** (een concept dat we in hoofdstuk 11 uit de doeken gaan doen)

9.5.1.1 DateTime.Now

Volgend voorbeeld toont hoe we een object kunnen maken dat de huidige datum tijd van het systeem bevat. Vervolgens printen we dit op het scherm:

```
1 DateTime currentTime = DateTime.Now;  
2 Console.WriteLine(currentTime);
```



`DateTime .Now` is een zogenaamde **static property** wat verderop in het boek zal uitgelegd worden.

9.5.1.2 Met constructor en new

De constructor van een klasse laat toe om bij het maken van een nieuw object, beginwaarden voor bepaalde instantievariabelen of properties mee te geven. De `DateTime` klasse heeft meerdere constructors gedefinieerd zodat je bijvoorbeeld een object kan aanmaken dat bij de start reeds de geboortedatum van de auteur bevat:

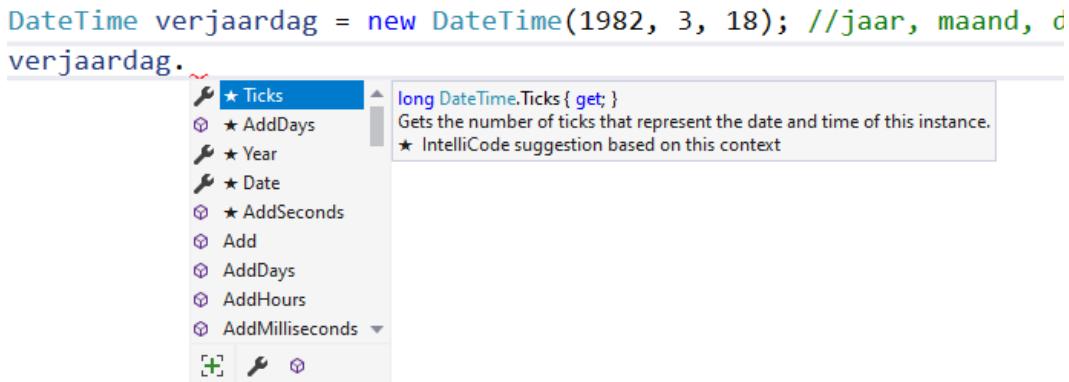
```
1 DateTime verjaardag = new DateTime(1981, 3, 18); //jaar, maand, dag
```

Ook is er een constructor om startdatum én -tijd mee te geven bij de objectcreatie:

```
1 //Volgorde: jaar, maand, dag, uur, minuten, seconden  
2 DateTime trouwMoment = new DateTime(2017, 4, 21, 10, 00,34 );
```

9.5.2 DateTime methoden

Van zodra je een DateTime object hebt gemaakt zijn er tal van nuttige methoden die je er op kan aanroepen. Visual Studio is zo vriendelijk om dit te visualiseren wanneer we de dot-operator typen achter een object:



Figuur 9.4: Iedere kubus stelt een methode voor. Iedere Engelse sleutel een property.

9.5.2.1 Add-methoden

Deze methoden kan je gebruiken om een bepaalde aantal dagen, uren, minuten op te tellen bij de huidige tijd en datum van een object. De ingebouwde methoden noemen allemaal AddX, waarbij X dan vervangen wordt door het soort element dat je wilt toevoegen: AddDays, AddHours, AddMilliseconds, AddMinutes, AddMonths, AddSeconds, AddTicks⁴, AddYears.

Het object zal voor ons de “berekening” hiervan doen en vervolgens een **nieuw DateTime object** teruggeven dat je moet bewaren wil je er iets mee doen.

In volgende voorbeeld wil ik ontdekken wanneer de wittebroodsweken van m’n huwelijk eindigen (pakweg 5 weken na de trouwdag).

```
1 DateTime eindeWitteBroodsweken = trouwMoment.AddDays(35);
2 Console.WriteLine(eindeWitteBroodsweken);
```

⁴Een *tick* is 100 nanoseconden, oftewel 1 tien miljoenste van een seconde. Dat lijkt een erg klein getal (wat het voor ons ook is) maar voor computers is dit het soort tijdsintervallen waar ze mee werken.

9.5.3 DateTime properties

Dit hoofdstuk heeft al aardig wat woorden verspild aan properties, en uiteraard heeft ook de DateTime klasse een hele hoop interessante properties die toelaten om de interne staat van een DateTime object te bewerken of uit te lezen.

Enkele nuttige properties van DateTime zijn: Date, Day, DayOfWeek, DayOfYear, Hour, Millisecond, Minute, Month, Second, Ticks, TimeOfDay, Today, UtcNow, Year.

Alle properties van DateTime zijn read-only en hebben dus een private setter die we niet kunnen gebruiken.

Een voorbeeld:

```
1 Console.WriteLine($"Einde in maand nr: {eindeWitteBroodsweken.Month}.");
  ");
2 Console.WriteLine($"Dat is een {eindeWitteBroodsweken.DayOfWeek}.");
```

Dit geeft op het scherm:

```
1 Je wittebroodsweken eindigen in maand nummer: 5.
2 Dat is een Friday.
```

9.5.4 Static methoden

Sommige methoden zijn **static** dat wil zeggen dat je ze enkel rechtstreeks op de klasse kunt aanroepen. Vaak zijn deze methoden hulpmethoden waar de individuele objecten niets aan hebben. We hebben dit reeds gebruikt bij de Math en Console-klassen.

We behandelen **static** uitgebreid verderop in het boek.

9.5.4.1 De tijd uit een string inlezen

Parsen laat toe dat je strings omzet naar een DateTime object. Dit is handig als je bijvoorbeeld de gebruiker via Console.ReadLine() tijd en datum wilt laten invoeren in de *Belgische notatie*:

```
1 string datumInvoer = Console.ReadLine();
2 DateTime datumVerwerkt = DateTime.Parse(datumInvoer, new System.
  Globalization.CultureInfo("nl-BE"));
3 Console.WriteLine(datumVerwerkt);
```

Indien je nu dit programma'tje zou uitvoeren en als gebruiker "8/11/2016" zou intypen, dan zal deze datum geparsed worden en in het object datumVerwerkt komen.



Zoals je ziet roepen we Parse aan op DateTime en dus niet op een specifiek object. Dat was ook zo reeds bijvoorbeeld bij **int**. Parse wat dus doet vermoeden dat zelfs het **int** datatype eigenlijk een klasse is!

9.5.4.2 IsLeapYear

Deze nuttige methode geeft een **bool** terug om aan te geven of de actuele parameter (type **int**) een schrikkeljaar voorstelt of niet:

```
1 DateTime vandaag = DateTime.Now;
2 if(DateTime.IsLeapYear(vandaag.Year))
3     Console.WriteLine("Dit jaar is een schrikkeljaar.");
```

9.5.5 TimeSpan

Je kan DateTime objecten ook van elkaar aftrekken (optellen gaat niet!). Het resultaat van deze bewerking geeft echter niet een DateTime object terug, **maar een TimeSpan object**. Dit is nieuwe object van het type TimeSpan (wat dus een andere klasse is) dat aangeeft hoe groot het verschil is tussen de 2 DateTime objecten kunnen we als volgt gebruiken:

```
1 DateTime vandaag = DateTime.Today;
2 DateTime geboorteDochter = new DateTime(2009, 6, 17);
3 TimeSpan verschil = vandaag - geboorteDochter;
4 Console.WriteLine($"{verschil.TotalDays} dagen sinds geboorte dochter
    .");
```

Je zal de DateTime klasse in véél van je projecten kunnen gebruiken waar je iets met tijd, tijdsverschillen of datums wilt doen. We hebben de klasse in deze sectie echter geen eer aangedaan. De klasse is veel krachtiger dan ik hier doe uitschijnen. Het is een goede gewoonte als beginnende programmeur om steeds de documentatie van nieuwe klassen er op na te slaan.⁵

⁵Wanneer je in je browser zoekt op “C#” gevolgd door de naam van de klasse dan zal je zo goed als zeker als eerste *hit* de officiële .NET documentatie krijgen op docs.microsoft.com.



Gaat het nog?! Dit was een stevig hoofdstuk he. We hebben zo maar eventjes 4 heel grote fasen doorlopen:

1. *Eerst keken we hoe OOP ons kan helpen in een real-life voorbeeld, Pong. We schreven code die hier en daar herkenbaar was, maar op andere plaatsen totaal nieuw was.*
2. *Vervolgens namen we de mammoet bij de horen en bekijken we de theorie van OO, die ons vooral verwarde.*
3. *Gelukkig gingen we dan ogenblikkelijk naar de praktijk over en zagen we dat methoden en properties de kern van iedere klasse blijkt te zijn.*
4. *Als afsluiter gooiden we dan de DateTime klasse open om een voorproefje te krijgen van hoe krachtig een goedgeschreven klasse kan zijn.*

Voor je verder gaat raad ik je aan om dit alles goed te laten bezinken én maximaal de komende oefeningen te maken. Het zal de beste manier zijn om de ietwat bizarre wereld van OOP snel eigen te maken.

10 Geheugen- en codebeheer

Dit hoofdstuk gaat een beetje overal over. In de eerste, en belangrijkste, plaats gaan we eens kijken wat er allemaal achter de schermen gebeurt wanneer we met objecten programmeren. Je zal namelijk ontdekken dat er een fundamenteel verschil is in het werken met bijvoorbeeld een object van het type `Student` tegenover werken met een eenvoudige variabele van het type `int`.

Vervolgens gaan we kort de keywords `using` en `namespace` bekijken. Die tweede heb je al bij iedere project bovenaan je code zien staan, nu wordt het tijd om toe te lichten waarom dat is.

Finaal lijkt het ons een goed moment om je robuustere, minder crashende, code te leren schrijven. Exception handling, de naam zegt het al, gaat ons helpen om die typische uitzonderingen (zoals deling door 0) in algoritmes op een elegante manier op te vangen (en dus niet door een nest van `if` structuren te schrijven in de hoop dat je iedere mogelijke uitzondering kunt opvangen).

10.1 Geheugenbeheer in C#

In hoofdstuk 8 deed ik reeds uit de doeken dat variabelen op 2 manieren in het geheugen kunnen leven:

- **Value types:** waren variabelen wiens waarde rechtstreeks op de geheugenplek stonden waar de variabele naar verwees. Dit gold voor alle bestaande, ingebakken datatypes zoals `int`, `bool`, `char` enz. alsook voor `enum` types.
- **Reference types:** deze variabelen bevatten als inhoud een geheugenadres naar een andere plek in het geheugen waar de effectieve waarde van deze variabele stond. We zagen dat dit voorlopig enkel bij arrays gebeurde.

Ook objecten zijn reference types. Hoofdstuk 8 liet uitschijnen dat vooral value type variabelen veelvuldig in programma's voorkwamen. Wel je zal nu ontdekken dat reference types véél meer voorkomen: **simpelweg omdat alles in C# een object is** (en dus ook arrays van objecten én zelfs valuetypes, enz.).

Om goed te begrijpen waarom reference types zo belangrijk zijn, zullen we nu eerst eens inzoomen op hoe het geheugen van een C# applicatie werkt.

10.1.1 Stack en heap

In hoofdstuk 8 toonde ik ook hoe alle variabelen in één grote “wolk geheugen” zitten, ongeacht of ze nu value types of reference types zijn. Dat klopt niet helemaal. Eigenlijk zijn er **2 soorten geheugens** die een C# applicatie tot z’n beschikking heeft.

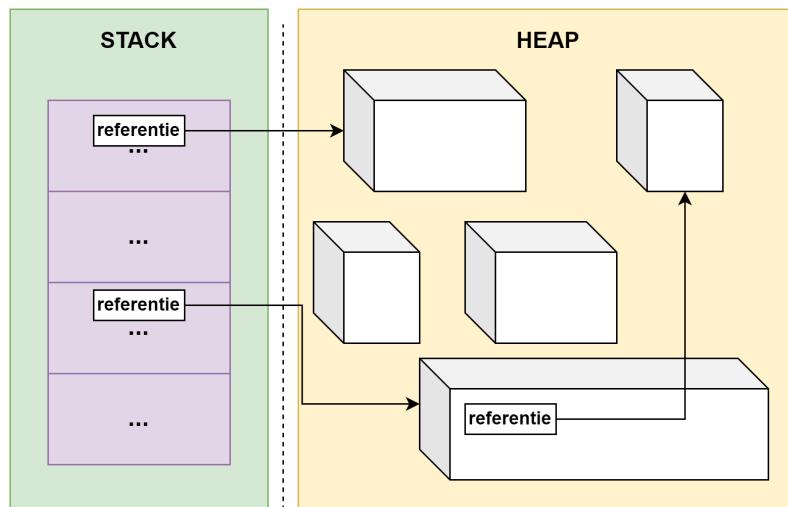
Wanneer een C# applicatie wordt uitgevoerd krijgt het twee soorten geheugen toegewezen dat het ‘naar hartelust’ kan gebruiken, namelijk:

1. Het kleine, maar snelle **stack** geheugen.
2. Het grote, maar tragere **heap** geheugen.

Afhankelijk van het soort variabele wordt ofwel de stack, ofwel de heap gebruikt. **Het is uitermate belangrijk dat je weet in welk geheugen de variabele zal bewaard worden!** Je hebt hier geen controle over, maar het beïnvloedt wel de manier waarop je code zal werken.

Volgende tabel vat samen welke type in welk geheugen wordt bewaard:

	Value types	Reference types
Inhoud	Eigenlijke data	Referentie naar de eigenlijke data
Locatie	Stack	Heap
Beginwaarde	0, 0.0, "", false , enz.	null
Effect = operator	Kopieert actuele waarde	Kopieert adres naar actuele waarde



Figuur 10.1: Stack en heap

10.1.2 Waarom twee geheugens?

Waarom plaatsen we niet alles in de stack? De reden hiervoor is dat bij het compileren van je applicatie er reeds zal berekend worden hoeveel geheugen de stack zal nodig hebben. Wanneer je programma dus later wordt uitgevoerd weet het OS perfect hoeveel geheugen het minstens moet reserveren bij het besturingssysteem.

Er is echter een probleem: de compiler kan niet alles perfect berekenen of voorspellen. Van een variabele van het type **int** is perfect geweten hoe groot die zal zijn (32 bit). Maar wat met een **string** die je aan de gebruiker vraagt? Of wat met een array waarvan we pas tijdens de uitvoer de lengte gaan berekenen gebaseerd op *runtime* informatie?

Het zou nutteloos (en zonde) zijn om reeds bij aanvang een bepaalde hoeveelheid stackgeheugen voor een array te reserveren als we niet weten hoe groot die zal worden. Beeld je in dat alle applicaties op je computer voor *alle* zekerheid een halve gigabyte aan geheugen zouden vragen. Je computer zou enkele terabyte aan geheugen nodig hebben. Het is dus veel realistischer om enkel het geheugen te reserveren waar de compiler 100% zeker van is dat deze zal nodig zijn.

De heap laat ons toe om geheugen op een wat minder gestructureerde manier in te palmen. Tijdens de uitvoer van het programma fungeert de heap als een grote speelplaats waar je overal dingen kunt neerzetten, zolang er maar ruimte vrij is. De stack daarentegen is het kleine bankje naast de zandbak: handig, snel, en met een precies bekende grootte.

10.1.3 Value types in de stack

Value type variabelen worden in de stack bewaard. **De effectieve waarde van de variabele wordt in de stack bewaard.** Dit zijn alle gekende, ‘eenvoudige’ datatypes die we tot nog toe gezien hebben:

- **sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool.**
- **struct** (niet besproken in dit boek, maar wel kort toegelicht in de appendix).
- enums (zie hoofdstuk 5).

10.1.4 = operator bij value types

Wanneer we een value-type willen kopiëren gebruiken we de `=`-operator die de waarde van de rechtse operand zal uitlezen en zal kopiëren naar de linkse operand:

```
1 int getal = 3;  
2 int anderGetal = getal;
```

Vanaf nu zal `anderGetal` de waarde 3 hebben. Als we nu één van beide variabelen aanpassen dan zal dit **een** effect hebben op de andere variabelen.

We zien hetzelfde effect wanneer we een methode maken die een parameter van het value type aanvaardt:

```
1 void VerhoogParameter(int a)
2 {
3     a++;
4     Console.WriteLine($"In methode {a}");
5 }
```

Bij de aanroep geven we een kopie van de variabele mee:

```
1 int getal = 5;
2 VerhoogParameter(getal);
3 Console.WriteLine($"Na methode {getal}");
```

De parameter `a` zal de waarde 5 gekopieerd krijgen. Maar wanneer we nu zaken aanpassen in `a` zal dit geen effect hebben op de waarde van `getal`.

De output van bovenstaand programma zal zijn:

```
1 In methode 6
2 Na methode 5
```

10.1.5 Reference types

Reference types worden in de heap bewaard. De *effectieve waarde* wordt in de heap bewaard, en in de stack zal enkel een **referentie** of **pointer** naar de data in de heap bewaard worden. Een referentie is niet meer dan het geheugenadres naar waar verwezen wordt (bv. 0xA3B3163).

Concreet zijn dit alle zaken die vaak redelijk groot zullen zijn of waarvan op voorhand niet kan voorspeld worden hoe groot ze *at runtime* zullen zijn (denk maar aan arrays, instanties van complexe klassen, enz.).

10.1.6 = operator bij reference types

Wanneer we de `=` operator gebruiken bij een reference type dan kopiëren we de referentie naar de waarde van de rechtse operand, niet de waarde zelf.

We bekijken nu de impact van de `=` operator bij:

- objecten: en zullen zien dat we dus de referentie, niet het object zelf, kopiëren.
- arrays: en zullen ook hier zien dat we niet de array zelf kopiëren, maar enkel de referentie naar de array in de heap.

10.1.6.1 = operator bij objecten

We zien dit gedrag bij alle reference types, zoals objecten:

```
1 Student stud = new Student();
```

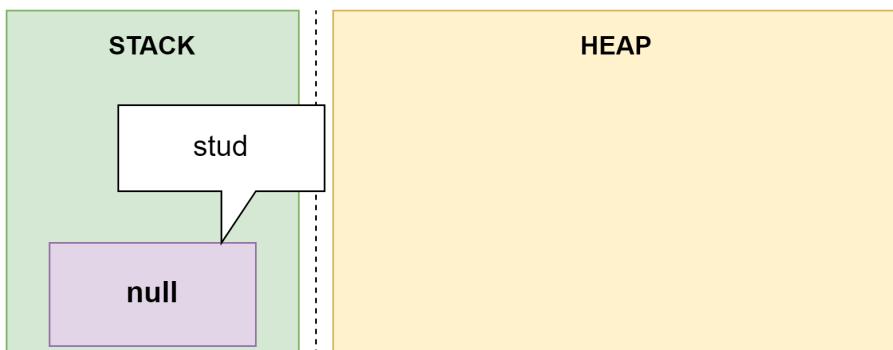
Wat gebeurt er hier?

1. **new Student()**: **new** roept de constructor van **Student** aan. Deze zal met behulp van een constructor een object in de **heap** aanmaken en vervolgens de geheugenlocatie ervan teruggeven.
2. Een variabele **stud** wordt in de **stack** aangemaakt en mag enkel een referentie naar een object van het type **Student** bewaren.
3. De geheugenlocatie uit de eerste stap wordt vervolgens in **stud** opgeslagen in de stack.

Laten we eens inzoomen op voorgaande door de code even in 2 delen op te splitsen:

```
1 Student stud;
2 stud = new Student();
```

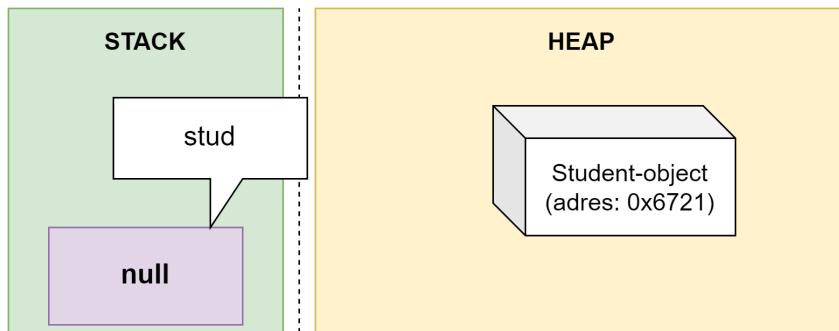
Het geheugen na lijn 1 ziet er zo uit:



Figuur 10.2: Na lijn 1

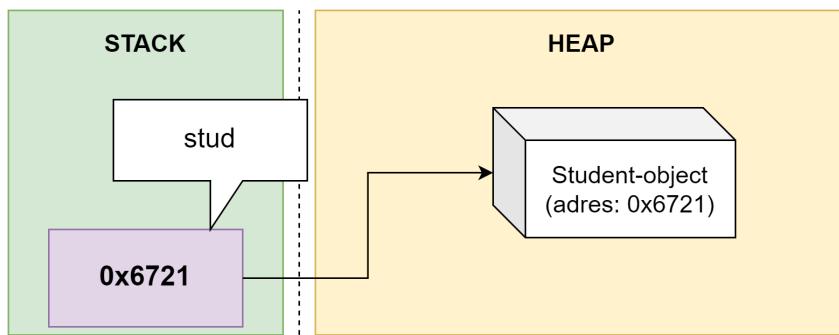
Merk op dat de variabele **stud** eigenlijk de waarde **null** heeft. We leggen later uit wat dit juist wil zeggen.

Lijn 2 gaan we nog trager bekijken: Eerst zal het gedeelte rechts van de `=`-operator uitgevoerd worden. Er wordt dus **in de heap** een nieuw Student-object aangemaakt:



Figuur 10.3: Na: new Student();

Vervolgens wordt de toekenning toegepast en wordt het geheugenadres van het object in de variabele `stud` geplaatst:



Figuur 10.4: Na: stud = new Student();



Ik ga nogal licht over het `new`-keyword en de constructor. Maar zoals je merkt is dit een ongelooflijk belangrijk mechanisme in de wereld van de objecten. Het brengt letterlijk objecten tot leven (in de heap) en zal als resultaat laten weten op welke plek in het geheugen het object staat.

10.1.6.2 = operator bij arrays

Zoals we in hoofdstuk 8 hebben gezien, zien we hetzelfde gedrag bij arrays:

```
1 int[] nummers = {4,5,10};
2 int[] andereNummers = nummers;
```

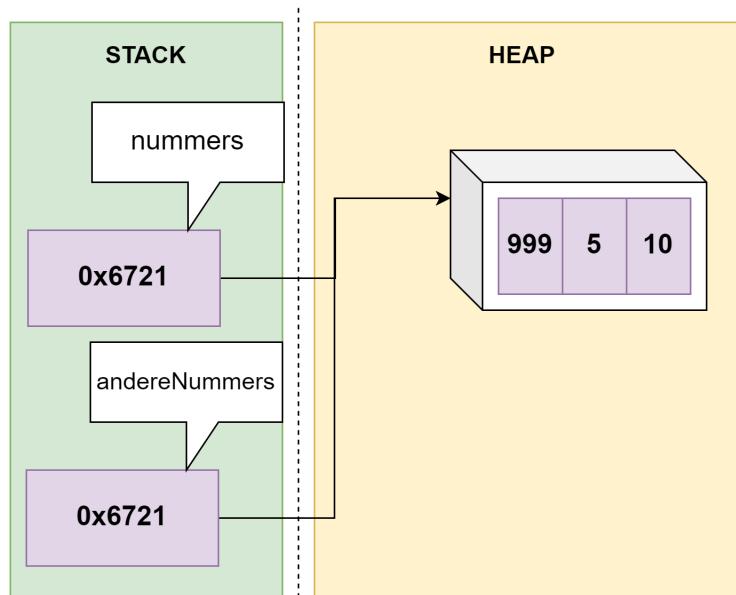
In dit voorbeeld zal `andereNummers` nu dus ook verwijzen naar de array in de heap waar de actuele waarden staan.

Als we dus volgende code uitvoeren dan ontdekken we dat beide variabelen naar dezelfde array verwijzen:

```
1 andereNummers[0] = 999;
2 Console.WriteLine(andereNummers[0]);
3 Console.WriteLine(numbers[0]);
```

We zullen dus als output krijgen:

```
1 999
2 999
```

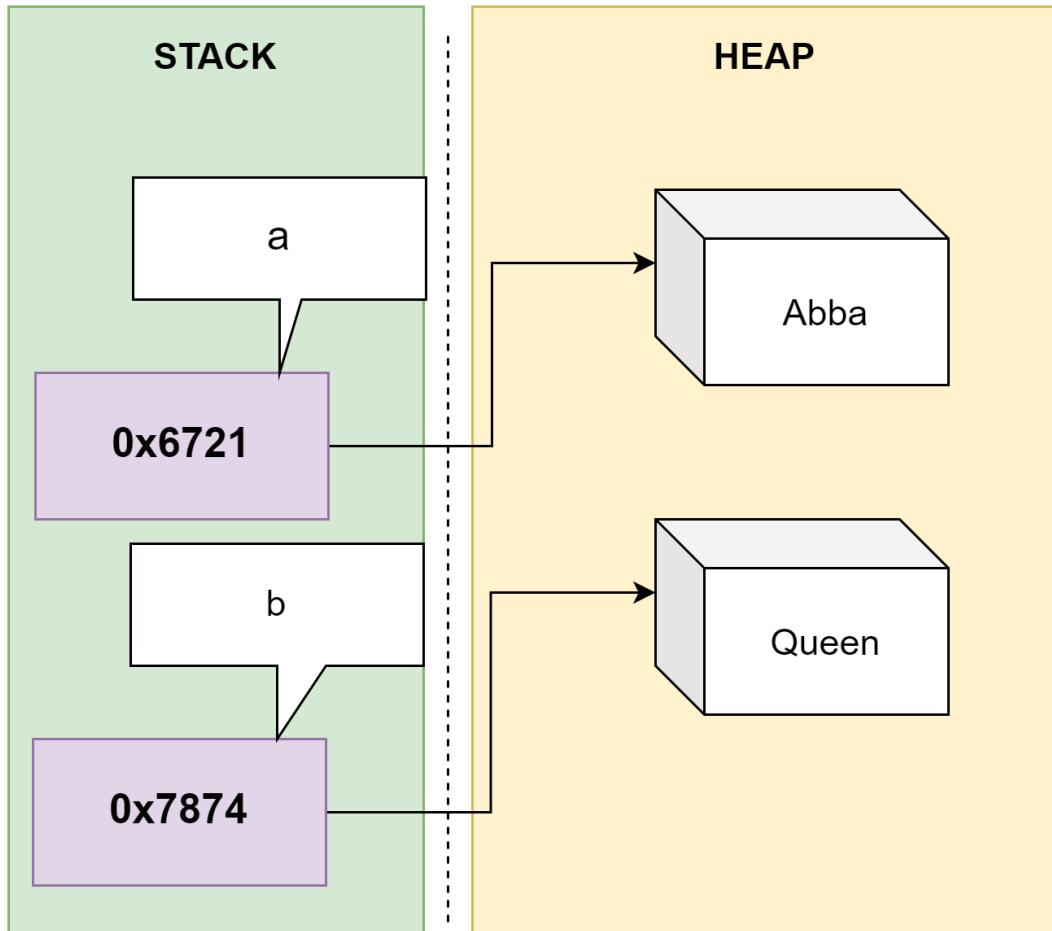


Figuur 10.5: De situatie op het einde.

Hetzelfde gedrag zien we bij objecten:

```
1 Student a = new Student("Abba");  
2 Student b = new Student("Queen");
```

Geeft volgende situatie in het geheugen:

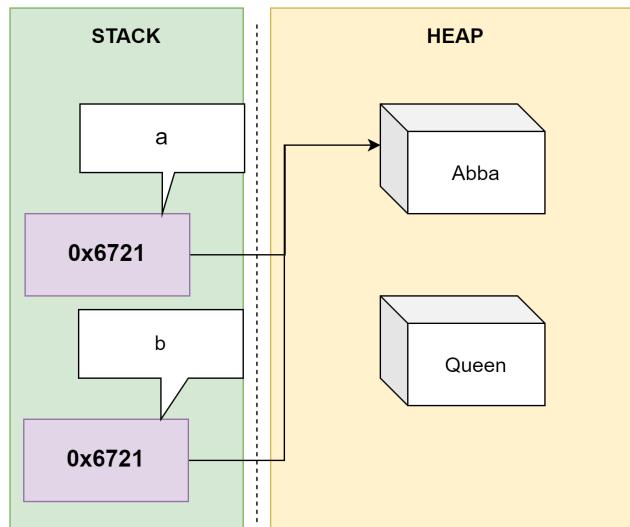


Figuur 10.6: Merk op dat de geheugenplekken willekeurig zijn. De auteur was te lui om telkens nieuwe geheugenplekken te verzinnen, daarom dat je sommige getallen ziet terugkomen.

Schrijven we dan het volgende:

```
1 b = a;
2 Console.WriteLine(a.Naam);
```

Dan zullen we in dit geval dus Abba op het scherm zien omdat zowel b als a naar hetzelfde object in de heap verwijzen. Het originele “Queen”-object zijn we kwijt en zal verdwijnen (zie Garbage Collector verderop).



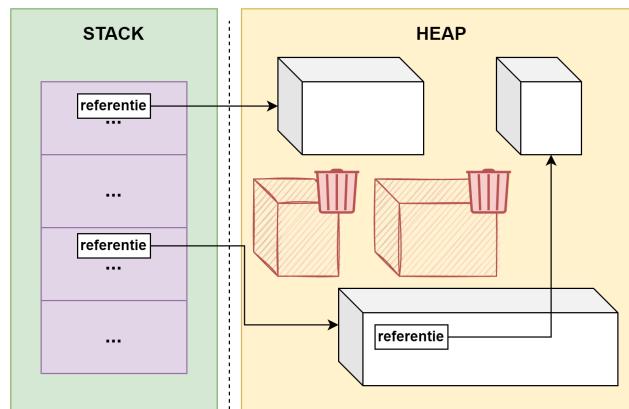
Figuur 10.7: Het is een goede gewoonte om dit soort tekeningen met pijlen steeds mentaal (of op papier) te maken wanneer je werken met referenties onder de knie wilt krijgen.



De meeste klassen zullen met value type-properties en instantievariabelen werken in zich, toch worden deze ook samen met het gehele object in de heap bewaard en niet in de stack. Kortom **het hele object** ongeacht de vorm (*datatypes*) van z'n inhoud wordt in de heap bewaard.

10.1.7 De Garbage Collector

Een stille held van .NET is de zogenaamde GC, de **Garbage Collector**. Dit is een geautomatiseerd onderdeel van ieder C# programma dat ervoor zorgt dat we geen geheugen noodloos gereserveerd houden. De GC zal geregeld het geheugen doorlopen en kijken of er in de heap objecten staan waar geen referenties naar verwijzen. Indien er geen referenties naar wijzen zal dit object verwijderd worden.



Figuur 10.8: Objecten in de heap waar geen referenties naar wijzen zullen ten gepaste tijde verwijderd worden.

In dit voorbeeld zien we dit in actie:

```

1 Held supermand = new Held();
2 Held batmand = new Held();
3 batmand = supermand;

```

Vanaf de laatste lijn zal er geen referentie meer naar het originele object zijn waar batmand naar verwees in de heap, daar we deze hebben overschreven met een referentie naar het eerste Held object in supermand. De GC zal dus dat tweede aangemaakte Held object verwijderen. Wil je dat niet dan zal je minstens 1 variabele moeten hebben die naar de data verwijst. Volgend voorbeeld toont dit:

```

1 Held supermand = new Held();
2 Held batmand = new Held();
3 Held bewaarEersteHeld = batmand;
4 batmand = supermand;

```

De variabele bewaarEersteHeld houdt dus een referentie naar die in batmand bij en we kunnen dus later via deze variabele alsnog aan de originele data.



De GC werkt niet continue daar dit te veel overhead van je computer zou vereisen. De GC zal gewoon om de zoveel tijd alle gereserveerde geheugenplekken van de applicatie controleren en die delen verwijderen die niet meer nodig zijn. Je kan de GC manueel de opdracht geven om een opkuisbeurt te starten met `GC.Collect()` maar dit is ten stelligste af te raden! De GC weet meestal beter dan ons wanneer er gekuist moet worden.

10.2 Objecten en methoden

10.2.1 Objecten als actuele parameters

Klassen zijn “gewoon” nieuwe datatypes. Alle regels die we dus al kenden in verband met het doorgeven van variabelen als parameters in een methoden blijven gelden voor de meeste klassen (behalve **static** klassen die we in volgend hoofdstuk zullen aanpakken).

Het enige verschil is dat we objecten by reference meegeven aan methoden. Aanpassingen aan het object in de methode zal dus betekenen dat je het originele object aanpast dat aan de methode werd meegegeven, net zoals we bij arrays zagen. Hier moet je dus zeker rekening mee houden.

Stel dat we volgende klasse hebben waarin we temperatuurmetingen willen opslaan, alsook wie de meting heeft gedaan:

```
1 internal class Meting
2 {
3     public int Temperatuur { get; set; }
4     public string OpgemetenDoor { get; set; }
5 }
```

We voegen vervolgens een methode aan de klasse toe die ons toelaat om deze meting op het scherm te tonen in een bepaalde kleur.

```
1 public void ToonMettingInKleur (ConsoleColor kleur)
2 {
3     Console.ForegroundColor = kleur;
4     Console.WriteLine($"[Temperatuur] graden C gemeten door: [
5         OpgemetenDoor]");
6     Console.ResetColor();
7 }
```

Het gebruik van deze klasse zou er als volgt kunnen uitzien:

```
1 Meting m1 = new Meting();
2 m1.Temperatuur = 26;
3 m1.OpgemetenDoor = "Lieven Scheire";
4 Meting m2 = new Meting();
5 m2.Temperatuur = 34;
6 m2.OpgemetenDoor = "Ann Dooms";
7
8 m1.ToonMettingInKleur(ConsoleColor.Red);
9 m2.ToonMettingInKleur(ConsoleColor.Pink);
```

10.2.2 Objecten in methoden aanpassen

Je kan ook methoden schrijven die meegegeven objecten aanpassen daar we deze **by reference** doorsturen. Een voorbeeld waarin een meting als parameter meegeven en toevoegen aan een andere meting, waarna we de originele meting “resetten”:

```

1 public void VoegMettingToeEnVerwijder(Meting inMetting)
2 {
3     Temperatuur += inMetting.Temperatuur;
4     inMetting.Temperatuur = 0;
5     inMetting.OpgemetenDoor = "";
6 }
```

We zouden deze methode als volgt kunnen gebruiken (ervan uitgaande dat we 2 objecten m1 en m2 van het type Meting hebben):

```

1 m1.Temperatuur = 26;
2 m1.OpgemetenDoor = "Lieven Scheire";
3 m2.Temperatuur = 5;
4 m2.OpgemetenDoor = "Lieven Scheire";
5 m1.VoegMettingToeEnVerwijder(m2);
6 Console.WriteLine($"{m1.Temperatuur} en {m2.Temperatuur});
```

Dit zal resulteren in volgende output:

```
1 31 en 0
```

10.2.3 Objecten als resultaat

Weer hetzelfde verhaal: ook klassen mogen het resultaat van een methoden zijn. Stel dat we een nieuw meting object willen maken dat de dubbele temperatuur bevat van het object waarop de methode wordt aangeroepen:

```

1 public Meting GenereerRandomMetting()
2 {
3     Meting result = new Meting();
4     result.Temperatuur = Temperatuur * 2;
5     result.OpgemetenDoor = $"{OpgemetenDoor} Junior";
6     return result;
7 }
```

Deze methode kan je dan als volgt gebruiken:

```

1 m1.Temperatuur = 26;
2 m1.OpgemetenDoor = "Lieven Scheire";
3 Meting m3 = m1.GenereerRandomMetting();
```

Het object m3 zal een temperatuur van 52 bevatten en zijn opgemeten door Lieven Scheire Junior.

10.2.4 Bevallen in code

In voorgaande voorbeeld zagen we reeds dat objecten dus objecten van het eigen type kunnen teruggeven. Laten we dat voorbeeld eens doortrekken naar hoe de bevalling van een kind in C# zou gebeuren.

Baby's zijn kleine mensjes. Het is dan ook logisch dat mensen een methode `PlantVoort` hebben (we houden geen rekening met het geslacht). Volgende klasse Mens is dus perfect mogelijk:

```
1 class Mens
2 {
3     public Mens PlantVoort()
4     {
5         return new Mens();
6     }
7 }
```

Vervolgens kunnen we het volgende doen:

```
1 Mens oermoeder = new Mens();
2 Mens dochter;
3 Mens kleindochter;
4 dochter = oermoeder.PlantVoort();
5 kleindochter = dochter.PlantVoort();
```

Het is een interessante oefening om deze code eens uit te tekenen in de stack en heap inclusief de verschillende referenties.



Ik ga voorgaande code over enkele pagina's nog uitbreiden om een meer realistisch voortplantingsscenario te hebben (sommige zinnen verwacht je nooit te zullen schrijven in je leven... *I was wrong*).

10.3 Object referenties en null

Zoals nu duidelijk is bevatten referentievariabelen steeds een referentie naar een object. Maar wat als we dit schrijven:

```
1 Student stud1;  
2 stud1.Naam = "Marc Jansens";
```

Dit zal een fout geven. Het object `stud1` bevat namelijk nog geen referentie. Maar wat dan wel?

Deze variabele bevat de waarde `null`. Net zoals bij value types die een default waarde hebben als je er geen geeft (bv. 0 bij een `int`), **zo bevatten reference type variabelen altijd `null` als standaardwaarde.**

`null` is een waarde die je kan toekennen aan eender welk reference type. Je doet dit om aan te geven dat er nog geen referentie naar een effectief object in de variabele staat. Je kan dus ook op deze waarde testen.

Van zodra je een referentie naar een object (een bestaand of eentje dat je net met `new` hebt aangemaakt) aan een reference type variabele toewijst (met de `=` operator) zal de `null` waarde uiteraard overschreven worden.



Merk op dat de GC enkel op de heap werkt. Indien er in de stack dus een variabele de waarde `null` heeft zal de GC deze nooit verwijderen!

10.3.1 NullReferenceException

Een veel voorkomende foutbericht tijdens de uitvoer van je applicatie is een `NullReferenceException`. Deze zal optreden wanneer je code een object probeert te benaderen wiens waarde `null` is (een onbestaand object met andere woorden).

Laten we dit eens simuleren:

```
1 Student stud1 = null;  
2 Console.WriteLine(stud1.Name);
```

Dit zal resulteren in een foutbericht in VS bij de lijn die de uitzondering detecteert: “`System.NullReferenceException: ‘Object reference not set to an instance of an object’`. `stud1` was `null`”.



We moeten in dit voorbeeld expliciet `= null` plaatsen daar Visual Studio slim genoeg is om je te waarschuwen voor eenvoudige potentiële `NullReference` fouten en je code anders niet zal compileren.

10.3.2 NullReferenceException voorkomen

Objecten die niet bestaan zullen altijd **null** hebben. Uiteraard kan je niet altijd al je code uitvlooien waar je misschien vergeten bent een object met **new** aan te maken.

Voorts kan het ook soms *by design* zijn dat een object voorlopig **null** is.

Gelukkig kan je controleren of een object **null** is als volgt:

```
1 if(stud1 == null)
2     Console.WriteLine("Oei. Object bestaat niet.")
```

10.3.2.1 Verkorte null controle notatie

Vaak moet je dit soort code schrijven:

```
1 if(stud1 != null)
2 {
3     Console.WriteLine(stud1.Name)
4 }
```

Op die manier voorkom je een NullReferenceException. Het is uiteraard omslachtig om steeds die check te doen. Je mag daarom ook schrijven:

```
1 Console.WriteLine(stud1?.Name)
```

Het vraagteken direct na het object geeft aan: “Gelieve de code na dit vraagteken enkel uit te voeren indien het object voor het vraagteken niet null is”.

Bovenstaande code zal dus gewoon een lege lijn op scherm plaatsen indien `stud1` effectief **null** is, anders komt de naam op het scherm.

10.3.3 Return null

Uiteraard mag je ook expliciet **null** teruggeven als resultaat van een methode. Stel dat je een methode hebt die in een array een bepaald object moet zoeken. Wat moet de methode teruggeven als deze niet gevonden wordt? Inderdaad, we geven dan **null** terug.

Volgende methode zoekt in een array van studenten naar een student met een specifieke naam en geeft deze terug als resultaat. Enkel als de hele array werd doorlopen en er geen match is wordt er **null** teruggegeven (de werking van arrays van objecten wordt later besproken):

```

1 static Student ZoekStudent(Student[] array, string naam)
2 {
3     Student gevonden = null;
4     for (int i = 0; i < array.Length; i++)
5     {
6         if (array[i].Name == naam)
7             gevonden = array[i];
8     }
9
10    return gevonden;
11 }
```

10.3.4 Bevallen in code met ouders

Tijd om het voorbeeld van de *voortplanting der mensch* er nog eens bij te nemen. Beeld je nu in dat we dichter naar de realiteit willen gaan (meestal toch het doel van OOP) en de baby eigenschappen van beide willen ouders geven. Stel dat mensen een maximum lengte hebben die ze genetisch kunnen halen, aangeduid via een auto-property MaxLengte. De maximale lengte van een baby is steeds de lengte van de grootste ouder (in de echte genetica is dat natuurlijk niet).

De klasse Mens breiden we uit naar:

```

1 internal class Mens
2 {
3     public int MaxLengte {get; set;}
4
5     public Mens PlantVoort(Mens dePapa)
6     {
7         Mens baby = new Mens();
8         baby.MaxLengte = MaxLengte;
9         if(dePapa.MaxLengte >= MaxLengte)
10            baby.MaxLengte = papa.MaxLengte;
11        return baby;
12    }
13 }
```

Mooi toch?!

Om het nu volledig te maken zullen we er voor zorgen dat enkel een vrouw kan voortplanten. Voorts kan ze zich enkel voortplanten met behulp van een man (merk op dat OOP als doel heeft de realiteit te benaderen, maar ook te vereenvoudigen naargelang het probleem).

Veronderstel dat het geslacht via een enumtype (**enum** `Geslachten {Man, Vrouw}`) in een auto-property `Geslacht` wordt bewaard. We voegen daarom bovenaan in de `PlantVoort`-methode nog een kleine check in én return'n een leeg (**null**) object als de voortplanting faalt (we zouden ook een Exception kunnen opwerpen):

```
1  public Mens PlantVoort(Mens dePapa)
2  {
3      if(Geslacht == Geslachten.Vrouw
4          && dePapa.Geslacht == Geslachten.Man)
5      {
6          Mens baby = new Mens();
7          baby.MaxLengte = MaxLengte;
8          if(dePapa.MaxLengte >= MaxLengte)
9              baby.MaxLengte = papa.MaxLengte;
10         return baby;
11     }
12     return null;
13 }
```

Volgende code produceert nu een kersverse baby:

```
1 Mens mama = new Mens();
2 mama.Geslacht = Geslachten.Vrouw;
3 mama.MaxLengte = 180;
4 Mens papa = new Mens();
5 papa.Geslacht = Geslachten.Man;
6 papa.MaxLengte = 169;
7
8 Mens baby = mama.PlantVoort(papa);
```



Hopelijk voel je bij dit voorbeeld hetzelfde enthousiasme als toen we Pong naar OOP omzetten. Probeer voorgaande voorbeeld eens te schrijven met je kennis VOOR je klassen en objecten kende? Doenbaar? Zeker. Veel werk? Dat nog meer. En daar is het ons om te doen: krachtige, makkelijker te onderhouden code leren schrijven!

10.4 Namespaces en using

Je zal het keyword **namespace** al vele malen bovenaan je code hebben zien staan .

```
1  namespace MyEpicGame
2  {
3      internal class Monster
```

De naam die achter de **namespace** staat is altijd die van je project, maar waarom is dit eigenlijk?

10.4.1 Wat zijn namespaces

Een **namespace** wordt gebruikt om te voorkomen dat 2 projecten die toevallig dezelfde klassenamen hebben in conflict komen. Beeld je in dat je een project van iemand anders toevoegt aan jouw project en je ontdekt dat in dat project reeds een klasse Student aanwezig is. Hoe weet C# nu welke klasse moet gebruikt worden? Want mogelijk wens je beide te gebruiken!

De namespace rondom een klasse is als het ware een extra stukje naamgeving waarmee je kan aangeven welke klasse je juist nodig hebt. In bovenstaand stukje code heb ik een project MyEpicGame gemaakt en zoals je ziet bevat het een klasse Monster. De volledige naam (of **Fully Qualified Type Name**) van deze klasse is `MyEpicGame.Monster`.

Als ik dus even later een project met volgende namespace, en zelfde klassenaam, importeer:

```
1  namespace NietZoEpicGame
2  {
3      internal class Monster
```

Dan kan ik deze klasse aanroepen als `NietZoEpicGame.Monster` en kan er dus geen verwarring optreden.



De politie uw vriend! Inderdaad. De auteur van dit boek heeft klachten gekregen over het feit dat hij het edele beroep van politie-agent ietwat besmeurd. We willen daarom even u attenderen en, zoals een goed agent betaamd, u de weg doorheen de stad wijzen.

Als u ons tegenkomt en vraagt "Waar is de Kerkstraat." Dan zullen wij u meer informatie moeten vragen. Zonder er bij te zeggen in welke gemeente u die straat zoekt, is de kans bestaande dat we u naar de verkeerde Kerkstraat sturen. Er zijn er namelijk best veel

Kerkstraten in België en Nederland. Wel, namespaces zijn exact dat. Je kan ze vergelijken als een stadsnaam die essentieel is bij een straatnaam. De stadsnaam laat toe om zonder verwarring een straat (de klasse in dit geval) te identificeren. Nog een fijne dag!

10.4.2 using in je code

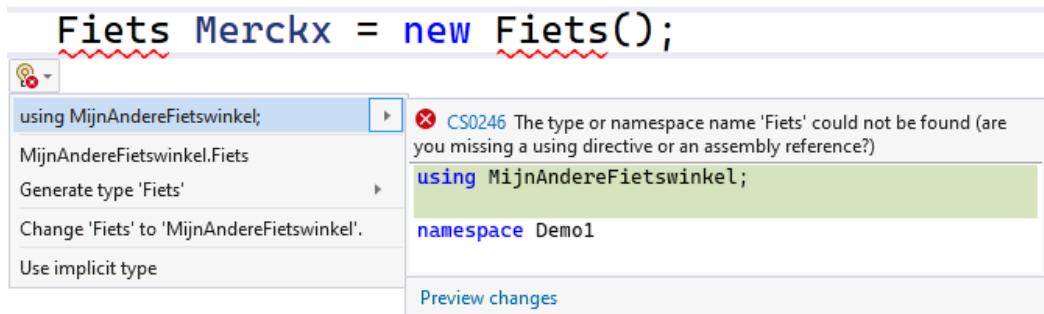
Wanneer je een bepaalde namespace nodig hebt (standaard laadt een C# 10 project er maar een handvol in) dan dien je dit bovenaan je bestand aan te geven met **using**. Bijvoorbeeld **using System.Diagnostics**. Je zegt dan eigenlijk: “Beste C#, als je een klasse zoekt en je vindt ze niet in dit project: kijk dan zeker in de *System.Diagnostics*-bibliotheek.”

10.4.3 Ontbrekende namespaces terugvinden

Het gebeurt soms dat je een klasse gebruikt en je weet zeker dat ze in jouw project of een bestaande .NET bibliotheek aanwezig is. Visual Studio kan je helpen de namespace van deze klasse te zoeken moest je daar te lui voor zijn.

Je doet dit door de naam van de klasse te schrijven (op de plek waar je deze nodig hebt) en dan op het lampje dat links in de rand verschijnt te klikken. Indien de klasse gekend is door VS zal je nu de optie krijgen om automatisch:

- oftewel **using**, met de juiste namespace, bovenaan je huidige codebestand te plaatsen.
- oftewel de volledige naam van de klasse uit te schrijven (dus inclusief de namespace).



Figuur 10.9: Handig toch!

Trouwens: de optie **Generate type ...** zal je ook vaak kunnen gebruiken. Wanneer de klasse in kwestie (Fiets hier) nog niet bestaat en je wilt deze automatische laten genereren (in een apart bestand) dan zal deze optie dat voor je doen.



Maar hoe weet C# nu welke bibliotheken allemaal beschikbaar zijn? Wel, je kan in je project via de solution explorer kijken welke bibliotheken (meestal in de vorm van DLL-bestanden) werden toegevoegd. In je solution explorer klik je hiervoor de *Dependencies* open. Daar kan je dan zien in welke bibliotheken VS mag zoeken als je een klasse nodig hebt die niet gekend is. Klik bijvoorbeeld eens onder *Dependencies* de sectie *FrameWorks* open en dan *MicrosoftT.NETCore.App*. Je zal er onder andere alle **System.** bibliotheken zien staan.

10.4.3.1 NuGet

Je kan ook extra bibliotheken toevoegen aan je Dependencies.

Rechterklik maar eens op Dependencies en zie wat je allemaal kunt doen. **Vooral de NuGet packages zijn een erg nuttig en krachtig hulpmiddel.**¹

Helaas kan niet alles over C# en .NET in één boek verzameld worden. Weet echter dat er erg nuttige, toffe en zelfs grappige NuGet packages bestaan. Zoek bijvoorbeeld maar eens naar de *Colorful.Console* NuGet!

¹Lees er alles over op docs.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio.

10.5 Exception handling

Het wordt tijd om de olifant in de kamer te benoemen. Het wordt tijd om een bekentenis te maken... Ben je er klaar voor?! Hier komt ie. Luister goed, maar zeg het niet door: ik heb al de hele tijd informatie voor je achter gehouden! Sorry, het was sterker dan mezelf. Maar ik deed het voor jou. Het was de enige manier om ervoor te zorgen dat je leerde programmeren zonder constant bugs in je code achter te laten. Dus ja, hopelijk neem je het me niet kwalijk?

Het wordt tijd om **exception handling** er bij te halen! Een essentiële programmeertechniek die ervoor zorgt dat je programma minder snel zal crashen indien er zich **uitzonderingen** tijdens de uitvoer voordoen.

Wat een dramatische start zeg. Waar was dat voor nodig?! De reden is eenvoudig: exception handling is een tweesijdend zwaard. Je zou exception handling kunnen gebruiken om al je bugs op te vangen, zodat de eindgebruiker niet ziet hoe vaak je programma zou crashen zonder exception handling.

Maar uiteindelijk blijf je wel met slechte code zitten en een gouden regel in programmeren is dat slechte code je altijd zal achtervolgen en je ooit dubbel en hard zal straffen voor iedere bug waar je te lui voor was om op te lossen. **Kortom, exception handling is de finale fase van goedgeschreven code.**

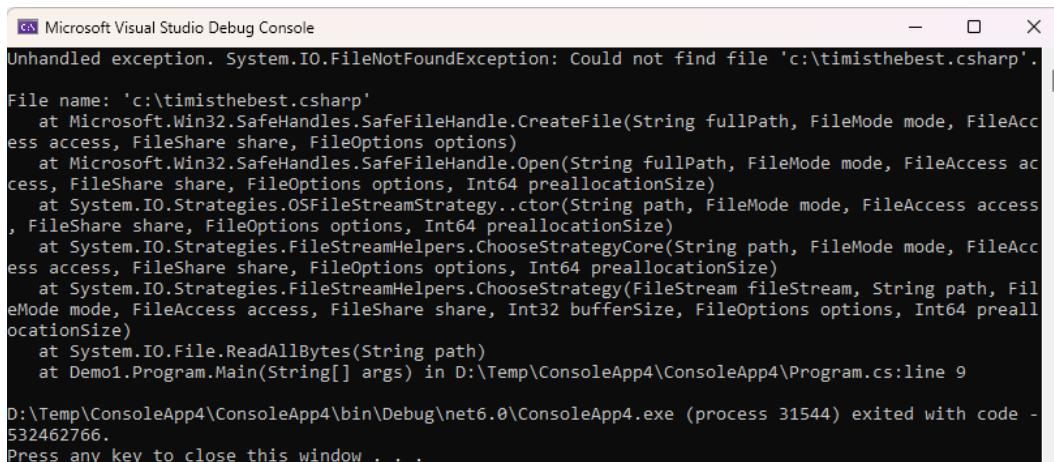
10.5.1 Waarom exception handling?

Veel fouten in je code zijn het gevolg van:

- **Het aanroepen van data die er niet is**: bijvoorbeeld een bestand dat werd verplaatst of hernoemd . Of wat denk van het wegvalLEN van het wifi-signaal net wanneer je programma iets van een online database nodig heeft.
- **Foute invoer door de gebruiker** : denk aan de gebruiker die een letter invoert terwijl het programma een getal verwacht.
- **Programmeefouten** : de ontwikkelaar gebruikt een object dat nog niet met de **new** operator werd geïnitialiseerd. Of bijvoorbeeld een deling door nul in een wiskundige berekening.

Voorgaande zaken zijn eigenlijk geen fouten, maar uitzonderingen (**exceptions**). Ze doen zich zelden voor, maar hebben wel een invloed op de correcte uitvoer van je programma. Je programma zal met deze uitzonderingen rekening moeten houden wil je een gebruiksvriendelijk programma hebben. Veel uitzonderingen gebeuren buiten de wil van het programma om (geen wifi , foute invoer, enz.). Door deze uitzonderingen af te handelen (**exception handling**), kunnen we ons programma instructies geven om alternatieve acties uit te voeren wanneer er een uitzondering optreedt.

Je hebt waarschijnlijk al eerder exceptions gezien in je eigen programma's. Als je programma plots een hele hoop tekst toont (waaronder het woord “Exception”) en vervolgens direct afsluit, heb je een exception gegenereerd die niet is afgehandeld.



```
Microsoft Visual Studio Debug Console
Unhandled exception. System.IO.FileNotFoundException: Could not find file 'c:\timisthebest.csharp'.
File name: 'c:\timisthebest.csharp'
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath, FileMode mode, FileAccess access, FileShare share, FileOptions options)
   at Microsoft.Win32.SafeHandles.SafefileHandle.Open(String fullPath, FileMode mode, FileAccess access, FileShare share, FileOptions options, Int64 preallocationSize)
   at System.IO.Strategies.OSFileStreamStrategy..ctor(String path, FileMode mode, FileAccess access, FileShare share, FileOptions options, Int64 preallocationSize)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String path, FileMode mode, FileAccess access, FileShare share, FileOptions options, Int64 preallocationSize)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategy(FileStream fileStream, String path, FileMode mode, FileAccess access, FileShare share, Int32 bufferSize, FileOptions options, Int64 preallocationSize)
   at System.IO.File.ReadAllBytes(String path)
   at Demo1.Program.Main(String[] args) in D:\Temp\ConsoleApp4\ConsoleApp4\Program.cs:line 9

D:\Temp\ConsoleApp4\ConsoleApp4\bin\Debug\net6.0\ConsoleApp4.exe (process 31544) exited with code -532462766.
Press any key to close this window . . .
```

Figuur 10.10: Een joekel van een foutbericht die je gebruiker huilend zal wegjagen.

Je moet zelfs niet veel moeite doen om uitzonderingen te genereren. Denk maar aan volgende voorbeeld waarbij je een exception kan genereren door een 0 in te geven, of iets anders dan een getal.

```
1 Console.WriteLine("Geef een getal aub");
2 int noemer = Convert.ToInt32(Console.ReadLine());
3 double resultaat = 100/noemer;
4 Console.WriteLine($"100/{noemer} is gelijk aan {resultaat}");
```

10.5.2 Exception handling met try en catch

Het mechanisme om exceptions af te handelen in C# bestaat uit 2 delen:

- Een **try blok**: binnen dit blok staat de code die je wil controleren op uitzonderingen omdat je weet dat die hier kunnen optreden.
- Een of meerdere **catch-blokken**: dit blok zal mogelijk exceptions die in het bijhorende try-block voorkomen opvangen. Met andere woorden: in dit blok staat de code die de uitzondering zal verwerken zodat het programma op een deftige manier verder kan of meer elegant zichzelf afsluiten (*graceful shutdown*).

De syntax is als volgt (let er op dat de catch blok onmiddellijk na het try-blok komt):

```
1 try
2 {
3     //code waar exception mogelijk kan optreden
4 }
5 catch
6 {
7     //exception handling code hier
8 }
```

10.5.3 Een try catch voorbeeld

In volgend stukje code kunnen uitzonderingen optreden zoals we zonet zagen:

```
1 string input = Console.ReadLine();
2 int converted = Convert.ToInt32(input)
```

Een FormatException zal optreden wanneer de gebruiker tekst of een kommagetal invoert. De Convert.ToInt32() methode kan niet met andere input dan gehele getallen werken.

Ik toon nu hoe we dit met exception handling kunnen opvangen²:

```
1 try
2 {
3     string input = Console.ReadLine();
4     int converted = Convert.ToInt32(input);
5 }
6 catch
7 {
8     Console.WriteLine("Verkeerde invoer!");
9 }
```

Indien er nu een uitzondering optreedt dan zal de tekst “Verkeerde invoer” getoond worden. Vervolgens gaat het programma verder met de code die mogelijk na het **catch**-blok staat.

10.5.4 Een exception genereren met throw

Je kan ook zelf eender waar in je code een uitzondering **opwerpen**. Je doet dit met het **throw** keyword. De werking is quasi dezelfde als het **return** keyword. Alleen zal bij een **throw** je terug gaan tot de eerste plek waar een **catch** klaarstaat om de uitzondering op te vangen. Om een uitzondering op te werpen dien je eerst een Exception object aan te maken en daar de nodige informatie in te plaatsen. In hoofdstuk 14 ga ik hier nog wat dieper op in, maar hier alvast een voorbeeldje:

```
1 //Een error treedt op
2 throw new Exception("Wow, dit loopt fout");
```

Afhankelijk van het soort fout kunnen we echter ook andere soort uitzonderingen opwerpen. Draai daarom snel deze pagina om en ontdek hoe dit kan!

²Merk op dat dit probleem eleganter kan opgelost worden met TryParse wat in het appendix wordt uitgelegd.

10.5.5 Meerdere catchblokken

Exception is een klasse van het .NET framework. Er zijn van deze basis-klasse meerdere Exception-klassen afgeleid die een specifieke uitzondering behelzen. Enkele veelvoorkomende zijn:

Klasse	Omschrijving
Exception	Basisklasse
SystemException	Klasse voor uitzonderingen die niet al te belangrijk zijn en die mogelijk verholpen kunnen worden.
IndexOutOfRangeException	De index is te groot of te klein voor de benadering van een array
NullReferenceException	Benadering van een niet-geïnitialiseerd object

Je kan in het catch blok aangeven welke soort exceptions je wil vangen in dat blok. Als je bijvoorbeeld alle Exceptions wil opvangen schrijf je:

```
1 catch (Exception e)
2 {
3 }
```

Hiermee vangen we dus **alle** Exceptions op, daar alle Exceptions van de klasse Exception afgeleid zijn en dus ook zelf een Exception zijn. De identifier e kies je zelf en wordt gebruikt om vervolgens in het **catch** block de nodige informatie uit het opgevangen Exception object (e) uit te lezen. Ik leg dit zo meteen uit.

We kunnen nu echter ook specifieke exceptions opvangen. De truc is om de meest algemene exception onderaan te zetten en naar boven toe steeds specieker te worden. We maken een *fallthrough mechanisme* (wat we ook in een **switch** al hebben gezien).

Stel bijvoorbeeld dat we weten dat de FormatException kan voorkomen en we willen daar iets mee doen. Volgende code toont hoe dit kan:

```
1 try
2 {
3     //...
4 }
5 catch (FormatException e)
6 {
7     Console.WriteLine("Verkeerd invoerformaat");
8 }
9 catch (Exception e)
10 {
11     Console.WriteLine("Exception opgetreden");
12 }
```

Indien een `FormatException` optreedt dan zal het eerste catch-blok uitgevoerd worden, in alle andere gevallen het tweede. Het tweede blok zal niet uitgevoerd worden indien een `FormatException` optreedt.

10.5.6 Welke exceptions worden gegooid?

De online .NET documentatie is de manier om te weten te komen welke exceptions een methode mogelijk kan opgooien. Gaan we bijvoorbeeld naar de documentatie van de `Int32.Parse`-methode³ dan zien we daar een sectie “Exceptions” waar klaar en duidelijk wordt beschreven wanneer welke exception wanneer wordt opgeworpen.

Exceptions

`ArgumentNullException`

`s` is `null`.

`ArgumentException`

`style` is not a `NumberStyles` value.

-or-

`style` is not a combination of `AllowHexSpecifier` and `HexNumber` values.

`FormatException`

`s` is not in a format compliant with `style`.

`OverflowException`

`s` represents a number less than `.MinValue` or greater than `.MaxValue`.

-or-

`s` includes non-zero, fractional digits.

Figuur 10.11: Een deel van de documentatie.

³Zie docs.microsoft.com/dotnet/api/system.int32.parse.

10.5.7 Werken met de exception parameter

De Exceptions die worden opgegooid door een methode zijn objecten van de Exception-klasse. Deze klasse bevat standaard een aantal interessante zaken, die je kan oproepen in je code.

Bovenaan de declaratie van het `catch`-blok geef je aan hoe het exception object in het blok zal heten. In de vorige voorbeelden was dit altijd e (standaardnaam).

Alle Exception-objecten bevatten volgende informatie:

Element	Omschrijving
Message	Foutmelding in relatief eenvoudige taal.
StackTrace	Lijst van methoden die de exception hebben doorgegeven.
TargetSite	Methode die de exception heeft gegenereerd (staat bij StackTrace helemaal bovenaan).
ToString()	Geeft het type van de exception, Message en StackTrace terug als string.

We kunnen via deze parameter meer informatie uit de opgeworpen uitzondering uitlezen en bijvoorbeeld aan de gebruiker tonen:

```
1 catch (Exception e)
2 {
3     Console.WriteLine("Exception opgetreden");
4     Console.WriteLine("Message:" + e.Message);
5
6     Console.WriteLine($"Targetsite: {e.TargetSite}");
7     Console.WriteLine("StackTrace: {e.StackTrace}");
8 }
```



Vanuit een security standpunt is het zelden aangeraden om Exception informatie zomaar rechtstreeks naar de gebruiker te sturen. Mogelijk bevat de informatie gevoelige informatie en zou deze door kwaadwillige gebruikers kunnen misbruikt worden om bugs in je programma te vinden.

10.5.8 Waar exception handling in code plaatsen?

De plaats in je code waar je je exceptions zal opvangen, heeft invloed op de totale werking van je code.

Stel dat je volgende stukje code hebt waarin je een methode hebt die een lijst van strings zal beschouwen als urls die moeten gedownload worden. Indien er echter fouten in de string staan dan zal er een uitzondering optreden bij lijn 16. De tweede url ("http:\\www.humo.be") bevat namelijk een bewuste fout: de schuine strepen staan in de verkeerde richting.



Als sneak preview tonen we ook ineens hoe arrays van objecten werken.

```
1 static void Main(string[] args)
2 {
3     string[] urllist = new string[3];
4     urllist[0] = "http://www.ziescherp.be";
5     urllist[1] = "http:\\www.humo.be";
6     urllist[2] = "timdams.com";
7     DownloadAllUris(urllist);
8 }
9
10 static public void DownloadAllUris(string[] urls)
11 {
12     System.Net.WebClient webClient = new System.Net.WebClient();
13
14     for(int i = 0; i < urls.Length;i++)
15     {
16         Uri uri = new Uri(urls[i]);
17         string result = webClient.DownloadString(uri);
18         Console.WriteLine($"{uri} gedownload. Resultaat: {result}");
19     }
20 }
```



De WebClient-klasse laat toe te werken met online zaken, zoals websites, restful API's, webservices, enz. Je kan er bijvoorbeeld heel makkelijk een webscraper mee maken.

We bekijken nu een aantal mogelijk try/catch locaties in deze code en zien welke impact deze hebben op de totale uitvoer van het programma.

10.5.9 Rondom methode-aanroep in z'n geheel

```

1  try
2  {
3      DownloadAllUris(urllist);
4  }
5  catch (Exception ex)
6  {
7      Console.WriteLine(ex.Message);
8 }
```

Zal resulteren in:

```

1  http://www.ziescherp.be gedownload!
2  Ongeldige URI: kan de Authority/Host niet parsen.
```

Met andere woorden, zolang de urls geldig zijn zal de download lukken. Bij de eerste fout die optreedt zal de volledige methode echter stoppen. Dit is waarschijnlijk enkel wenselijk indien de code erna de informatie van ALLE urls nodig heeft.

10.5.10 Rond afzonderlijke elementen in de loop

Mogelijk wil je echter dat je programma blijft werken indien er 1 of meerdere urls niet werken. We plaatsen dan de try catch niet rond de methode `DownloadAllUris`, maar net binnenin de methode zelf rond het gedeelte dat kan mislukken:

```

1  for(int i = 0; i < urls.Length;i++)
2  {
3      try
4      {
5          Uri uri = new Uri(urls[i]);
6          string result = webClient.DownloadString(uri);
7          Console.WriteLine($"{uri} gedownload. Resultaat: {result}");
8      }
9      catch (Exception ex)
10     {
11         Console.WriteLine(ex.Message);
12     }
13 }
```

Dit zal resulteren in:

```

1  http://www.ziescherp.be gedownload!
2  Ongeldige URI: kan de Authority/Host niet parsen.
3  Ongeldige URI: de indeling van de URI kan niet worden bepaald.
```

Met andere woorden, indien een bepaalde url niet geldig is dan zal deze overgeslagen worden en gaat de methode verder naar de volgende. Op deze manier kunnen we alsnog alle urls trachten te downloaden.

10.5.11 finally

Soms zal je na een try-catch-blok ook nog een **finally** blok zien staan. Dit blok laat je toe om code uit te voeren die ALTIJD moet uitgevoerd worden, ongeacht of er een exception is opgetreden of niet. Je kan dit gebruiken om bijvoorbeeld er zeker van te zijn dat het bestand dat je wou uitlezen terug afgesloten wordt.

```
1  try
2  {
3      Uri uri = new Uri(urls[i]);
4      string result = webClient.DownloadString(uri);
5      Console.WriteLine(${uri} gedownload. Resultaat: {result});
6  }
7  catch (Exception ex)
8  {
9      Console.WriteLine(ex.Message);
10 }
11 finally
12 {
13     //Plaats hier zaken die sowieso moeten gebeuren.
14 }
```

11 Gevorderde klasseconcepten

Nu we weten wat er allemaal achter de schermen gebeurt met onze objecten, wordt het tijd om wat meer geavanceerde concepten van klassen en objecten te bekijken.

We hebben al ontdekt dat een klasse kan bestaan uit:

- **Instantievariabelen:** variabelen die de toestand van het individuele object bijhouden.
- **Methoden:** om objecten voor ons te laten werken (gedrag).
- **Properties:** om op een gecontroleerde manier toegang tot de interne staat van de objecten te verkrijgen.

Uiteraard is dit niet alles. In dit hoofdstuk bekijken we:

- **Constructors:** een gecontroleerde manier om de beginstaat van een object in te stellen.
- **static:** die je de mogelijkheid geeft een (deel van je) klasse te laten werken als een object.
- **Object initializer syntax:** een recente C# aanvulling die het aanmaken van nieuwe objecten vereenvoudigd.

11.1 Constructors

11.1.1 Werking new operator

Objecten die je aanmaakt komen niet zomaar tot leven. Nieuwe objecten maken we aan met behulp van de **new** operator zoals we al gezien hebben:

```
1 Student frankVermeulen = new Student();
```

De **new** operator doet 3 dingen:

- Het maakt een object aan in het heap geheugen.
- Het roept de **constructor** van het object aan voor eventuele extra initialisatie.
- Het geeft een referentie naar het object in het heap geheugen terug.

Via de constructor van een klasse kunnen we extra code meegeven die moet uitgevoerd worden **telkens een nieuw object van dit type wordt aangemaakt**.

De constructor is een unieke methode die wordt aangeroepen bij het aanmaken van een object. Daarom dat we dus ronde haakjes zetten bij **new Student()**.

Momenteel hebben we in de klasse `Student` de constructor nog niet explicet beschreven. Maar zoals je aan bovenstaande code ziet bestaat deze constructor al wel degelijk. Hij doet echter niets extra. Zo krijgen de instantievariabelen gewoon hun default waarde toegekend, afhankelijk van hun datatype.



De naam “constructor” zegt duidelijk waarvoor het concept dient: *het construeren van objecten*. Constructors mogen maar op 1 moment in het leven van een object aangeroepen worden: tijdens hun geboorte m.b.v. `new`. **Je mag een constructor op geen enkel ander moment gebruiken!**

11.1.2 Soorten constructors

Als programmeur van eigen klassen zijn er 3 opties voor je:

- Je gebruikt **geen** zelfgeschreven constructors: het leven gaat voort zoals het is. Je kunt objecten aanmaken zoals eerder getoond. Een *onzichtbare* default constructor wordt voor je uitgevoerd.
- Je hebt enkel een **default constructor** nodig: je kan nog steeds objecten met `new Student()` aanmaken. Maar je gaat zelf beschrijven wat er moet gebeuren in de default constructor. De default constructor herken je aan het feit dat je geen parameters meegeeft aan de constructor tijdens de `new` aanroep.
- Je gebruikt één of meerdere **overloaded constructors**: hierbij zal je dan actuele parameters kunnen meegeven bij de creatie van een object. Denk maar aan `new Student(24, "Jos")`.



Constructors zijn soms gratis, soms niet. Een lege default constructor voor je klasse krijg je standaard wanneer je een nieuwe klasse aanmaakt. Je ziet deze niet en kan deze niet aanpassen. Je kan echter daarom altijd objecten met `new Student()` aanmaken. **Van zodra je echter beslist om zelf één of meerdere constructors te schrijven zal C# zeggen: “Ok, jij je zin, nu doe je alles zelf”. De default constructor die je gratis kreeg zal ook niet meer bestaan. Heb je die nodig dan zal je die dus zelf moeten schrijven!**

Een nadeel van C# is dat het soms dingen voor ons achter de schermen doet, en soms niet. Het is mijn taak je dan ook duidelijk te maken wanneer dat wél en wanneer dat net niet gebeurt. Ik vergelijk het altijd met het werken met aannemers: soms ruimen ze hun eigen rommel op nadien, maar soms ook niet. Alles hangt er van af hoe ik die aannemer heb opgetrommeld.

11.1.3 Default constructors

De default constructor is een constructor die geen extra parameters aanvaardt. Een default constructor bestaat ALTIJD uit volgende vorm:

- Iedere constructor is altijd **public**.
- Heeft geen returntype, ook niet **void**.
- Heeft als naam de naam van de klasse zelf.
- Heeft geen extra formele parameters.

Stel dat we een klasse Student hebben:

```
1 internal class Student
2 {
3     public int UurVanInschrijven {private set; get;}
4 }
```

We willen telkens een Student-object worden aangemaakt bijhouden op welk uur van de dag dit plaatsvond. Eerst schrijven de default constructor, deze ziet er als volgt uit:

```
1 internal class Student
2 {
3     public Student()
4     {
5         // zet hier de code die bij initialisatie moet gebeuren
6     }
7     public int UurVanInschrijven {private set; get;}
8 }
```

Zoals verteld moet de constructor de naam van de klasse hebben, **public** zijn en geen returntype definiëren.

Vervolgens voegen we de code toe die we nodig hebben:

```
1 internal class Student
2 {
3     public Student()
4     {
5         UurVanInschrijven = DateTime.Now.Hour;
6     }
7
8     public int UurVanInschrijven {private set; get;}
9 }
```

Telkens we nu een object zouden aanmaken met **new** Student() zal de waarde in UurVanInschrijven afhangen van het moment waarop we de code uitvoeren. Beeld je in dat we dit programma uitvoeren om half twaalf 's morgens:

```
1 Student eenStudent = new Student();
```

Dan zal de property UurVanInschrijven van eenStudent op 11 worden ingesteld.



Constructors zijn soms nogal zwaarwichtig indien je enkel een eenvoudige auto-property een startwaarde wenst te geven. Wanneer dat het geval is mag je dit ook als volgt doen:

```
1 internal class Student
2 {
3     public int UurVanInschrijven {private set; get;} = 2;
4 }
```

11.1.4 Overloaded constructors

Soms wil je parameters aan een object meegeven bij de creatie ervan. We willen bijvoorbeeld de bijnaam meegeven die het object moet hebben bij het aanmaken.

Met andere woorden, stel dat we dit willen schrijven:

```
1 Student jos = new Student("Lord Oakenwood");
```

Als we dit met voorgaande klasse uitvoeren zal de code een fout geven. C# vindt geen constructor die een **string** als actuele parameter aanvaardt.

Net zoals bij overloading van methoden kunnen we ook constructors overladen. De code is verrassend gelijkaardig aan method overloading:

```
1 internal class Student
2 {
3     public Student(string bijnaamIn)
4     {
5         bijNaam = bijnaamIn;
6     }
7     public string BijNaam { get; private set; }
8 }
```

Dat was eenvoudig, hé?

Maar denk eraan: je hebt een overloaded constructor geschreven. Hierdoor zegt C# eigenlijk tegen je: "Ok, je schrijft zelf constructors? Trek je plan nu maar. De default constructor zal je ook nu zelf moeten schrijven."

Je kan nu enkel je objecten nog via de overloaded constructors aanmaken. Schrijf je **new Student()** dan zal je een foutbericht krijgen. Wil je de default constructor toch nog hebben dan zal je die dus ook expliciet moeten schrijven, bijvoorbeeld:

```
1 internal class Student
2 {
3     private const string DEFBIJNAAM = "Geen";
4     //Default
5     public Student()
6     {
7         BijNaam = DEFBIJNAAM;
8     }
9     //Overloaded
10    public Student(string bijnaamIn)
11    {
12        BijNaam = bijnaamIn;
13    }
14    public string BijNaam { get; private set; }
15 }
```



Voorgaande wil ik nog eenmaal herhalen. Herinner je m'n voorbeeld van die aanname-sers die soms wel en soms niet opruimden? Ik zal nog eens samenvatten hoe het zit met constructors in C#:

Als je geen constructors schrijft krijg je een default constructor gratis. Die doet echter niets extra buiten alle instantievariabelen en properties default waarden geven.

Van zodra je één constructor zelf schrijft krijg je niets meer gratis én zal je dus zelf die constructors moeten bijschrijven die jouw code vereist.

11.1.4.1 Meerdere overloaded constructors

Wil je meerdere overloaded constructors dan mag dat ook. Je wilt misschien een constructor die de bijnaam vraagt alsook een **bool** om mee te geven of het om een werkstudent gaat:

```
1 internal class Student
2 {
3     private const string DEFBIJNAAM = "Geen";
4     //Default
5     public Student()
6     {
7         BijNaam = DEFBIJNAAM;
8     }
9
10    //Overloaded 1
11    public Student(string bijnaamIn)
12    {
13        BijNaam = bijnaamIn;
14    }
15
16    //Overloaded 2
17    public Student(string bijnaamIn, bool isWerkStudentIn)
18    {
19        BijNaam = bijnaamIn;
20        IsWerkStudent = isWerkStudentIn
21    }
22
23    public string BijNaam { get; private set; }
24    public string IsWerkStudent { get; private set; }
25
26 }
```



Merk op dat je ook **full properties best aanroeft in je constructor** en niet rechtstreeks de achterliggende instantievariabele. Zo kan je ogenblikkelijk de typische controles in een **set** in gebruik nemen.

Beeld je in dat het schoolsysteem crasht wanneer een nieuwe student een onbeleefd bijnaam invoert. Wanneer dit gebeurt moet de bijnaam altijd gewoon op "Good boy" gezet worden, ongeacht de effectieve bijnaam van de student. Via een **set**-controle kunnen we dit doen én vervolgens passen we de auto-property aan naar een full property zodat er een ingebouwde controle kan plaatsvinden:

```
1  internal class Student
2  {
3      private const string DEFBIJNAAM = "Good boy";
4      //Default
5      public Student()
6      {
7          bijNaam = DEFBIJNAAM;
8      }
9
10     //Overloaded
11     public Student(string bijnaamIn)
12     {
13         bijNaam = bijnaamIn;
14     }
15
16     public string BijNaam
17     {
18         private set
19         {
20             if(value == "stommerik") //pardon my french
21             {
22                 bijNaam = DEFBIJNAAM;
23             }
24             else
25                 bijNaam = value;
26         }
27         get
28         {
29             return bijNaam;
30         }
31     }
32
33     private string bijNaam;
34 }
```

Deze manier voorkomt dat de constructors verantwoordelijk zijn opdat properties de juiste waarden krijgen. Leg steeds de verantwoordelijk bij het element zelf. Door dit te doen hoef je ook niet in iedere constructor te controleren doorgegeven parameters wel geldig zijn. Ook hier blijft de regel gelden: als je dubbele code dicht bij elkaar ziet staan dan is de kans groot dat je dit kan vereenvoudigen.

11.1.5 Constructors hergebruiken met `this()`

Beeld je in dat je volgende klasse hebt:

```
1  internal class Microfoon
2  {
3      public Microfoon(string merkIn, bool isUitverkochtIn)
4      {
5          IsUitverkocht = isUitverkochtIn;
6          Merk = merkIn;
7      }
8
9      public Microfoon(string merkIn)
10     {
11         IsUitverkocht = false;
12         Merk = merkIn;
13     }
14
15     public Microfoon()
16     {
17         Merk = "Onbekend";
18         isUitverkocht = true;
19     }
20
21     public string Merk { get; set;}
22     public bool IsUitverkocht {get; set;}
23 }
```

Bij voorgaande code gaat er mogelijk bij sommige van jullie een alarmbelletje af vanwege de kans op quasi dezelfde code in de verschillende constructors. En dat is een terecht alarm!

Om te voorkomen dat we steeds dezelfde toewijzingen moeten schrijven in constructors laat C# toe dat je een andere constructor kunt aanroepen bij een constructor call.

We gebruiken hier een speciale methode aanroep (`this()`) bij de constructorsignatuur. Via deze aanroep kunnen we dan eventueel parameters meegeven, afhankelijk van wat we nodig hebben. De compiler zal aan de hand van de parameters (of het ontbreken) beslissen welke constructor nodig is.

Dit gebeurt met behulp van de klassieke *method overload resolution* en de **betterness** regel.

Voorgaande klasse gaan we herschrijven zodat alle constructors de bovenste overloaded constructor gebruiken en zo voorkomen dat we te veel dubbele code hebben:

```

1 internal class Microfoon
2 {
3     public Microfoon(string merkIn, bool isUitverkochtIn)
4     {
5         IsUitverkocht = isUitverkochtIn;
6         Merk = merkIn;
7     }
8
9     public Microfoon(string merkIn): this(merkIn, false)
10    {   }
11
12    public Microfoon(): this ("Onbekend", true)
13    {   }
14
15    public string Merk { get; set; }
16    public bool IsUitverkocht {get; set; }
17 }
```

Bij de tweede overloaded constructor geven we de binnenkomende parameter merkIn gewoon door naar de **this()** aanroep. Voorts voegen we er nog een tweede literal (**false**) aan toe. De compiler zal nu via **method overload resolution** op zoek gaan naar de best passende constructor, wat in dit geval de bovenste overloaded constructor zal zijn.

Uiteraard ben je vrij om in de constructor zelf nog steeds code te plaatsen. Het is gewoon belangrijk dat je de volgorde begrijpt waarin de constructor-code wordt doorlopen. Stel dat we volgende constructor toevoegen:

```

1 public Microfoon(bool isUitverkochtIn): this("Bovarc",
2     isUitverkochtIn)
3     {
4         Merk = "Wit Product";
5     }
```

Wanneer we een object aanmaken (met **new Microfoon(true)**) dan zal uiteindelijk dit object van het merk **Wit Product** zijn. Er gebeurt namelijk het volgende:

1. De overloaded constructor **Microfoon(bool isUitverkochtIn)** wordt aangeroepen.
2. Ogenblikkelijk wordt de meegegeven actuele parameter **isUitverkochtIn** doorgegeven om de overloaded constructor **Microfoon(string merkIn, bool isUitverkochtIn)** te benaderen.
3. Deze constructor zal het **Merk** op **Bovarc** zetten en **IsUitverkocht** op **true** (daar we die parameter doorgeven).
4. We keren nu terug naar de constructor **Microfoon(bool isUitverkochtIn)** en voeren de code hiervan uit. Bijgevolg wordt de waarde in **Merk** overschreven met **Wit Product**.

11.1.5.1 Welke constructors moet ik nu eigenlijk allemaal voorzien?

Dit hangt natuurlijk af van de soort klasse dat je maakt. Een constructor is minimaal nodig om ervoor te zorgen dat alle variabele die essentieel zijn in je klasse een beginwaarde hebben. Beeld je volgende klasse voor die een breuk voorstelt:

```

1 internal class Breuk
2 {
3     public int Noemer {get; private set;}
4     private int Teller {get; private set;}
5     public double BerekenBreuk()
6     {
7         return (double)Teller/Noemer;
8     }
9 }
```

De methode zal een `DivideByZeroException` opleveren als ik de methode `BerekenBreuk` zou aanroepen nog voor de `Noemer` een waarde heeft gekregen (deling door nul, weet je wel):

```

1 Breuk eenBreuk = new Breuk();
2 int resultaat = eenBreuk.BerekenBreuk(); //BAM! Een exception!
```

Via een constructor kunnen we dit soort bugs voorkomen. We beschermen ontwikkelaars hiermee dat ze jouw klasse foutief gebruiken. Door een overloaded constructor te schrijven die een noemer en teller vereist verplichten we de ontwikkelaar jouw klasse correct te gebruiken. Je kan niet per ongeluk breuk-objecten met de default constructor aanmaken.

Eerst veranderen we de auto-property `Noemer` naar een full property:

```

1 private int noemer;
2 public int Noemer
3 {
4     get
5     {
6         return noemer;
7     }
8     private set
9     {
10        if(value != 0)
11            noemer = value;
12        else
13            noemer = 1; //of werp Exception op zoals eerder uitgelegd
14    }
15 }
```

En vervolgens voegen we een overloaded constructor toe:

```
1 public Breuk(int tellerIn, int noemerIn)
2 {
3     Teller = tellerIn;
4     Noemer = noemerIn
5 }
```

Finaal wordt dan onze klasse:

```
1 internal class Breuk
2 {
3     public Breuk(int tellerIn, int noemerIn)
4     {
5         Teller = tellerIn;
6         Noemer = noemerIn
7     }
8     private int Teller {get; private set;}
9
10    private int noemer;
11    public int Noemer
12    {
13        get
14        {
15            return noemer;
16        }
17        private set
18        {
19            if(value != 0)
20                noemer = value;
21            else
22                noemer = 1; //of werp Exception op zoals eerder
23                uitgelegd.
24        }
25    }
}
```

Hierdoor kan ik geen Breuk objecten meer als volgt aanmaken: Breuk eenBreuk = **new** Breuk(); Maar ben ik verplicht deze als volgt aan te maken:

```
Breuk eenBreuk = new Breuk(21,8);
```

11.1.6 Een wereld met OOP: Pong constructors

We zullen deze nieuwe informatie gebruiken om onze Pong-klasse uit het eerste hoofdstuk te verbeteren door deze de nodige constructors te geven. Namelijk een default die een balletje aanmaakt dat naar rechtsonder beweegt, en één overloaded constructor die toelaat dat we zelf kunnen kiezen wat de beginwaarden van X, Y, VX en VY zullen zijn:

```

1 internal class Balletje
2 {
3     public Balletje(int xin, int yin, int vxIn, int vyIn)
4     {
5         X = xin;
6         Y = yin;
7         VX = vxIn;
8         VY = vyIn;
9     }
10
11    public Balletje(): this(5,5,1,1)
12    {
13    }
14
15    }
16    //...

```

We kunnen nu op 2 manieren balletjes aanmaken:

```

1 Balletje bal1 = new Balletje();
2 Balletje bal2 = new Balletje(10,8,-2,1);

```



Je zou ook kunnen overwegen om in de default constructor het balletje een willekeurige locatie en snelheid te geven:

```

1 static Random rng =new Random();
2
3 public Balletje()
4 {
5     X = rng.Next(0, Console.WindowWidth);
6     Y = rng.Next(0, Console.WindowWidth);
7     VX = rng.Nex(-2,3);
8     VY = rng.Nex(-2,3);
9 }

```

11.2 Object initializer syntax

Het is niet altijd duidelijk hoeveel overloaded constructors je juist nodig hebt. Meestal beperken we het tot de default constructor en 1 of 2 heel veel gebruikte overloaded constructors.

Dankzij **object initializer syntax** kan je ook parameters tijdens de aanmaak van objecten meegeven zonder dat je hiervoor een specifieke constructor moet schrijven.

Object initializer syntax laat je toe om tijdens creatie van een object, properties beginwaarden te geven.



Object initializer syntax is een eerste glimp in het feit waarom properties zo belangrijk zijn in C#. Je kan object initializer syntax enkel gebruiken om via properties je object extra beginwaarden te geven.

Stel dat we volgende klasse hebben waarin we enkele auto-properties gebruiken. Merk op dat dit even goed full properties mochten zijn. Voor object initializer syntax maakt dat niet uit, het ziet toch enkel maar het **public** gedeelte van de klasse:

```
1 internal class Meting
2 {
3     public double Temperatuur {get;set;}
4     public bool IsGeconfermeerd {get;set;}
5 }
```

We kunnen deze properties beginwaarden geven via volgende initializer syntax:

```
1 Meting meting = new Meting() { Temperatuur = 3.4, IsGeconfermeerd =
    true};
```

Object initializer syntax bestaat er uit dat je een object aanmaakt met de **default constructor** en dat je dan tussen accolades een lijst van properties en hun beginwaarden kunt meegeven. Object initializer werkt enkel indien het object een default constructor heeft. Je hoeft deze niet expliciet aan te maken. Indien je klasse geen andere constructors heeft zal er dus een default constructor zijn, zoals ik eerder vertelde.



Bovenstaande code mag ook iets koper nog:

```
1 Meting meting = new Meting { Temperatuur = 3.4,
    IsGeconfermeerd = true};
```

Zie je het verschil? De ronde haakjes van de default constructor mag je dus achterwege laten.

De volgorde waarin je code wordt uitgevoerd is wel belangrijk. Je ziet het niet duidelijk, maar sowieso wordt eerst nu de default constructor aangeroepen. Pas wanneer die klaar is zullen de properties de waarden krijgen die je meegeeft tussen de accolades. Als je dus zelf een default constructor in `Meting` had geschreven dan had eerst die code uitgevoerd zijn geweest. Voorgaande voorbeeld zal intern eigenlijk als volgt plaatsvinden:

```
1 Meting meting = new Meting();  
2 meting.Temperatuur = 3.4;  
3 meting.IsGeconfermeerd = true;
```



Je bent niet verplicht alle properties via deze syntax in te stellen, enkel de zaken die je wilt meegeven tijdens de objectcreatie.

11.3 required properties

Object initializer syntax werd ontwikkeld om de wildgroei aan overloaded constructors in te perken. Echter, dit bracht een nieuw probleem met zich mee. Met behulp van overloaded constructors kan je gebruikers van je klasse verplichten om bepaalde begininformatie van het object bij de creatie mee te geven. Object initializer syntax werkt enkel met een default constructor, en dus was een nieuw keyword vereist. Welkom **required**!

Door **required** voor een property te plaatsen kan je aangeven dat deze property verplicht moet ingesteld worden wanneer je een object aanmaakt met object initializer syntax:

```
1 internal class Meting
2 {
3     public double Temperatuur {get;set;}
4     public required bool IsGeconfermeerd {get;set;}
5 }
```

Wanneer we nu een **Meting** als volgt aanmaken:

```
1 Meting meting = new Meting { Temperatuur = 0.7};
```

Dan krijgen we een foutbericht: *Required member ‘Meting.IsGeconfermeerd’ must be set in the object initializer or attribute constructor.*¹ Enkel als we dus minstens **IsGeconfermeerd** ook instellen zal onze code werken:

```
1 Meting meting = new Meting { IsGeconfermeerd = true};
```



Het **required** keyword werd pas geïntroduceerd in C# 11.0 en zal enkel werken indien je applicaties ontwikkelt in .NET 7 of nieuw.

¹Attribute constructors worden niet in dit boek behandeld.

11.4 Static

Herinner je je dat ik bij de definitie van een klasse het volgende schreef: “95% van de tijd zullen we in dit boek de voorgaande definitie van een klasse beschrijven, namelijk de blauwdruk voor de objecten die er op gebaseerd zijn. Je zou kunnen zeggen dat de klasse een fabriekje is dat objecten kan maken. Echter, wanneer we het **static** keyword zullen bespreken gaan we ontdekken dat heel af en toe een klasse ook als een soort object door het leven kan gaan.”

Laat ik hier eens dieper op ingaan: Je hebt het keyword **static** al een paar keer zien staan aan de start van methodesignaturen. Maar vanaf hoofdstuk 9 werd er dan weer nadrukkelijk verteld géén **static** voor methoden in klassen te plaatsen. *Wat is het nu?*

Bij klassen en objecten duidt **static** aan dat een methode of variabele “gedeeld” wordt over alle objecten van die klasse. Wanneer je **static** ergens voorplaatst dan kan je dit element aanroepen **zonder dat je een instantie van die klasse nodig hebt**.

static kan op verschillende plaatsen in een klasse gebruikt worden:

1. Bij *instantievariabelen* om een gedeelde variabele aan te maken, over de objecten heen. We spreken dan niet meer over een instantievariabele maar over een **static field**.
2. Bij *methoden* om zogenaamde methoden-bibliotheken of hulpmethoden aan te maken (denk maar aan `Math.Pow()` en `DateTime.IsLeap()`). We spreken dan over een **static method**.
3. Bij de klasse zelf om te voorkomen dat er objecten van de klasse aangemaakt kunnen worden (bijvoorbeeld de `Console` en `Math` klasse). Je raadt het nooit, maar dit noemt dan een **static class**. De klasse doet zich dan voor als een uniek *object-achtig* concept.
4. Bij *properties*. We hebben al met 1 **static property** gewerkt namelijk de `readonly` property `Now` van de `DateTime` klasse (`DateTime.Now`).



Ook een constructor kan **static** gemaakt worden, maar dat ga ik in dit boek niet bespreken. Samengevat kan je een *static constructor* gebruiken indien je een soort *oer-constructor* wilt hebben die eenmalig wordt aangeroepen wanneer het allereerste object van een klasse wordt aangemaakt. Wanneer een tweede instantie wordt aangemaakt zal de *static constructor* niet meer aangeroepen worden.



We vereenvoudigen bewust het keyword **static** wat om verwarring te voorkomen. Het “delen van informatie dankzij **static**” is een gevolg, niet de reden. Met **static** geven we eigenlijk aan dat het element bij de klasse zelf behoort én niet bij instanties van die klasse.

11.4.1 Static fields

Zonder het keyword **static** heeft ieder object z'n eigen instantievariabelen. Aanpassingen binnen het object aan die variabelen hebben geen invloed op andere objecten van hetzelfde type. Ik toon je eerst de werking zoals je die gewend bent. Vervolgens leg ik uit hoe **static** in de praktijk werkt.

11.4.1.1 Zonder static fields

Gegeven volgende klasse:

```
1 internal class Mens
2 {
3     private int geboorteJaar;
4     public int Geboortejaar
5     {
6         get { return geboorteJaar; }
7         private set { geboorteJaar = value; }
8     }
9     public void Jarig()
10    {
11        Geboortejaar++;
12    }
13 }
```

Als we dit doen:

```
1 Mens m1 = new Mens();
2 Mens m2 = new Mens();
3 m1.Jarig();
4 m1.Jarig();
5 m2.Jarig();
6 Console.WriteLine($"{m1.Geboortejaar}");
7 Console.WriteLine($"{m2.Geboortejaar}");
```

Dan zien we volgende uitvoer:

```
1 2
2 1
```

Ieder object houdt de stand van z'n eigen variabelen bij. Ze kunnen elkaars interne - zowel publieke als private - staat niet rechtstreeks veranderen.

11.4.1.2 En nu, mét static fields

Laten we eens kijken wat er gebeurt indien we een instantievariabele **static** maken.

We maken de variabele **private int** geboorteJaar static als volgt: **private static int** geboorteJaar = 1;. We krijgen dan:

```

1 internal class Mens
2 {
3     private static int geboorteJaar = 1;
4     public int Geboortejaar
5     {
6         get { return geboorteJaar; }
7         private set { geboorteJaar = value; }
8     }
9     public void Jarig()
10    {
11        Geboortejaar++;
12    }
13 }
```

We hebben er nu voor gezorgd dat ALLE objecten de variabele geboorteJaar delen. Er wordt van deze variabele dus maar één “instantie” in het geheugen aangemaakt.

Voeren we nu terug volgende code uit:

```

1 Mens m1 = new Mens();
2 Mens m2 = new Mens();
3 m1.Jarig();
4 m1.Jarig();
5 m2.Jarig();
6 Console.WriteLine($"{m1.Geboortejaar}");;
7 Console.WriteLine($"{m2.Geboortejaar}");;
```

Dan wordt de uitvoer:

```

1 4
2 4
```

We zien dat de variabele geboorteJaar dus niet meer per object individueel wordt bewaard, maar dat het één globale variabele als het ware is geworden en géén instantievariabele meer is. **static** laat je dus toe om informatie over de objecten heen te delen.



Gebruik static niet te pas en te onpas: vaak druijt het in tegen de concepten van OO en wordt het vooral misbruikt.

Ga je dit soort **static** variabelen -ook wel static fields genoemd - vaak nodig hebben? Niet zo vaak. Het volgende concept daarentegen wel!

11.4.2 Static methoden en klassen

Heb je er al bij stil gestaan waarom je dit kan doen:

```
1 Math.Pow(3,2);
```

Zonder dat we objecten moeten aanmaken in de trend van:

```
1 Math myMath = new Math(); //dit mag niet!
2 myMath.Pow(3,2)
```

De reden dat je de Math-bibliotheek kan aanroepen rechtstreeks **op de klasse** en niet op objecten van die klasse is omdat de methoden in die klasse als **static** gedefinieerd staan.

De klasse is op de koop toe ook zelf **static** gemaakt. Zo kan er zeker geen twijfel bestaan: deze klasse kan niét in een object gegoten worden.

De klasse zal er dus zo ongeveer uitzien:

```
1 internal static class Math
2 {
3     public static double Pow(int getal, int macht)
4     {
5         //enz.
```



Zoals je hopelijk al merkt zijn er aardig wat keywords die je nog voor methode en klasse-definities kunt plaatsen. Zo zijn er al:

- De access modifier zoals **internal** of **private**.
- Het keyword **static** of niet.
- Het returntype bij methoden (bv. **void**, **int**, enz.)

11.4.2.1 Voorbeeld van static methoden

Stel dat we enkele veelgebruikte methoden willen groeperen en deze gebruiken zonder telkens een object te moeten aanmaken dan doen we dit als volgt:

```
1 internal static class EpicLibrary
2 {
3     static public void ToonInfo()
4     {
5         Console.WriteLine("Ik ben ik");
6     }
7
8     static public int TelOp(int a, int b)
9     {
10        return a+b;
11    }
12 }
```

We kunnen deze methoden nu als volgt aanroepen:

```
1 EpicLibrary.ToonInfo();
2 int opgeteld = EpicLibrary.TelOp(3,5);
```

Mooi toch?!

Dankzij **static** kunnen we dus eigen bibliotheken van methoden én properties aanmaken die we kunnen aanroepen rechtstreeks op de klasse. We kunnen deze aanroepen zonder dat er een instantie van de klasse moet zijn.

Je mag ook hybride klassen maken waarin sommige delen **static** zijn en andere niet. De `DateTime` klasse uit het eerste hoofdstuk bijvoorbeeld is zo'n klasse. De meeste dingen gebeurden *non-static* toch was er ook bijvoorbeeld de **static** property `Now` om de huidige tijd terug te krijgen, alsook de `IsLeapYear` hulpmethode die we rechtstreeks op de klasse `DateTime` moesten aanroepen:

```
1 bool gaIkOpPensioenInEenSchrikkeljaar = DateTime.IsLeapYear(2048);
```

11.4.3 Intermezzo: Debug.WriteLine

Even een kort intermezzo dat we in de volgende sectie gaan gebruiken, namelijk de werking van de `Debug` klasse.

De `Debug` klasse (die in de `System.Diagnostics` namespace staat) kan je gebruiken om eenvoudig zaken naar het *debug output* venster te sturen tijdens het debuggen. Dit is handig om te voorkomen dat je debug informatie steeds naar het console-scherm moet sturen. Het zou niet de eerste keer zijn dat iemand vergeet een bepaalde `Console.WriteLine` te verwijderen uit het finale product en zo mogelijk gevoelige debug-informatie naar de eindgebruikers lekt.

Volgende code toont een voorbeeld (merk lijn 1 op die vereist is):

```

1  using System.Diagnostics;
2
3  namespace debugdemo
4  {
5      internal class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World! Console");
10             Debug.WriteLine("Hello World! Debug");
11         }
12     }
13 }
```

Als je voorgaande code uitvoert in debugger modus, dan zal je enkel de tekst Hello World! Console in je console zien verschijnen. De andere lijn kan je terugvinden in het “Output” venster in Visual Studio:

The screenshot shows the Visual Studio IDE. The code editor displays the following C# code:

```

9
10
11
12
13 }
```

The output window below shows the execution of the program:

```

Output
Show output from: Debug
'debugdemo.exe' (CoreCLR: DefaultDomain): Loaded 'C:\Program Files\dotnet\shared\Microsoft
'debugdemo.exe' (CoreCLR: clrhost): Loaded 'd:\Temp\debugdemo\debugdemo\bin\Debug\netcor
'debugdemo.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NET
Hello World! Debug
The program '[18528] debugdemo.exe' has exited with code 0 (0x0).
```

Figuur 11.1: Het is wat zoeken tussen de andere output die VS genereert, daarom heb ik “onze” output even geel gemarkeerd.



Mooi zo. Nu we dat hebben bekeken kunnen terug keren naar het gebruik van het **static** keyword. Of zoals mijn grootvader zaliger altijd zei “goto static!”.

MILJAAR!

11.4.3.1 Nog een voorbeeld van het gebruik van static

In het volgende voorbeeld gebruik ik een **static** variabele om bij te houden hoeveel objecten (via de constructor met behulp van `Debug.WriteLine`) er van de klasse reeds zijn aangemaakt:

```
1 internal class Fiets
2 {
3     private static int aantalFietsen = 0;
4     public Fiets()
5     {
6         aantalFietsen++;
7         Debug.WriteLine($"Er zijn nu {aantalFietsen} gemaakt");
8     }
9     public static void VerminderFiets()
10    {
11        aantalFietsen--;
12        Debug.WriteLine($"STATIC: Er zijn {aantalFietsen} fietsen");
13    }
14 }
```

Merk op dat we de methode `VerminderFiets` enkel via de klasse kunnen aanroepen daar deze **static** werd gemaakt. We kunnen echter nog steeds `Fiets`-objecten aanmaken aangezien de klasse zelf niet **static** werd gemaakt.

Laten we de uitvoer van volgende code eens bekijken:

```
1 Fiets merckx = new Fiets();
2 Fiets steels = new Fiets();
3 Fiets evenepoel = new Fiets();
4 Fiets.VerminderFiets();
5 Fiets aerts = new Fiets();
6 Fiets.VerminderFiets();
```

Dit zal debug uitvoer geven:

```
1 Er zijn nu 1 gemaakt
2 Er zijn nu 2 gemaakt
3 Er zijn nu 3 gemaakt
4 STATIC:Er zijn 2 fietsen
5 Er zijn nu 3 gemaakt
6 STATIC:Er zijn 2 fietsen
```

11.4.4 Static tegenover non-static

Van zodra je een methode hebt die **static** is dan zal deze methode enkel andere **static** methoden en variabelen kunnen aanspreken. Dat is logisch: een **static** methode heeft geen toegang tot de gewone niet-statistische variabelen van een individueel object, want welk object zou hij dan moeten benaderen? Het omgekeerde kan nog wel natuurlijk.

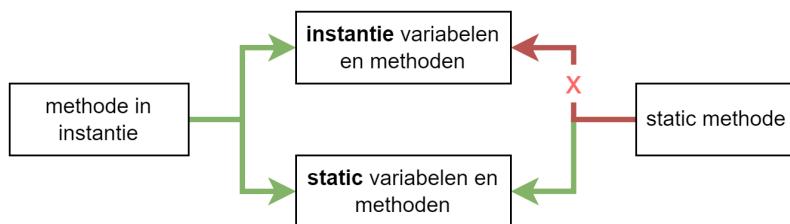
Volgende code zal dus een fout geven:

```

1  internal class Mens
2  {
3      private int gewicht = 50;
4
5      private static void VerminderGewicht()
6      {
7          gewicht--;
8      }
9  }
```

De foutbericht die verschijnt “**An object reference is required for the non-static field, method, or property ‘Program.Mens.gewicht’**” zal bij lijn 7 staan.

Volgende vereenvoudiging maakt duidelijk wat kan aangeroepen worden en wat niet:



Figuur 11.2: De pijl duidt aan of methoden en variabelen kunnen bereikt worden of niet.



Een eenvoudige regel is te onthouden dat van zodra je in een **static** omgeving bent, je niet meer naar de niet-static delen van je code zal geraken.

Dit verklaart ook waarom je bij console applicaties in Program.cs steeds alle methoden **static** moet maken. De Main van een console-applicatie is namelijk als volgt beschreven:**public static void Main() {}**. Zoals je ziet is de Main methode als **static** gedefinieerd. Willen we dus vanuit deze methode andere methoden aanroepen dan moeten deze als **static** aangeduid zijn..

11.4.5 Static properties

Beeld je in dat je een pong-variant moet maken waarbij meerdere balletjes over het scherm moeten botsen. Je wilt echter niet dat de balletjes zelf allemaal apart moeten weten wat de grenzen van het scherm zijn. Mogelijk wil je bijvoorbeeld dat je code ook werkt als het speelveld kleiner is dan het eigenlijke Console-scherm.

We gaan dit oplossen met een static property waarin we de grenzen voor alle balletjes bijhouden. Aan onze klasse `Balletje` voegen we dan alvast het volgende toe:

```
1 static public int Breedte { get; set; }
2 static public int Hoogte { get; set; }
```

In ons hoofdprogramma (`Main`) kunnen we nu de grenzen voor alle balletjes tegelijk vastleggen:

```
1 Balletje.Hoogte = Console.WindowHeight;
2 Balletje.Breedte = Console.WindowWidth;
```

Maar even goed maken we de grenzen voor alle balletjes gebaseerd op zelf gekozen waarden:

```
1 Balletje.Hoogte = 20;
2 Balletje.Breedte = 10;
```



We zouden zelfs de grenzen van het veld dynamisch kunnen maken en laten afhangen van het huidige level.

De interne werking van de balletjes hoeft dus geen rekening meer te houden met de grenzen van het scherm. We passen de `Update`-methode aan, rekening houdend met deze nieuwe kennis:

```
1 public void Update()
2 {
3     if (X + VX >= Balletje.Breedte || X + VX < 0)
4     {
5         VX = -VX;
6     }
7     X = X + VX;
8
9     if (Y + VY >= Balletje.Hoogte || Y + VY < 0)
10    {
11        VY = -VY;
12    }
13    Y = Y + VY;
14 }
```

En nu kunnen we vlot balletjes laten rond bewegen op bijvoorbeeld een klein deeltje maar van het scherm:

```
1 static void Main(string[] args)
2 {
3     Console.CursorVisible = false;
4     Balletje.Hoogte = 15;
5     Balletje.Breedte = 15;
6
7     Balletje m1 = new Balletje(1,1,1,1);
8     Balletje m2 = new Balletje(2,2,-2,1);
9
10    while (true)
11    {
12        m1.Update();
13        m1.TekenOpScherm();
14
15        m2.Update();
16        m2.TekenOpScherm();
17
18        System.Threading.Thread.Sleep(50);
19        Console.Clear();
20    }
21 }
```

11.4.6 static en Random



Je zal **static** minder vaak nodig hebben dan non-static zaken. Alhoewel: wanneer je werkt met een klasse waarin je een Random-number generator gebruikt, dan is het een goede gewoonte deze generator **static** te maken zodat alle objecten deze ene generator gebruiken. Anders bestaat de kans dat je objecten dezelfde random getallen zullen aanmaken wanneer ze toevallig op quasi hetzelfde moment werden geïnstantieerd of methoden in aanroeft.

Test maar eens wat er gebeurt als je volgende klasse hebt:

```
1 internal class Dobbelsteen
2 {
3     public int Werp()
4     {
5         Random gen = new Random(); //SLECHT IDEE!
6         return gen.Next(1,7);
7     }
8 }
```

Wanneer je nu dezelfde dobbelsteen 10 maal snel na elkaar rolt is de kans groot dat je geregeld dezelfde getallen gooit:

```
1 Dobbelsteen testDobbel = new Dobbelsteen();
2 for(int i = 0 ; i < 10; i++)
3 {
4     Console.WriteLine(testDobbel.Werp());
5 }
```

De reden? Een nieuw aangemaakt Random-object gebruikt de tijd waarop het wordt aangemaakt als een zogenaamde seed. Een seed zorgt ervoor dat je dezelfde reeks getallen kan genereren wanneer de seed dezelfde is -een concept dat nuttig is in cryptografie. Uiteraard willen we dat niet bij een dobbelsteen. Het is niet omdat een dobbelsteen snel na elkaar wordt geworpen (of aangemaakt) dat die dobbelsteen dan regelmatig dezelfde getallen na elkaar gooit.

We lossen dit op door de generator static te maken zodat er maar één generator bestaat die alle dobbelstenen en hun methoden delen. Dit is erg eenvoudig opgelost: je verhuist je generator naar buiten de methode en plaatst er **static** voor:

```
1 internal class Dobbelsteen
2 {
3     static Random gen = new Random();
4     public int Werp()
5     {
6         return gen.Next(1,7);
7     }
8 }
```

12 Arrays en klassen

Arrays van value types kwamen we al in hoofdstuk 8 tegen. In dit hoofdstuk demonstreer ik dat ook arrays van objecten perfect mogelijk zijn. We weten reeds dat klassen niets meer dan zijn dan nieuwe datatypes. Het is dus ook logisch dat wat we reeds met arrays konden, we dit kunnen blijven doen met objecten.

Maar, er is één grote maar: omdat we met objecten werken moeten we rekening houden met het feit dat de individuele objecten in je array **reference values** hebben en dus mogelijk **null** zijn. Met andere woorden: het is van essentieel belang dat je het hoofdstuk rond geheugenmanagement in C# goed begrijpt, want we gaan het gereeld nodig hebben.

Na Exceptions is het weer tijd voor een andere bekentenis: arrays zijn fijn, maar nogal omslachtig qua gebruik. Er zit echter in .NET een soort *array on steroids* datatype dat ons nooit nog zal doen teruggrijpen naar arrays. Welke dat zijn? Lees verder en ontdek het zelf!

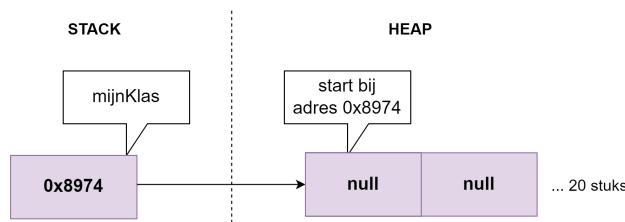
Let's go!

12.1 Arrays van objecten aanmaken

Een array van objecten aanmaken doe je als volgt:

```
1 Student[] mijnKlas = new Student[20];
```

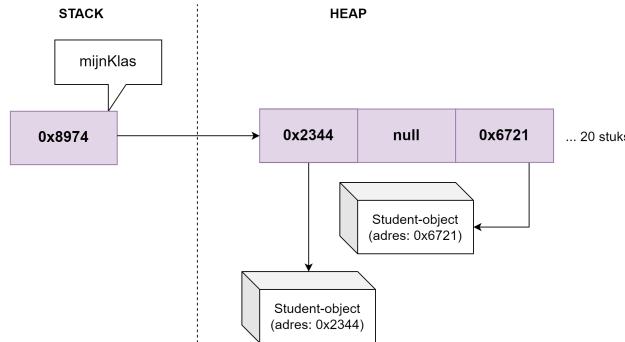
De **new** zorgt er echter enkel voor dat er een referentie naar een nieuwe array wordt teruggegeven, waar ooit 20 studenten-objecten in kunnen komen. **Maar: er staan nog géén objecten in deze array. Alle elementen in deze array zijn nu nog null.**



Figuur 12.1: De referentie naar een, nu nog, lege array is aangemaakt.

Willen we nu elementen in deze array plaatsen dan moeten we dit ook explicet doen en moeten we dus objecten aanmaken en hun referentie in de array bewaren:

```
1  mijnKlas[0] = new Student();
2  mijnKlas[2] = new Student();
```



Figuur 12.2: De situatie in het geheugen nadat 2 objecten werden aangemaakt en in de array werden geplaatst.

Uiteraard kan dit ook in een loop indien relevant voor de opgave. Volgende voorbeeld vult een reeds aangemaakte array met evenveel objecten als de arrays groot is:

```
1  for(int i = 0; i < mijnKlas.Length; i++)
2  {
3      mijnKlas[i] = new Student();
4 }
```

12.1.1 Individueel object benaderen

Van zodra een object in de array staat kan je deze vanuit de array aanspreken door middel van de index en de *dot*-operator om de juiste methode of property op het object aan te roepen:

```
1  mijnKlas[3].Name = " Duke Peekaboo";
```

Uiteraard mag je ook altijd de referentie naar een individueel object in de array kopiëren. Denk er aan dat we de hele tijd met referenties werken en de GC dus niet tussenbeide zal komen zolang er minstens 1 referentie naar het object is. Indien de student op plek 4 in de array aan de start een geboortejaar van 1981 had, dan zal deze op het einde van volgende code als geboortejaar 1983 hebben, daar we op hetzelfde objecten het geboortejaar verhogen in zowel lijn 2 als 3:

```
1  Student tijdelijkeStudent = mijnKlas[3];
2  mijnKlas[3].Geboortejaar++;
3  tijdelijkeStudent.Geboortejaar++;
```



Probeer je objecten te benaderen die nog niet bestaan dan zal je uiteraard een `NullReferenceException` krijgen.

12.1.2 Array initializer syntax

Je kan ook een variant op de object initializer syntax gebruiken waarbij de objecten reeds van bij de start in de array worden aangemaakt. Als bonus zorgt dit er ook voor dat we geen lengte moeten meegeven, de compiler zal deze zelf bepalen.

Volgende voorbeeld maakt een nieuwe array aan die bestaat uit 2 nieuwe studenten, alsook 1 bestaande met de naam `jos`:

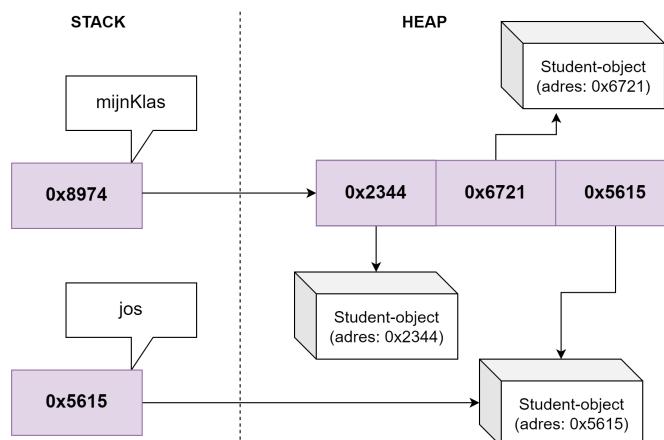
```

1 Student jos = new Student();
2 //...
3
4 Student[] mijnKlas = new Student[]
5 {
6     new Student(),
7     new Student(),
8     jos
9 };

```

Let op de puntkomma helemaal achteraan. Die wordt als eens vergeten.

Het kan niet genoeg benadrukt worden dat een goede kennis van de heap, stack en referenties essentieel is om te leren werken met arrays van objecten. Uit voorgaande stukje code zien we duidelijk dat een goed inzicht in referenties je van veel leed beschermen. Bekijk eens de eindsituatie van voorgaande code:



Figuur 12.3: De situatie in het geheugen op het einde.

Zoals je merkt zal nu de student `jos` niet verwijderd worden indien we op gegeven moment

schrijven `jos = null` daar het object nog steeds bestaat via de array. We kunnen met andere woorden op 2 manieren de student `jos` momenteel bereiken, via de array of via `jos`:

```
1 jos.Naam = "Joske Vermeulen";
2 mijnKlas[2].Naam = "Franske Vermeulen"; //we overschrijven "Joske
Vermeulen"
```

12.1.2.1 Null-check met ?

Ook hier kan je met `?` een null-check schrijven:

```
1 mijnKlas[3]?.Name = "Romeo Montague ";
```

Merk op dat het eerste vraagteken controleert of de array zelf niet `null` is.

12.1.3 Object arrays als parameters en return

Ook arrays mag je als parameters en returntype gebruiken in methoden. De werking hiervan is identiek aan die van value-types zoals volgende voorbeeld toont. Eerst maken we een methode die als resultaat een referentie naar een lege array van 10 studenten teruggeeft.

```
1 static Student[] CreateEmptyStudentArray()
2 {
3     return new Student[10];
4 }
```

Vervolgens kunnen we deze dan aanroepen en het resultaat (de referentie naar de lege array) toewijzen aan een nieuwe variabele (van hetzelfde datatype, namelijk `Student []`):

```
1 Student[] resultaat = CreateEmptyStudentArray();
```

12.2 List collectie

Een `List<>`-collectie is de meest standaard collectie die je kan beschouwen als een veiligere variant op een doodnormale array. Een `List` heeft alle eigenschappen die we al kennen van arrays, maar ze zijn wel krachtiger. Het giet een klasse “rond” het concept van de array, waardoor je toegang krijgt tot een hoop nuttige methoden die het werken met arrays vereenvoudigen.

12.2.1 List aanmaken

De klasse `List<>` is een generieke klasse. Tussen de `< >`tekens plaatsen we het datatype dat de lijst zal moeten gaan bevatten. Bijvoorbeeld:

```
1 List<int> alleGetallen = new List<int>();  
2 List<bool> binaryList = new List<bool>();  
3 List<Pokemon> pokeDex = new List<Pokemon>();  
4 List<string[]> listOfStringarrays = new List<string[]>();
```

Zoals je ziet hoeven we bij het aanmaken van een `List` geen begin grootte mee te geven, wat we wel bij arrays moeten doen. Dit is één van de voordelen van `List`: ze groeien mee.



In dit boek behandel ik het concept generieke klassen enkel in de appendix. Generieke klassen oftewel **generic classes** zijn een handig concept om je klassen nog multifunctioneler te maken doordat we zullen toelaten dat bepaalde datatypes niet hardcoded in onze klasse moet gezet worden. `List<>` is zo'n eerste voorbeeld, maar er zijn er tal van anderen én je kan ook zelf dergelijke klassen schrijven. Bekijk zeker de appendix indien je dit interesseert.



De generieke `List<>` klasse bevindt zich in de `System.Collections.Generic` namespace. Je dient deze namespace dus als **using** bovenaan toe te voegen wil je deze klasse kunnen gebruiken in C# 9.0 en ouder.

12.2.2 Elementen toevoegen

Via de Add () -methode kan je elementen toevoegen aan de lijst. Je dient als parameter aan de methode mee te geven wat je aan de lijst wenst toe te voegen. **Deze parameter moet uiteraard van het type zijn dat de List verwacht.**

In volgende voorbeeld maken we een List aan die objecten van het type string mag bevatten en vervolgens plaatsen we er twee elementen in.

```
1 List<string> mijnPersonages = new List<string>();  
2 mijnPersonages.Add("Reinhardt");  
3 mijnPersonages.Add("Mercy");
```

Ook meer complexe datatypes kan je dus toevoegen:

```
1 List<Pokemon> pokedex = new List<Pokemon>();  
2 pokedex.Add(new Pokemon());
```

Via object syntax initializer kan dit zelfs nog sneller:

```
1 List<Pokemon> pokedex = new List<Pokemon>()  
2 {  
3     new Pokemon(),  
4     new Pokemon()  
5 };
```



Je kan ook een stap verder gaan en ook binnenin deze initializer syntax dezelfde soort initialize syntax gebruiken om de objecten individueel aan te maken:

```
1 List<Pokemon> pokedex = new List<Pokemon>()  
2 {  
3     new Pokemon() {Naam = "Pikachu", HP_Base = 5},  
4     new Pokemon() {Naam = "Bulbasaur", HP_Base = 15}  
5 }
```

12.2.3 Elementen indexeren

Het leuke van een List is dat je deze ook kan gebruiken als een gewone array, waarbij je met behulp van de indexer elementen individueel kan aanroepen. Stel bijvoorbeeld dat we een lijst hebben met minstens 4 strings in. Volgende code toont hoe we de string op positie 3 kunnen uitlezen en hoe we die op positie 2 overschrijven, net zoals we reeds kenden van arrays:

```
1 Console.WriteLine(mijnPersonages[3]);  
2 mijnPersonages[2] = "Torbjorn";
```

Ook de klassieke werking met loops blijft gelden. **De enige aanpassing is dat List<> niet met Length werkt maar met Count:**

```
1 for (int i = 0 ; i < mijnPersonages.Count; i++)  
2 {  
3     Console.WriteLine(mijnPersonages[i])  
4 }
```

12.2.4 Wat kan een List nog?

Interessante methoden en properties voorts zijn:

- **Clear()**: methode die de volledige lijst leegmaakt en de lengte (Count) terug op 0 zet.
- **Insert()**: methode om een element op een specifieke plaats in de lijst in te voegen.
- **IndexOf()**: geeft de index terug van het element item in de rij. Indien deze niet in de lijst aanwezig is dan wordt -1 teruggegeven.
- **RemoveAt()**: verwijdert een element op de index die je als parameter meegeeft.
- **Sort()**: alle elementen in de lijst worden gesorteerd. Merk op dat dit niet altijd werkt zoals je verwacht. Lees zeker in hoofdstuk 17 de sectie omtrent “Interfaces in de praktijk” eerst voor je probeert een lijst van eigen objecten te sorteren.



Let op met het gebruik van **IndexOf** en objecten. Deze methode zal controleren of de referentie dezelfde is van een bepaald object en daar de index van teruggeven. Je kan deze methode dus wel degelijk met arrays van objecten gebruiken, maar je zal enkel je gewenste object terugvinden indien je reeds een referentie naar het object hebt en dit meegeeft als parameter.

12.2.5 Een wereld met OOP: Pong list

Ikzelf ben fan van List. Het maakt je code vaak leesbaarder dan arrays. Voorts geeft het je de optie om dynamisch groeiende (en krimpende) arrays te hebben, zonder dat je daar veel *boilerplate* code voor moet schrijven. Herinner je onze Pong-code waarin we 100 balletjes op het scherm lieten vliegen?

```
1 const int AANTAL_BALLETJES = 100;
2 Random r = new Random();
3 Balletje[] veelBalletjes = new Balletje[AANTAL_BALLETJES];
4 for (int i = 0; i < veelBalletjes.Length; i++) //balletjes aanmaken
5 {
6     veelBalletjes[i] = new Balletje();
7     veelBalletjes[i].X = r.Next(10, 20);
8     veelBalletjes[i].Y = r.Next(10, 20);
9     veelBalletjes[i].VX = r.Next(-2, 3);
10    veelBalletjes[i].VY = r.Next(-2, 3);
11 }
12
13 while (true)
14 {
15     for (int i = 0; i < veelBalletjes.Length; i++)
16     {
17         veelBalletjes[i].Update(); //update alle balletjes
18     }
19     for (int i = 0; i < veelBalletjes.Length; i++)
20     {
21         veelBalletjes[i].TekenenOpScherm(); //teken alle balletjes
22     }
23     System.Threading.Thread.Sleep(50);
24     Console.Clear();
25 }
```

Vooral de code in de **while** wordt nu leesbaarder dankzij **List<Balletje>** (we gaan ook ineens gebruik maken van onze nieuwe default constructor die de random startwaarde instelde):

```
1 const int AANTAL_BALLETJES = 100;
2 List<Balletje> veelBalletjes = List<Balletje>();
3 for (int i = 0; i < AANTAL_BALLETJES; i++) //balletjes aanmaken
4 {
5     veelBalletjes.Add(new Balletje());
6 }
7
8 while (true)
9 {
10    foreach(var bal in veelBalletjes)
11    {
12        bal.Update(); //update alle balletjes
13    }
14    foreach(var bal in veelBalletjes)
15    {
16        bal.TekenOpScherm(); //teken alle balletjes
17    }
18    System.Threading.Thread.Sleep(50);
19    Console.Clear();
20 }
```

Deze code zou je aan iemand die geen C# kan kunnen tonen en met een beetje geluk zal die de code begrijpen. Dat is het fijne van hogere programmeertalen zoals C#: ze zijn veel leesbaarder dan talen die *dichter tegen het metaal zitten*, zoals C en C++.

Als leuke extra bij C# is dat het een erg levende taal is. Jaarlijks komen er nog nieuwe concepten bij. Meestal zijn die ietwat obscuur, maar vaak maken ze de code wel een pak leesbaarder dan ervoor. Alhoewel ik graag werk met arrays, zorgen Lists er bijvoorbeeld voor dat we veel minder met vierkante haakjes moeten werken én verstoppen ze een hoop code om bijvoorbeeld lijsten te doen groeien en krimpen.

Dit verstoppen kan uiteraard soms een probleem zijn indien je hoog-performante code moet schrijven. Aan de andere kant: denk je dat jij betere code kunt schrijven dan de ontwikkelaars van de Add-methode bij List?

12.3 Foreach loops

In het hoofdstuk over loops besprak ik reeds de **while**, **do while** en **for**-loops. Er is echter een vierde soort loop in C# die vooral zijn nut zal bewijzen wanneer we met arrays van objecten werken: de **foreach** loop.

Wanneer je geen indexering nodig hebt, maar wel over **alle elementen** in een array wenst te gaan, dan is de **foreach** loop nuttig. Een **foreach** loop zal ieder element in de array één voor één in een tijdelijke variabele plaatsen (de **iteration variable**). Vervolgens kan binnenin de loop iedere iteratie met die iteration variabele gewerkt worden. Het voordeel hierbij is dat je geen teller nodig hebt en dat de loop zelf de lengte van de array zal bepalen: *je code wordt net iets leesbaarder*. Zeker als we dit bijvoorbeeld vergelijken met hoe een **for** loop geschreven is.

Volgende code toont de werking waarbij we een **double**-array hebben en alle elementen ervan op het scherm willen tonen:

```
1 double[] metingen = {1.2, 0.89, 3.15, 0.1};
2 foreach (double meting in metingen)
3 {
4     Console.WriteLine(meting);
5 }
```

Het belangrijkste nieuwe concept is de **iteration variable** die we hier definiëren als `meting`. Deze moet van het type zijn van de individuele elementen in de array. De naam die je aan de iteration variabele geeft mag je zelf kiezen. Vervolgens schrijven we het nieuwe keyword **in** gevuld door de array waar we over wensen te itereren.

De eerste keer dat we in de loop gaan, zal het element `metingen[0]` aan `meting` toegewezen worden voor gebruik in de loop-body. Vervolgens wordt `metingen[1]` toegewezen, enz.

De output zal dan zijn:

```
1 1.2
2 0.89
3 3.15
4 0.1
```

Stel dat we een array van Studenten hebben, `deKlas`, en wensen van deze studenten de naam en geboortejaar op het scherm te tonen. Ook dit kan dan met een **foreach** erg eenvoudig:

```
1 foreach (Student student in deKlas)
2 {
3     Console.WriteLine($"{student.Naam}, {student.Geboortejaar}");
4 }
```

Merk op dat al deze voorbeelden ook met een List in plaats van een array werken.

12.3.1 Opgelet bij het gebruik van foreach loops

De foreach loop is weliswaar leesbaarder en eenvoudiger in gebruikt, er zijn ook 3 erg belangrijke nadelen aan:

- De foreach iteration variabele is *read-only*: je kan dus geen waarden in de array aanpassen, enkel uitlezen. Dit ogenschijnlijk eenvoudige zinnetje heeft echter veel gevolgen. Je kan met een **foreach**-loop dus **nooit de inhoud van de variabele aanpassen** (lees zeker de waarschuwing hieronder). Wens je dat wel te doen, dan dien je de klassieke **while**, **do while** of **for** loops te gebruiken.
- De foreach loop gebruik je enkel als je **alle elementen van een array wenst te benaderen**. In alle andere gevallen zal je een ander soort loop moeten gebruiken (daar ik geen fan van **break** ben).
- Voorts heb je geen teller (die je gratis bij een **for** krijgt) om bij te houden hoeveel objecten je al hebt benaderd. Heb je dus een teller nodig dan zal je deze manueel moeten aanmaken zoals je ook bij een **while** en **do while** loop moet doen.



Het feit dat de foreach iteration variabele *read-only* is wil niet zeggen dat we de inhoud van het onderliggend object niet kunnen aanpassen. De iteration variabele krijgt bij een array van objecten telkens een referentie naar het huidige element. **Deze referentie kunnen we niet aanpassen**, maar we mogen wel de referentie “volgen” om vervolgens iets in het huidige object zelf aan te passen.

Dit mag dus niet:

```
1 foreach (Student eenStudent in deKlas)
2 {
3     eenStudent = new Student();
4 }
```

Maar dit mag wél:

```
1 foreach (Student eenStudent in deKlas)
2 {
3     eenStudent.Geboortejaar++;
4 }
```



Met de VS snippet **foreach** gevolgd door twee maal op de tab-toets te duwen krijg je een kant-en-klare **foreach** loop.

12.4 Het var keyword

C# heeft een **var** keyword. Je mag dit keyword gebruiken ter vervanging van het datatype op voorwaarde dat de compiler kan achterhalen wat het type (*implicit type*) moet zijn. De compiler kan het type ontdekken aan de hand van de expressie rechts van de toekenningsoperator.

```
1 var getal = 5; //var zal int zijn
2 var myArray = new double[20]; //var zal double[] zijn
3 var tekst = "Hi there handsome"; //var zal string zijn
4 var ikke = new Leerkracht(); //var zal Leerkracht zijn
```



Opgelet: het **var** keyword is gewoon een *lazy programmer syntax toevoeging* om te voorkomen dat je als programmeur niet constant het type moet schrijven.

Bij JavaScript heeft **var** een totaal andere functie, daar zegt het eigenlijk: “het type dat je in deze variabele kan steken is...variabel”. Met andere woorden, het kan de ene keer **string** zijn, dan **int**, enz.

Bij C# gaat dit niet: eens je een variabele aanmaakt dan zal dat type onveranderbaar zijn en kan je er alleen waarden aan toekennen van dat type.

JavaScript is namelijk een *dynamically typed language*. C# is daarentegen een *statically typed language*. Er is één uitzondering bij C#: wanneer je met **dynamic** leert werken kan je C# ook tijdelijk als een *dynamically typed* taal gebruiken (maar dat wordt niet besproken in dit boek).

12.5 var en foreach

Wanneer je de Visual Studio code snippet voor **foreach** gebruikt (**foreach** [tab] [tab]) dan zal deze code ook een **var** gebruiken voor de iteration variabele. De compiler kan aan de te gebruiken array of List zien wat het type van een individueel element in de array moet zijn.

De foreach die we zonet gebruikten kan dus herschreven worden naar:

```
1 foreach (var student in deKlas)
2 {
3     Console.WriteLine($"{student.Naam}, {student.Geboortejaar}");
4 }
```

Merk op dat dit hoegenaamd geen invloed heeft op je applicatie. Wanneer je code gaat compileren die het keyword **var** bevatten dan zal de compiler eerst alle *vars* vervangen door het juiste type, én dan pas beginnen compileren.

12.6 Nuttige collectie-klassen

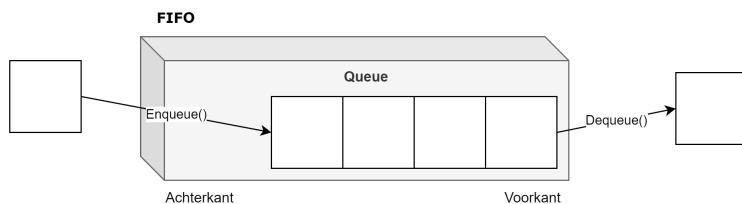
Naast de generieke List collectie, zijn er nog enkele andere nuttige generieke ‘collectie-klassen’ die je geregeld in je projecten kan gebruiken. We bespreken nu de Dictionary, Queue en Stack-collecties.

12.6.1 Queue<> collectie

Een queue (uitgesproken als *kjoe*) stelt een “first in, first out”-lijst (FIFO) voor. Een Queue stelt de rijen voor die we in het echte leven ook hebben wanneer we bijvoorbeeld aanschuiven aan een ticketverkoop of in de supermarkt. Met deze klasse kunnen we zo’n rij simuleren en ervoor zorgen dat steeds het eerste/oudste element in de rij als eerste wordt behandeld. Nieuwe elementen worden achteraan de rij toegevoegd.

We gebruiken onder andere volgende 2 methoden¹ om met een Queue-lijst te werken:

- Enqueue(T item): Voeg een item achteraan de lijst toe.
- Dequeue(): geeft een referentie naar het eerste element in de queue terug en verwijdert dit vervolgens.



Figuur 12.4: De Queue: een wachtrij van objecten en een verdomd moeilijk woord om te schrijven.

Voorbeeld:

```

1 Queue<string> wachtrij = new Queue<string>();
2 wachtrij.Enqueue("Ik stond hier eerste.");
3 wachtrij.Enqueue("Ik tweedes.");
4 wachtrij.Enqueue("Ik laatste.");
5 Console.WriteLine(wachtrij.Dequeue());
6 Console.WriteLine(wachtrij.Dequeue());

```

Dit zal op het scherm tonen:

```

1 Ik stond hier eerste.
2 Ik tweedes.

```

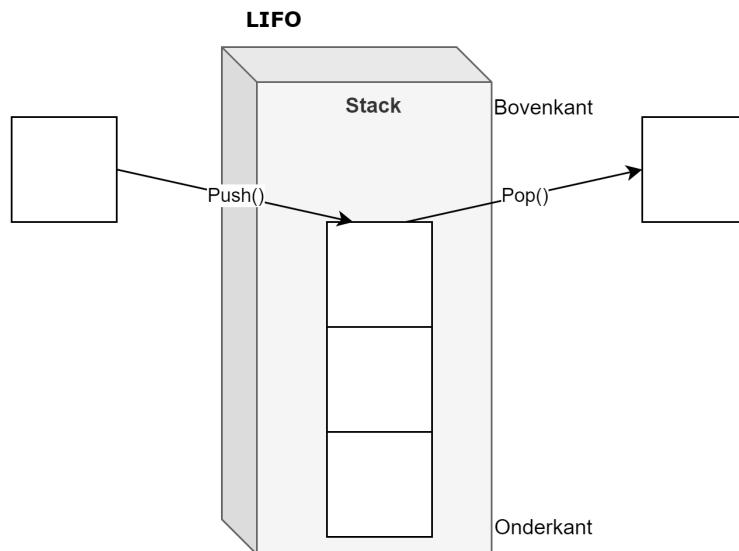
¹Een andere interessante methode is **Peek()**: hiermee kunnen we kijken in de queue wat het eerste element is, zonder het te verwijderen.

12.6.2 Stack<> collectie

Daar waar een queue “first in,first out” is, is een stack “last in,first out” (LIFO). Met andere woorden het recentst toegevoegde element zal steeds vooraan staan en als eerste verwerkt worden. Je kan dit vergelijken met een stapel papieren waar je steeds bovenop een nieuw papier legt.

Ook de klasse Stack heeft verschillende methoden, waarvan volgende 2 methoden het interessantst zijn:

- Push(T item): plaats een nieuw element bovenop de stapel.
- Pop(): geeft het bovenste element in de stack terug en verwijdert dit vervolgens.



Figuur 12.5: De stack: een toren van objecten

Voorbeeld:

```

1 Stack<string> stapel = new Stack<string>();
2 stapel.Push("Ik was eerste hier.");
3 stapel.Push("Ik tweede.");
4 stapel.Push("Ik als laatste.");
5
6 Console.WriteLine(stapel.Pop());
7 Console.WriteLine(stapel.Pop());

```

Dit zal dus het volgende resultaat geven:

```

1 Ik als laatste.
2 Ik tweede.

```

12.6.3 Dictionary<> collectie

In een **dictionary** wordt ieder element voorgesteld door een sleutel (**key** of index) en een waarde (**value**).

De sleutel moet een unieke waarde zijn zodat het element kan opgevraagd worden uit de dictionary aan de hand van deze sleutel zonder dat er duplicaten zijn.

Bij de declaratie van de **Dictionary** dien je op te geven wat het datatype van de key zal zijn, alsook het type van de waarde (**value**).

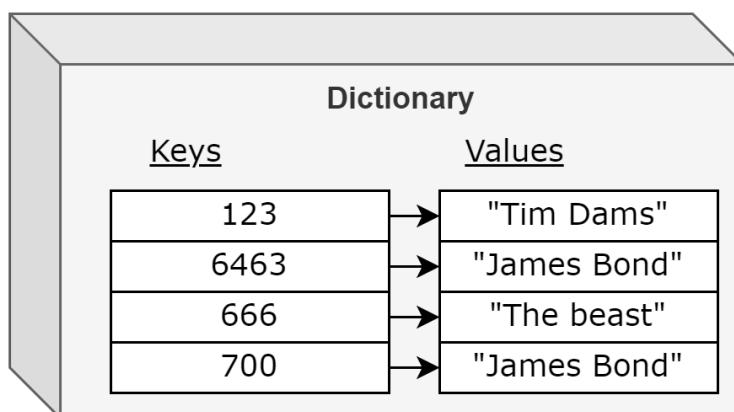


De **Dictionary**-klasse emuleert dus letterlijk de werking van een woordenboek, waarbij ieder woord uniek is en een bijhorende uitleg heeft. Het woord is de sleutel, de bijhorende uitleg is de waarde.

12.6.3.1 Gebruik Dictionary

In het volgende voorbeeld maken we een **Dictionary** van klanten aan. Iedere klant heeft een unieke ID (de key is van het type **int**) alsook een naam (die niet noodzakelijk uniek is en de waarde voorstelt):

```
1 Dictionary<int, string> klanten = new Dictionary<int, string>();
2 klanten.Add(123, "Tim Dams");
3 klanten.Add(6463, "James Bond");
4 klanten.Add(666, "The beast");
5 klanten.Add(700, "James Bond");
```



Figuur 12.6: Visuele voorstelling van de net aangemaakte Dictionary

Bij de declaratie van **klanten** plaatsen we dus tussen de < > twee datatypes: het eerste duidt het datatype van de key aan, het tweede dat van de values.

We kunnen een specifiek element opvragen aan de hand van de key. Stel dat we de waarde (naam) van de klant met key (id) gelijk aan 123 willen tonen, dan schrijven we:

```
1 Console.WriteLine(klanten[123]);
```

We kunnen nu met behulp van bijvoorbeeld een **foreach**-loop alle elementen tonen. Hier kunnen we de key met de .Key-property uitlezen en het achterliggende object of waarde met .Value. Value en Key hebben daarbij ieder het type dat we hebben gedefinieerd toen we het Dictionary-object aanmaakten, in het volgende geval is de Key dus van het type **int** en Value van het type **string**:

```
1 foreach (var item in klanten)
2 {
3     Console.WriteLine(item.Key + "\t:" + item.Value);
4 }
```

De key werkt dus net als de index bij gewone arrays. Alleen heeft de key nu geen relatie meer met de positie van het element in de collectie, maar is een unieke identifier van het element in kwestie.

12.6.3.2 Eender welk type voor key en value

De key kan zelfs een **string** zijn en de waarde een ander type. In het volgende voorbeeld hebben we eerder een klasse Student aangemaakt. We maken nu een student aan en voegen deze toe aan de studentenLijst. Vervolgens willen we het geboortejaar van een bepaalde student tonen op het scherm en vervolgens verwijderen we deze student:

```
1 var studentenLijst = new Dictionary<string, Student>();
2 Student stud = new Student() { Naam = "Tim", Geboortejaar = 2001 };
3 studentenLijst.Add("AB12", stud);
4 Console.WriteLine(studentenLijst["AB12"].Geboortejaar);
5 studentenLijst.Remove("AB12");
```



Lijn1 is zo'n typisch voorbeeld waar het gebruik van het keyword **var** effectief een meerwaarde heeft. Het zorgt ervoor dat de code mooi op 1 lijn past en leesbaar blijft.

13 Overerving

Het wordt tijd om de derde letter in het acroniem *A P/E* aan te pakken. We hadden reeds **abstractie** en **encapsulatie** bekeken. Nu is het de beurt aan **inheritance**, oftewel overerving.

Programmeurs zijn luie wezens. Ieder concept dat hen toelaat minder code te schrijven zullen ze dan ook omarmen. Dubbele code wil namelijk ook zeggen dat er dubbel zoveel plekken zijn waar bugs kunnen optreden én die aangepast moeten worden wanneer de specificaties veranderen.

Indien 2 of meer klassen een aantal gelijkaardige stukken code hebben is er mogelijk een verband tussen die twee klassen. Denk maar aan de klassen `Monster` en `Held` in een avonturenspel. Beide klassen hebben vermoedelijk bepaalde properties en methoden die identiek, of bijna identiek zijn qua implementatie.

Wat we hier zien is het concept **overerving**. Beide klassen hebben duidelijk een soort gemeenschappelijke “voorouder”. Net zoals in de natuur waar apen en mensen afstammen van een gemeenschappelijke voorouder, kan je dit concept ook in OOP hebben.

De zogenaamde “child-klasse” is de klasse die overerft van een “parent-klasse”. Deze child-klasse zal een specialisatie zijn: het zal meer kunnen dan z’n parent. Ook in de natuur zien we dit. De homo sapiens sapiens (wij!) is evolutionair gezien een verbetering tegenover de homo erectus (kleinere hersenen). De homo erectus is op zijn beurt een verbetering van zijn voorouder: de homo habilis kon nog niet op 2 ledematen rondwandelen.

Kijken we terug naar `Monster` en `Held` dan is het duidelijk dat een gemeenschappelijke parent-klasse misschien wel de klasse `Karakter` is.

Dankzij overerving kunnen we de gemeenschappelijk code van de child-klassen verhuizen naar deze parent-klasse. De child-klassen zullen enkel nog de code bevatten die uniek is voor hen: de code die de specialisatie beschrijft.



Deze introductie doet uitschijnen dat overerving enkel z’n nut heeft om *dubbele code* te vermijden, wat niet zo is. Dubbele code vermijden via overerving is eerder een gevolg ervan. Overerving is een erg krachtig concept dat in de komende hoofdstukken telkens zal terugkomen bij polymorfisme, interfaces, enz.

13.1 Wat is overerving

Overerving (**inheritance**) laat ons toe om klassen te specialiseren vanuit een reeds bestaande parent- of basisklasse. Wanneer we een klasse van een andere klasse overerven dan zeggen we dat deze nieuwe klasse een child-klasse of sub-klasse is van de bestaande parent-klasse of super-klasse.

De child-klasse kan alles wat de parent-klasse kan, maar de nieuwe klasse kan nu ook extra specialisatie-code krijgen.

13.1.1 Is-een relatie

Wanneer twee klassen met behulp van een “x is een y”-relatie kunnen beschreven worden dan weet je dat overerving mogelijk is.

- Een paard **is een dier**. Het paard is child-klasse. Dier is parent-klasse.
- Een tulp **is een plant**.
- Zowel een dier als een plant zijn levende wezens.

Als je dus in een programmeeropdracht het werkwoord *zijn* tegenkomt - was, is, zijn, zal zijn, enz. - dan is de kans groot dat overerving mogelijk is. Ik ga echter dit idee verderop in het boek uitbreiden met interfaces en polymorfisme. Deze twee begrippen hangen nauw samen met overerving en zullen soms een “betere oplossing” zijn dan pure overerving.



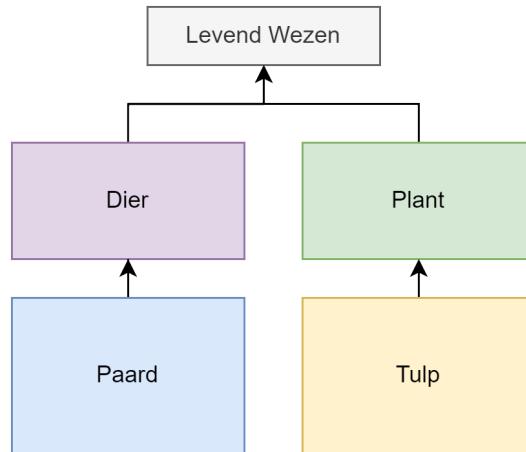
Wanneer we “x **heeft een y**” zeggen gaat het **niet** over overerving, maar over een associatie (zoals compositie of aggregatie) wat ik in het volgende hoofdstuk zal uitleggen.



Het is niet omdat 2 klassen delen gelijkaardige (of dezelfde) code hebben dat hier dus automatisch overerving van toepassing is. Enkel indien er een realistische “is een”-relatie bestaat kan overerving toegepast worden.

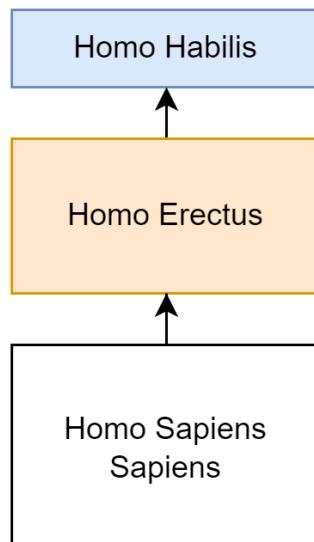
13.1.2 Overerving beschrijven

In UML-notatie duiden we een overervings-relatie aan met een pijl van de child- naar de parentklasse:



Figuur 13.1: Wat zou een parent-klasse van Levend Wezen kunnen zijn?

En als we het voorbeeld van de mens en z'n voorgangers nemen dan zou een vereenvoudigd UML-schema er als volgt uitzien:



Figuur 13.2: Oververing van onze soort.

13.2 Overerving in C#

Overving in C# duid je aan met behulp van het dubbele punt(:) bij de klassedefinitie, als volgt:

```

1 internal class Paard : Dier
2 {
3     public bool KanHinnikken{get;set;}
4 }
5
6 internal class Dier
7 {
8     public void Eet()
9     {
10     //...
11 }
12 }
```

We zeggen dus dat **Paard** overerft van de klasse **Dier**. Het paard is dus een specialisatie van dier. Objecten van het type **Dier** kunnen enkel de **Eet**-methode aanroepen. Objecten van het type **Paard** kunnen de **Eet**-methode aanroepen én ze hebben ook een property **KanHinnikken**. Een paard kan dus alles wat een dier kan en wat het zelf kan. Een dier kan enkel wat het zelf kan:

```

1 Dier aDier = new Dier();
2 Paard bPaard = new Paard();
3 aDier.Eet();
4 bPaard.Eet();
5 bPaard.KanHinnikken = false;
6 aDier.KanHinnikken = false; //!!! zal niet werken!
```

13.2.1 Transitief

Overerving in C# is transitief. Dit wil zeggen dat de child-klasse ALLES overerft van de parent-klasse: methoden, properties, enz.

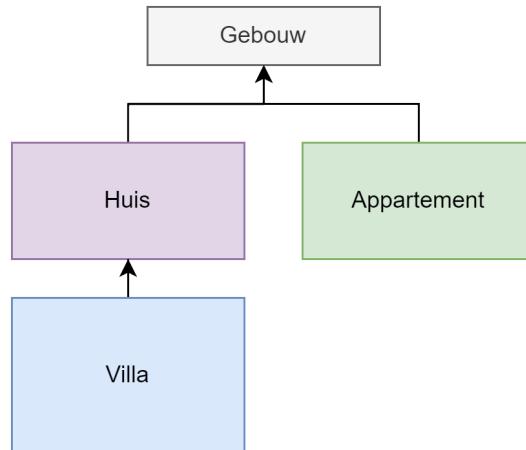
Dit kleine, korte zinnetje herbergt aardig wat kracht. Dankzij overerving kunnen we onze klasse dus proper en kort houden indien er een “is een” relatie bestaat. Er zijn echter ook enkele kanttekeningen aangaande overerving die ik in de komende secties uit te doeken zal doen.

Alhoewel overerving transitief is, wil dat niet zeggen dat **private** variabelen plots zichtbaar zijn in de child-klasse. De child-klasse erft ook de **private** instantievariabelen over, maar C# houdt zich wel aan de regels en zal voorkomen dat de child-code aan deze private parent instantievariabelen kan.

13.2.2 Overerving en het geheugen

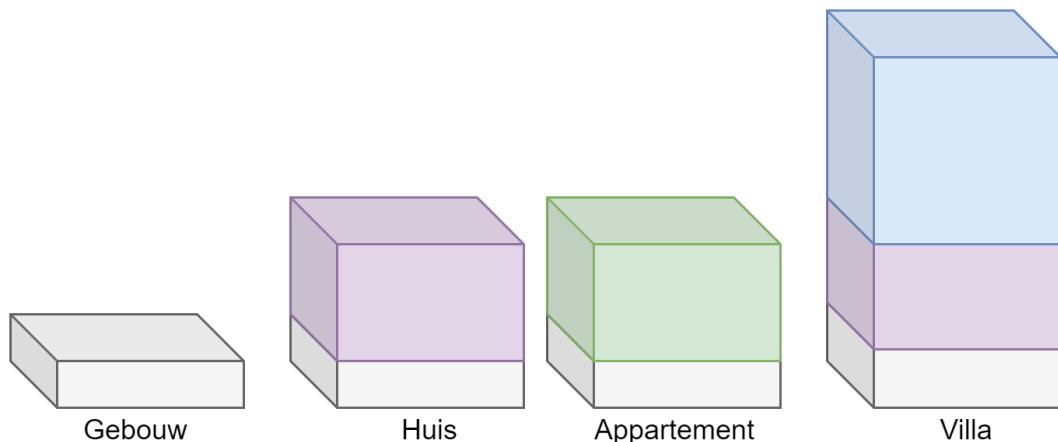
Tijd om eens te kijken hoe het voorgaande er uitziet in de heap en de stack, met een voorbeeld: een applicatie om aan gebouwbeheer te doen.

Beeld je in dat je volgende klassehiërarchie hebt vastgelegd:



Figuur 13.3: Villa, Huis en Appartement zijn een Gebouw. En Villa is op de koop toe een specialisatie van Huis.

Vervolgens maken we van iedere klasse 1 object aan. De objecten in het geheugen (de heap) zullen er dan als volgt uitzien:

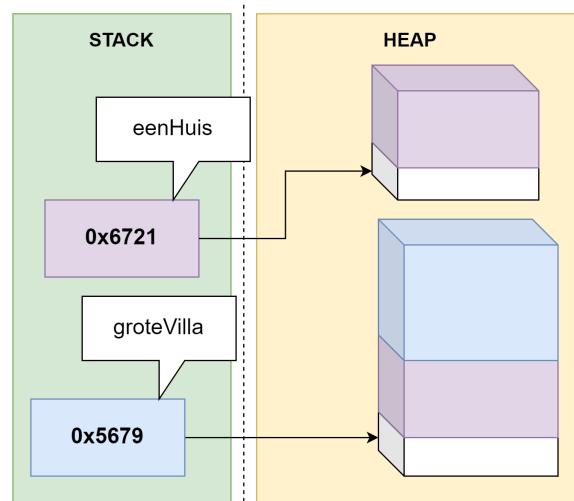


Figuur 13.4: We zien duidelijk dat een Appartement-object bestaat uit 2 delen: het Gebouw en de specialisatie Appartement.

Laten we eens 2 objecten aanmaken en kijken wat er in de heap en stack gebeurt:

```
1 Huis eenHuis = new Huis();
2 Villa groteVilla = new Villa();
```

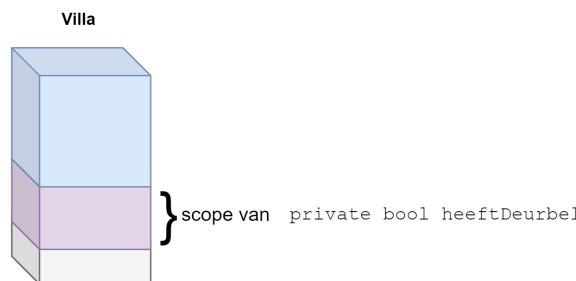
Dat ziet er dan als volgt uit:



Figuur 13.5: Blik op het geheugen nadat we een Huis en Villa hebben geïnstantieerd.

Ook hier zien we duidelijk dat een **Villa** object alle “code” in zich heeft die zowel in de klasse **Villa** staat (de specialisatie) alsook die waarvan wordt overgeërfd, **Huis**. Bijgevolg heeft **groteVilla** dus ook de “code” van **Gebouw** “in zich” door de transitiviteits-eigenschap van overerving.

Echter, de **private** delen van een klasse blijven beperkt tot dat stuk waar de variabele of methode origineel toe hoort. Als er dus in de klasse **Huis** een variabele **private bool heeftDeurbel** was, dan zal de code in de klasse **Villa** daar niet aan geraken:



Figuur 13.6: Private is echt private. Enkelde code binnen de klasse Huis kan deze bool benaderen.

13.2.3 **protected** keyword

Ook al is overerving transitief, hou er rekening mee dat private variabelen en methoden van de parent-klasse NIET rechtstreeks aanroepbaar zijn in de child-klasse. **private** geeft aan dat het element enkel in de klasse zichtbaar is:

```

1 internal class Paard: Dier
2 {
3     public void MaakOuder()
4     {
5         geboortejaar++; // !!! dit zal een error geven!
6     }
7 }
8 internal class Dier
9 {
10    private int geboortejaar;
11 }
```

Je kan dit oplossen door de **protected** access modifier te gebruiken in de plaats van **private**. Met **protected** geef je aan dat het element enkel zichtbaar is binnen de klasse **én** binnen child-klassen:

```

1 internal class Paard: Dier
2 {
3     public void MaakOuder()
4     {
5         geboortejaar++; // werkt nu wel
6     }
7 }
8 internal class Dier
9 {
10    protected int geboortejaar;
11 }
```



Alhoewel **protected** z'n nut heeft, is het meestal veiliger om alles nog steeds via properties te doen. Je kan dus beter van een property met **private set** er één met **protected set** van maken, zodat de achterliggende instantievariabele beschermd blijft.

13.2.4 Multiple inheritance

In C# is het niet mogelijk om een klasse van meer dan één parent-klasse te laten overerven (zogenaamde *multiple inheritance*). Dit is wel mogelijk in sommige andere object georiënteerde talen. Het is in C# dus niet mogelijk om een klasse Mens te maken die tegelijkertijd overerft van de klasse Aap en van Tekening.

Als puntje bij paaltje komt zal je trouwens bijna nooit multiple inheritance in de echte wereld tegenkomen (het typische tegenvoorbeeld is het vogelbekdier...maar hoe vaak ga je dat moeten modelleren in een project). Vaker zullen compositie en interfaces de oplossing zijn voor je probleem: 2 essentiële OOP aspecten die ik in de volgende hoofdstukken uit de doeken zal doen.

13.2.5 sealed

Soms wil je niet dat van een klasse nog nieuwe klassen kunnen overgeerfd worden. Je lost dit op door het keyword **sealed** voor de klasse te zetten:

```
1 internal sealed class DoNotInheritMe
2 {
3     //...
4 }
```

Als je later dan dit probeert:

```
1 internal class ChildClass:DoNotInheritMe
2 {
3     //...
4 }
```

zal dit resulteren in een foutbericht, namelijk `Cannot derive from sealed type 'DoNotInheritMe'`.

13.3 Constructors bij overerving

Wanneer je een object instantieert van een child-klasse dan gebeuren er meerdere zaken na elkaar, in volgende volgorde:

- Eerst wordt de constructor aangeroepen van de basis-klasse.
- Gevolgd door de constructors van alle parent-klassen.
- Finaal de constructor van de klasse zelf.

Dit is logisch: de child-klasse heeft de “fundering” nodig van z’n parent-klasse om te kunnen werken.

Volgende voorbeeld toont dit in actie:

```
1 internal class Soldaat
2 {
3     public Soldaat()
4     {
5         Debug.WriteLine("Soldaat is aangemaakt.");
6     }
7 }
8
9 internal class VeldArts : Soldaat
10 {
11     public VeldArts()
12     {
13         Debug.WriteLine("Veldarts is aangemaakt.");
14     }
15 }
```

Indien je vervolgens een object aanmaakt van het type `VeldArts`:

```
1 VeldArts RexGregor = new VeldArts();
```

Dan zien we de volgorde van constructor-aanroep in het debug output venster:

```
1 Soldaat is aangemaakt.
2 Veldarts is aangemaakt.
```

Er wordt dus verondersteld in dit geval dat er een default constructor in de basis-klasse aanwezig is.

13.3.1 Overloaded constructors en base()

Indien je klasse Soldaat een overloaded constructor heeft, dan wisten we al dat deze niet automatisch een default constructor heeft. Volgende code zou dus een probleem geven indien je een VeldArts wilt aanmaken via `new VeldArts()`:

```
1 internal class Soldaat
2 {
3     public Soldaat(bool kanSchieten)
4     {
5         //Doe soldaten dingen
6     }
7 }
8
9 internal class VeldArts:Soldaat
10 {
11     public VeldArts()
12     {
13         Debug.WriteLine("Veldarts is aangemaakt.");
14     }
15 }
```

Wat je namelijk niet ziet bij child-klassen en hun constructors is dat er eigenlijk een impliciete aanroep naar de constructor van de parent-klasse wordt gedaan. Bij alle constructors staat er eigenlijk `:base()` achter, wat je ook zelf kunt schrijven:

```
1 internal class VeldArts:Soldaat
2 {
3     public VeldArts(): base()
4     {
5         Debug.WriteLine("Veldarts is aangemaakt.");
6     }
7 }
```

Door `base()` achter de constructor te zetten ze je: “*roep de default constructor van de parent-klasse aan*”. Je mag hier echter ook parameters meegeven en de compiler zal dan zoeken naar een overloaded constructor in de basis-klasse die deze volgorde van parameters kan accepteren.

We zien hier hoe we ervoor moeten zorgen dat we terug via `new VeldArts()` objecten kunnen aanmaken zonder dat we de constructor(s) van Soldaat moeten aanpassen:

```
1 internal class Soldaat
2 {
3     public Soldaat(bool kanSchieten)
4     {
5         //Doe soldaten dingen
6     }
7 }
8 internal class VeldArts:Soldaat
9 {
10    public VeldArts():base(true)
11    {
12        Debug.WriteLine("Veldarts is aangemaakt.");
13    }
14 }
```

De default constructor van `VeldArts` zal de actuele parameter `kanSchieten` steeds op `true` zetten.

Uiteraard wil je misschien kunnen meegeven bij het aanmaken van een `VeldArts` wat de start-waarde van `kanSchieten` moet zijn. Dit vereist dat je een overloaded constructor in `VeldArts` aanmaakt, die op zijn beurt de overloaded constructor van `Soldaat` aanroeft.

Je schrijft dan een overloaded constructor in `VeldArts` bij:

```
1 internal class Soldaat
2 {
3     public Soldaat(bool kanSchieten)
4     {
5         //Doe soldaten dingen
6     }
7 }
8
9 internal class VeldArts:Soldaat
10 {
11    public VeldArts(bool kanSchieten): base(kanSchieten)
12    {}
13
14    public VeldArts():base(true) //Default
15    {
16        Debug.WriteLine("Veldarts is aangemaakt.");
17    }
18 }
```

Merk op hoe we de formele parameter `kanSchieten` doorgeven als actuele parameter aan `base`-aanroep.

Uiteraard mag je ook de default constructor aanroepen vanuit de child-constructor. Alle combinaties zijn mogelijk, zolang de constructor in kwestie maar bestaat in de parent-klasse.

Een hybride aanpak is ook mogelijk. Volgend voorbeeld toont 2 klassen, Huis en Gebouw waarbij we de constructor van Huis zodanig beschrijven dat deze bepaalde parameters “voor zich houdt” en andere als het ware doorsluist naar de aanroep van z’n parent-klasse:

```
1 internal class Gebouw
2 {
3     public int AantalVerdiepingen { get; private set; }
4     public Gebouw(int verdiepingenIn)
5     {
6         AantalVerdiepingen = verdiepingenIn;
7     }
8 }
9 internal class Huis: Gebouw
10 {
11     public bool HeeftTuintje { get; private set; };
12     public Huis(bool heeftTuin, int aantalVer): base(aantalVer)
13     {
14         HeeftTuintje = heeftTuin;
15     }
16 }
```

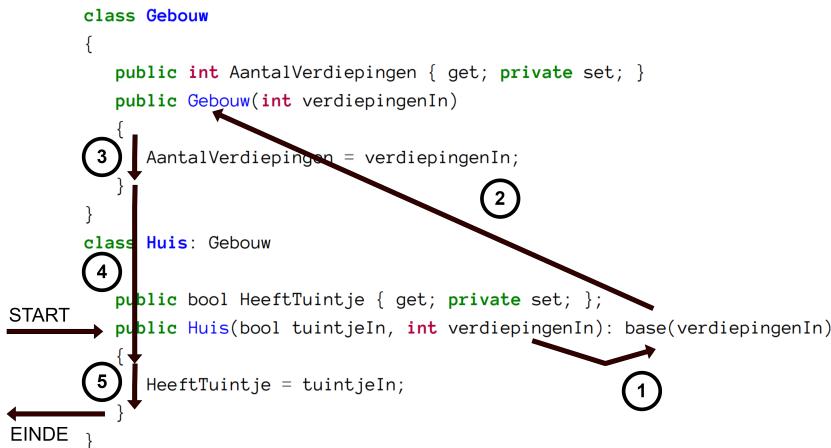
Vanaf nu kan ik een huis als volgt bouwen:

```
1 Huis peperkoekenHuis = new Huis(true, 1);
```

13.3.2 Volgorde van constructors

De volgorde waarin alles gebeurt in voorgaande voorbeeld is belangrijk om te begrijpen. Er wordt een hele machine in gang gezet wanneer we volgende korte stukje code schrijven:

```
1 Huis eenEigenHuis = new Huis(true,5);
```



Figuur 13.7: Achter de schermen gebeurt er aardig wat bij overerving wanneer we een object aanmaken.

Start: overloaded constructor van `Huis` wordt opgeroepen.

1. Nog voor dat deze echter iets kan doen, wordt de formele parameter `verdiepingenIn` (die de waarde 5 heeft gekregen) doorgegeven als actuele parameter om de constructor van de basis-klasse aan te roepen.
2. De overloaded constructor van `Gebouw` wordt dus aangeroepen.
3. De code van deze constructor wordt uitgevoerd: het aantal verdiepingen van het gebouw/huis wordt ingesteld.
4. Wanneer het einde van de constructor wordt bereikt, zal er teruggegaan worden naar de constructor van `Huis`.
5. Nu wordt de code van de `Huis` constructor uitgevoerd: `HeeftTuintje` krijgt de waarde `true`.

Einde: Finaal keren we terug en staat er nu een gloednieuw object in de heap, wiens geheugenlocatie we kunnen toewijzen aan `eenEigenHuis`.

13.4 Virtual en Override

Het is fijn dat onze child-klasse alles kan dat onze parent-klasse doet. Maar soms is dat beperkend:

- Mogelijk wil je een bestaande methode van de parent-klasse uitbreiden/aanvullen met extra functionaliteit.
- Soms wil je gewoon de volledige implementatie van een methode of property herschrijven in je child-klasse.

De keywords **virtual** en **override** gaan je hiermee kunnen helpen.

13.4.1 De werking van child-klassen aanpassen

Om te voorkomen dat child-klassen zomaar eender welke methode of property van de parent-klasse kunnen aanpassen gaan we de hulp van het **virtual** keyword inroepen. Standaard is het geen goede gewoonte om de bestaande werking van een klasse in de child-klasse aan te passen: beeld je in dat je een essentieel stuk code aanpast waardoor je hele klasse plots niet meer werkt!

Soms willen we echter kunnen aangeven dat de werking van een property of methode door een child-klassen mag aangepast worden. Dit geven we aan met het **virtual** keyword.

Vervolgens dient de child-klasse het keyword **override** te gebruiken om expliciet aan te geven dat er een methode of property komt wiens werking die van de parent-klasse zal wijzigen.



Enkel indien een element met **virtual** werd aangeduid, kan je deze dus met **override** aanpassen. Uiteraard ben je niet verplicht om elke *virtueel* element ook effectief te overriden. **virtual geeft enkel aan dat dit een mogelijkheid is, geen verplichting.**

13.4.2 Een voorbeeld met vliegende objecten

Stel je voor dat je een applicatie hebt met 2 klassen, Vliegtuig en Raket. Een raket is een vliegtuig, maar kan veel hoger vliegen dan een vliegtuig. Omdat we weten dat potentiële childklassen op een andere manier zullen willen vliegen, zullen we de methode Vlieg **virtual** zetten:

```

1 internal class Vliegtuig
2 {
3     public virtual void Vlieg()
4     {
5         Console.WriteLine("Het vliegtuig vliegt door de wolken.");
6     }
7 }
8 internal class Raket: Vliegtuig
9 {
10
11 }
```



Merk op dat we het keyword **virtual** mee opnemen in de methodesignatuur op lijn 3, en dat deze dus niets te maken heeft met het returntype en de zichtbaarheid van de methode. Dit zou bijvoorbeeld een perfect legale methodesignatuur kunnen zijn: **protected virtual int** SayWhatNow().
Terzijde: **static** methoden kunnen niet **virtual** gezet worden.

Stel dat we 2 objecten aanmaken en laten vliegen:

```

1 Vliegtuig topGun = new Vliegtuig();
2 Raket spaceX1 = new Raket();
3 topGun.Vlieg();
4 spaceX1.Vlieg();
```

De uitvoer zal dan zijn twee maal dezelfde zin tonen: Het vliegtuig vliegt door de wolken.



Enkel **public** methoden en properties kan je **virtual** instellen.

Momenteel doet het **virtual** keyword niets. Het is enkel een signaal aan mede-programmeurs: “hey, als je wilt mag je de werking van deze methode aanpassen als je van deze klasse overerft.”

Een raket is een vliegtuig, toch vliegt het anders. We willen dus de methode `Vlieg` anders uitvoeren voor een raket. Daar hebben we **override** voor nodig. Door override voor een methode in de child-klasse te plaatsen zeggen we “gebruik deze implementatie en niet die van de parent klasse.”

```
1 internal class Raket:Vliegtuig
2 {
3     public override void Vlieg()
4     {
5         Console.WriteLine("De raket verdwijnt in de ruimte.");
6     }
7 }
```

De uitvoer van volgende code zal nu anders zijn:

```
1 Vliegtuig topGun = new Vliegtuig();
2 Raket spaceX1 = new Raket();
3 topGun.Vlieg();
4 spaceX1.Vlieg();
```

Uitvoer:

```
1 Het vliegtuig vliegt door de wolken.
2 De raket verdwijnt in de ruimte.
```



Indien je iets **override** moet de signatuur van je methode of property identiek zijn aan deze van de parent-klasse. Het enige verschil is dat je het keyword **virtual** vervangt door **override**.

Als je in VS override begint te typen in een child-klassen dan kan je met behulp van de tab-toets heel snel de overige code van de signatuur schrijven.

13.5 Het base keyword

Het **base** keyword laat ons toe om bij **override** van een methode of property in de child-klasse toch te verplichten om de parent-implementatie toe te passen. Dit kan handig zijn wanneer je in je child-klasse de bestaande implementatie wenst uit te breiden.

Stel dat we volgende 2 klassen hebben:

```

1 internal class Restaurant
2 {
3     protected int kosten = 0;
4     public virtual void PoetsAlles()
5     {
6         kosten += 1000;
7     }
8 }
9 internal class Frituur:Restaurant
10 {
11     public override void PoetsAlles()
12     {
13         kosten += (1000 + 500); //SLECHT IDEE! Wat als de
14             basiskosten in het restaurant veranderen?
15 }
```

Het poetsen van een Frituur is duurder (1000 basis + 500 voor ontsmetting) dan een gewoon Restaurant. Als we echter later beslissen dat de basisprijs (in Restaurant) moet veranderen dan moet je ook in alle child-klassen doen, wat natuurlijk geen goede programmeerstijl is.

base lost dit voor ons op. De Frituur-klasse herschrijven we naar:

```

1 internal class Frituur:Restaurant
2 {
3     public override void PoetsAlles()
4     {
5         base.PoetsAlles(); //eerste basiskost wordt opgeteld
6         kosten += 500; //kosten eigen aan frituur worden bijgeteld
7     }
8 }
```

Het **base** keyword laat ons toe om in onze code expliciet een methode of property van de parent-klasse aan te roepen. Ook al overschrijven we de implementatie van PoetsAlles toch kan de originele versie van de parent-klasse nog steeds gebruikt worden.



We hebben een soortgelijke werking ook reeds gezien bij de constructors van overgeërfde klassen.

Je kan zelf beslissen waar in je code je **base** aanroeft. Soms doe je dat aan de start van de methode, soms op het einde, soms halverwege. Alles hangt er van af wat je juist nodig hebt.



“Ik denk dat ik een extra voorbeeldje nodig ga hebben.”

Laten we eens kijken. Beeld je in dat je volgende basisklasse hebt:

```

1 internal class Oermens
2 {
3     public virtual int VoorzieVoedsel()
4     {
5         return 15; //kg
6     }
7 }
```

Wanneer 1 van mijn dorpsgenoten voedsel zoekt door te jagen zal hij 15 kg vlees verzamelen.

De moderne mens, die overerft van de oermens, is natuurlijk al iets beter in het maken van voedsel en kan dagelijks standaard 100 kg voedsel maken.

Echter, er bestaan ook jagers die nog op de klassieke manier voedsel kunnen verzamelen (maar ze zijn wel gewoon moderne mensen, dus geen klasse apart hier). Uiteraard hebben zij de technieken van de oermens verbeterd en zullen sowieso toch iets meer voedsel nog kunnen verzamelen met de traditionele methoden, namelijk 20 kg bovenop de basishoeveelheid van 15 kg.

```

1 internal class ModerneMens: Oermens
2 {
3     public bool IsJager {get; set;}
4
5     public override int VoorzieVoedsel()
6     {
7         if (IsJager)
8             return base.VoorzieVoedsel() + 20;
9         return 100;
10    }
11 }
```

13.5.1 Een wereld met OOP: Pong overerving

Dankzij overerving zijn we nu in staat om Pong uit te breiden met andere soort balletjes. De eerste vraag die je moet stellen is dan “welke werking in de klasse Balletje gaan we potentieel willen aanpassen?”. Laten we veronderstellen dat we enkel de Update mogelijk willen veranderen. We voegen daarom het **virtual** keyword aan die methode toe:

```
1 virtual public void Update()
```

Voor de rest passen we hier niets aan. Dankzij overerving kunnen we de klasse Balletje nu onaangeroerd laten en onze nieuwe functionaliteit toevoegen via child-klassen.

Stel dat we een nieuw Balletje willen ontwikkelen, genaamd CentreerBalletje. Dit balletje heeft als eigenschappen dat het terug naar het midden van het scherm *teleporteert* wanneer het de linker- of rechterzijde van het scherm raakt. Dit zal er zo uitzien:

```
1 internal class CentreerBalletje : Balletje
2 {
3     public override void Update()
4     {
5         if(X+VX >= Console.WindowWidth || X+VX < 0)
6         {
7             X = Console.WindowWidth / 2;
8             Y = Console.WindowHeight / 2;
9         }
10        base.Update();
11    }
12 }
```

We hoeven nu enkel in het hoofdprogramma alle Balletje-variabelen te vervangen door CentreerBalletje.



Dankzij polymorfisme verderop gaan we ontdekken dat zelfs dit eigenlijk mag!

```
1 Balletje bal1 = new CentreerBalletje();
```


14 Gevorderde overervingsconcepten

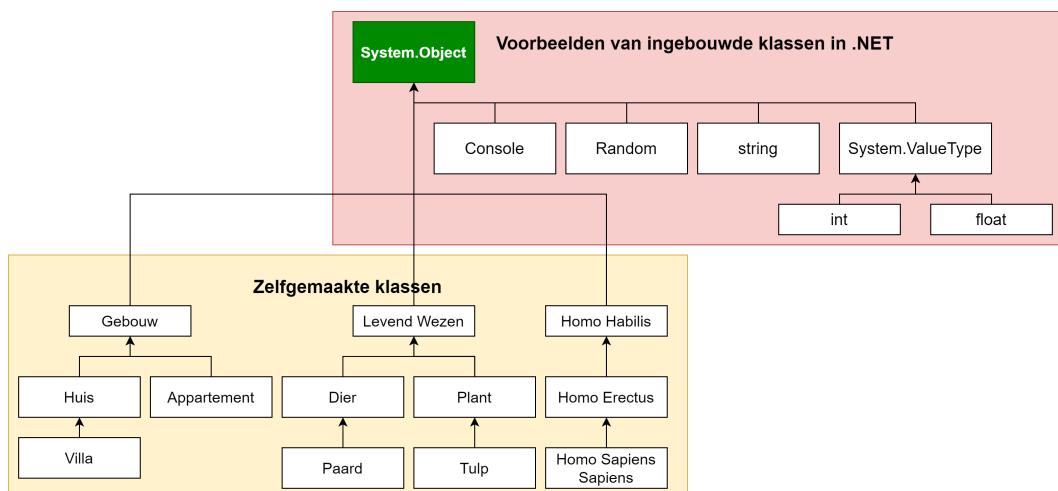
C# houdt van objecten. De hele taal is letterlijk opgebouwd om maximaal het object georiënteerd programmeren te omarmen. Van zodra je een nieuw project aanmaakt kan je niet naast de **internal class** Program zien. Hoe meer je C# en de bestaande bibliotheken bekijkt, hoe duidelijker dit wordt. Alles is een klasse.

Maar dan stelt zich natuurlijk de vraag: *staat er nog iets boven alle klassen die wij aan het maken zijn? Is er misschien een soort oer-klasse waar alle klassen van overerven?*

De vraag stellen is ze beantwoorden! Er is effectief een oer-klasse, genaamd de **System.Object. Object-klasse, waar alles en iedereen in C# van moet overerven.**

14.1 System.Object

Alle klassen in C# zijn afstammelingen van de **System.Object** klasse. Zowel de bestaande ingebouwde klassen zoals Random en Console. Maar ook klassen die je zelf maakt erven over van System.Object. En ja, zelfs de bestaande valuetype datatypes zoals **int** en **bool** zijn verre afstammelingen van System.Object.



Figuur 14.1: Enkele voorbeelden. Merk op dat er véél meer ingebouwde klassen in .NET zitten dan degene die ik hier toon.

Indien je een klasse schrijft zonder een expliciete parent dan zal deze steeds System.Object als rechtstreekse parent hebben. Ook afgeleide klassen stammen dus uiteindelijk af van System.Object. Concreet wil dit zeggen dat alle klassen System.Object-klassen zijn en dus ook de bijhorende functionaliteit ervan hebben.



Om de klasse Object niet te verwarren met het concept “object” zullen we hier steeds praten over System.Object.

14.1.1 Impliciete overerving

Wanneer je een klasse Student aanmaakt als volgt: `class Student{ }`. Dan gebeurt er een zogenaamde impliciete overerving van System.Object. Er staat dus eigenlijk:

```
1 internal class Student: System.Object
2 { }
```

Wat je trouwens ook expliciet zelf mag schrijven, dat maakt niet uit. **Maar van zodra je een klasse schrijft die nergens expliciet van overerft, dan zal deze automatisch van System.Object overerven.¹**:

14.1.2 Hoe ziet System.Object er uit?

Wanneer je een lege klasse maakt dan zal je misschien al gezien hebben dat instanties van deze nieuwe klasse reeds 4 methoden ingebouwd hebben, dit zijn uiteraard de methoden die in de System.Object klasse staan gedefinieerd:

Methode	Beschrijving
Equals()	Gebruikt om te ontdekken of twee instanties gelijk zijn.
GetHashCode()	Geeft een unieke <i>hash</i> terug van het object; nuttig om o.a. te sorteren.
GetType()	Geeft het datatype (de klasse) van het object terug.
ToString()	Geeft een string terug die het object voorstelt.

Deze methoden zijn redelijk nutteloos in het begin. Enkel door ze zelf te overriden zullen ze hun nut bewijzen. Uiteraard kan je de methoden testen om te zien wat er gebeurt.

¹ Je kan in de .NET documentatie altijd opzoeken waar een klasse van overerft. De Type klasse bijvoorbeeld erft final ook van System.Object over. Eerst erft Type over van de MemberInfo klasse, die op zijn beurt overerft van de oer-klasse.

14.1.3 GetType()

Stel dat je een klasse Student hebt gemaakt in je project. Je kan dan op een object van deze klasse de GetType ()-methode aanroepen om te weten wat het type van dit object is:

```
1 Student stud1 = new Student();
2 Console.WriteLine(stud1.GetType());
```

Dit zal als uitvoer de namespace gevuld door het type van het object op het scherm geven . Als je project bijvoorbeeld StudentManager heet (en je namespace dus vermoedelijk ook) dan zal er op het scherm verschijnen: StudentManager . Student.

Wil je enkel het type zonder namespace dan is het nuttig te beseffen dat GetType () eigenlijk een object teruggeeft van het type Type met meerdere eigenschappen, waaronder Name. Volgende code zal enkel Student op het scherm tonen:

```
1 Student stud1 = new Student();
2 Console.WriteLine(stud1.GetType().Name);
```

14.1.4 ToString(): het werkpaardje van System.Object

Deze methode vind ik het nuttigst. Wanneer je schrijft:

```
1 Console.WriteLine(stud1);
```

Wordt er eigenlijk een impliciete aanroep naar ToString gedaan. Er staat dus eigenlijk:

```
1 Console.WriteLine(stud1.ToString());
```

Op het scherm verschijnt dan StudentManager . Student. Waarom? Wel, de methode ToString() wordt in System.Object() ongeveer als volgt beschreven:

```
1 public virtual string ToString()
2 {
3     return GetType();
4 }
```

Merk twee zaken op:

1. GetType() wordt aangeroepen en die output krijg je dus terug.
2. De methode is **virtual** gedefinieerd.

Alle 4 methoden in System.Object zijn **virtual, en je kan deze dus **override**'n!**

Nu komen we tot het hart van deze methoden. Aangezien ze alle 4 **virtual** zijn, kunnen we de werking ervan naar onze hand zetten in onze eigen klassen. Aardig wat .NET bibliotheken rekenen er namelijk op dat je deze methoden op de juiste manier hebt aangepast, zodat ook jouw nieuwe

klassen perfect kunnen samenwerken met deze bibliotheken. Een eerste voorbeeld hiervan toonde ik net: de `Console.WriteLine` methode gebruikt van iedere parameter dat je er aan meegeeft de `ToString`-methode om de parameter op het scherm als **string** te tonen.

14.1.4.1 `ToString()` overiden

Het zou natuurlijk fijner zijn dat de `ToString()`-methode van onze student nuttigere info teruggeeft.

Stel dat we de Voornaam gevolgd door de Geboortejaar (ook een autoprop) willen terugkrijgen. We kunnen dat eenvoudig verkrijgen door `ToString()` te overiden:

```

1 internal class Student
2 {
3     public int Geboortejaar {get; set;}
4     public string Voornaam {get; set;}
5     public override string ToString()
6     {
7         return ${Voornaam} ({Geboortejaar});
8     }
9 }
```

Wanneer je nu `Console.WriteLine(stud1);` - gelet dat hij de properties Voornaam en Geboortejaar heeft - zou schrijven dan wordt je output: Tim Dams (1981).



Een extra handigheidje van `ToString` is dat deze methode wordt gebruikt tijdens het debuggen om je objecten samen te vatten in het watch-venster.

14.1.5 De `Equals()` methode

Ook deze methode kan je overiden om twee objecten met elkaar te vergelijken:

```
1 if(stud1.Equals(stud2))
```

De `Equals()`-methode heeft als signatuur: `public virtual bool Equals(Object o)`
Twee objecten zijn gelijk voor .NET als aan volgende afspraken wordt voldaan:

- Het moet **false** teruggeven indien de parameter `o` **null** is.
- Het moet **true** teruggeven indien je het object met zichzelf vergelijkt (bv. `stud1.Equals(stud1)`).
- Het mag enkel **true** teruggeven als zowel `stud1.Equals(stud2)`; als `stud2.Equals(stud1)`; waar zijn.
- Indien `stud1.Equals(stud2)` true teruggeeft en `stud1.Equals(stud3)` ook **true** is, dan moet `stud2.Equals(stud3)` ook **true** zijn.

14.1.5.1 Equals() overiden

Het is echter aan de maker van de klasse om te beslissen wanneer 2 objecten van eenzelfde type gelijk zijn. Het is dus niet zo dat iedere waarde van een instantievariabele bijvoorbeeld gelijk moet zijn opdat 2 objecten gelijk zijn. Alles hangt af van de wijze waarop de klasse dienst moet doen.

Stel dat we vinden dat een student gelijk is aan een andere student indien z'n Voornaam en Geboortejaar dezelfde is, we kunnen dan de Equals-methode overiden als volgt in de Student klasse:

```

1 public override bool Equals(Object o)
2 {
3     Student temp = (Student)o; //Zie opmerking na code!
4     return (Geboortejaar == temp.Geboortejaar && Voornaam == temp.
5         Voornaam);
}
```



De lijn `Student temp = (Student)o;` zal het **object** `o` casten naar een `Student`. Doe je dit niet dan kan je niet aan de interne `Student`-variabelen van het **object** `o`. Dit concept, **polymorfisme** (zie nog steeds hoofdstuk 16....We komen dichter!).

14.1.6 GetHashCode() overiden

Indien je `Equals` overide dan moet je eigenlijk ook `GetHashCode` overiden, daar er wordt verondersteld dat twee gelijke objecten ook dezelfde unieke hashcode teruggeven. Wil je dit dus implementeren dan zal je dus een (bestaand) algoritme moeten schrijven dat een uniek nummer genereert voor ieder niet-gelijk object. Algoritmes bespreken om zelf een hash te genereren liggen niet in de scope van dit boek.

14.1.7 ReferenceEquals()

Als je nog wat dieper zou graven in de documentatie van `System.Object` zou je ontdekken dat er ook een **static** methode met als signatuur **static bool** `ReferenceEquals(object obj1, object obj2)` bestaat. Deze handige methode laat je toe om te controleren of 2 variabelen dezelfde referentie hebben. Je kan hiermee dus kijken of 2 variabelen naar hetzelfde object in de heap verwijzen. Het gebruik ervan is eenvoudig:

```

1 if(ReferenceEquals(student1,student3))
2 {
3     Console.WriteLine("Beide bevatten zelfde referentie!");
4 }
```

Nu stelt zich de vraag: waarom deze controle niet met de == doen? Alhoewel dit perfect toegestaan is, moet je je ervan bewust zijn dat de werking van == kan overschreven worden. Ik leg dit niet uit, maar kijk zeker eens in de appendix naar het hoofdstuk **Operator overloading**.

Je hebt dus geen garantie dat in alle projecten de == werkt zoals je zou verwachten. Prefereer daarom `ReferenceEquals()` te gebruiken. Merk op dat we `System.` voor de methode-naam mogen weglaten, net zoals we `Object` in plaats van `System.Object` mogen schrijven.



“Ik ben nog niet helemaal mee...”

Niet getreurd, je bent niet de enige: het is allemaal een hoop nieuwe kennis om te verwerken. En ik vermoed dat je nu niet bepaald overweldigd bent van de nieuwe kennis. Mogelijk heb je nu zo iets van? “Ok..wow?! Wat krijg ik nu juist extra wetende dat al mijn klassen overerven van een oer-klas? 4 methoden en wat beloofde compatibiliteit met andere .NET bibliotheken? Call me ...unimpressed”. Begrijpelijke reactie.

Hou vol, we zijn een hoop puzzelstukjes aan het opnemen die finaal zullen samenkommen om een gigantisch knappe OOPuzzel te maken (see what I did there?) In die puzzel zal polymorfisme onze sterspeler worden. Het zal ons toelaten erg krachtige code te schrijven.

Polymorfisme wordt onze doelpuntenmaker, maar `System.Object` zal steeds de perfecte voorzet geven!

14.2 Abstracte klassen

Aan de start van hoofdstuk 9 gaf ik volgende twee definities:

- **Een klasse** is als een **blauwdruk** dat het gedrag en toestand beschrijft van alle objecten van deze klasse.
- Een individueel **object** is een **instantie** van een klasse en heeft een eigen *toestand, gedrag en identiteit*.

Niemand die zich hier vragen bij stelde? Als ik in het echte leven zeg: “Geef mij eens de blauwdruk van een object van het type meubel.” Wat voor soort meubel zie je voor je bij het lezen van deze zin? Een tafel? Een kast? Een zetel? Een bed?

En wat zie je voor je als ik vraag om een “geometrische figuur” in te beelden. Een cirkel? Een rechthoek? Een kubus? Een buckyball? Kortom, er zijn in het leven ook soms eerder abstracte dingen die niet op zich in objecten kunnen gegoten worden zonder meer informatie.

Toch is het concept “geometrische figuur” een belangrijk concept: we weten dat alle geometrische figuren een gemeenschappelijke definitie hebben, namelijk - met dank aan Encyclo.nl- dat het *twee- of meerdimensionale grafische elementen zijn waarvan de vorm wiskundig te berekenen valt*. En dus is er ook een bestaansreden voor een klasse GeometrischeFiguur. **Objecten van deze abstracte klasse maken daarentegen lijkt ons nutteloos.**

Het is dit concept, **abstracte klasse** dat ik in dit hoofdstuk uit te doeken doe. Het laat ons toe klassen te definiëren die niet kunnen geïnstantieerd worden, maar die wel dienst kunnen doen als parentklasse voor andere klassen.

14.2.1 Abstracte klassen in C#

Laten we voorgaande eens praktisch binnen C# bekijken. Soms maken we een parent-klasse waarvan geen instanties kunnen gemaakt worden: denk aan de parent-klasse Dier. Voorbeelden van subklassen van Dier zijn Paard en Wolf. Van Paard en Wolf is het logisch dat je instanties kan maken (echte paardjes en wolfjes) maar van ‘een dier’? Hoe zou dat er uit zien? Maar toch willen we bepaalde delen gemeenschappelijk maken (alle dieren hebben bijvoorbeeld zuurstof nodig).

Met behulp van het keyword **abstract** kunnen we aangeven dat een klasse abstract is: **je kan overerven van deze klasse, maar je kan er geen instanties van aanmaken.**

We plaatsen **abstract** voor de klasse definitie om dit aan te duiden.

Een voorbeeld:

```
1 internal abstract class Dier
2 {
3     public string Naam {get;set;}
4 }
```

We kunnen nu geen objecten meer van het type `Dier` aanmaken. Volgende code zal een foutbericht geven: `Dier hetDier = new Dier();`

Maar, we mogen dus wel klassen overerven van deze klasse en instanties van deze nieuwe klasse aanmaken:

```
1 internal class Paard: Dier
2 {
3     //...
4 }
5
6 internal class Wolf: Dier
7 {
8     //...
9 }
```

En dan zal dit wel werken: `Wolf wolfje = new Wolf();`

En als we polymorfisme gebruiken (*soon!*) dan mag dit ook: `Dier paardje = new Paard();`



In het begin lijkt **abstract** een beperkende factor: je kan minder dan ervoor. Maar het heeft dus één heel duidelijke functie: je kan een parent-klasse maken waarin de gedeelde functionaliteit van je child-klassen in zit, zonder dat je deze parent-klasse op zich kunt gebruiken.

14.2.2 Abstracte methoden

Het is logisch dat we mogelijk ook bepaalde zaken in de abstracte klasse als **abstract** kunnen aanduiden. Beeld je in dat je een methode `MaakGeluid` hebt in je klasse `Dier`. Wat voor een geluid maakt ‘een dier’? We kunnen dus ook geen implementatie (code) geven in de abstracte parent klasse, maar willen wel zeker ervoor zorgen dat alle child-klassen van `Dier` geluid kunnen maken, op wat voor manier dan ook.

Via abstracte methoden geven we dit aan: we hoeven enkel de methode signatuur te geven, met ervoor **abstract**:

```
1 internal abstract class Dier
2 {
3     public abstract string MaakGeluid();
4 }
```

Door het keyword **abstract** zijn child-klassen verplicht deze abstracte methoden te overrodden!



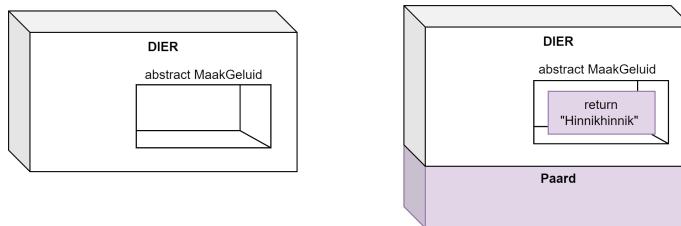
Merk op dat er geen codeblock-accolades na de signatuur van abstracte methodes komt.

De Paard-kLASSE wordt dan²:

```

1 internal class Paard: Dier
2 {
3     public bool HeeftTetanus {get; set;}
4
5     public override string MaakGeluid()
6     {
7         return "Hinnikhinnik";
8     }
9 }
```

Dit is dus niet hetzelfde als **virtual** waar een **override** MAG. Bij **abstract** MOET je **override**'n. We komen dan ook bij de essentie van het abstracte klasse concept: ze laten ons toe om klassen te maken waar nog *gaten* in zitten qua implementatie. We maken als het ware een soort klasse-sjabloon, die de child-klassen nog verder moeten inkleuren.



Figuur 14.2: Alhoewel de code voor MaakGeluid staat beschreven in de klasse Paard, zal deze als het ware ingevuld worden op de plek ervoor in de klasse Dier.



14.2.2.1 Abstracte methoden enkel in abstracte klassen

Van zodra een klasse een abstracte methode of property heeft dan ben je verplicht om de klasse ook abstract te maken.

Het zou heel vreemd zijn om objecten in het leven te kunnen roepen die letterlijk stukken ontbrekende code hebben...

²En idem voor de Wolf-klasse uiteraard, maar hopelijk met een dreigender geluid.

14.2.3 Abstracte properties

Properties kunnen **virtual** gemaakt worden, en dus ook **abstract**. Net zoals bij abstracte methoden, kunnen we met abstracte properties de overgeërfde klassen verplichten een eigen implementatie van de property te schrijven.

Volgend voorbeeld toont hoe dit werkt:

```
1 internal abstract class Dier
2 {
3     abstract public int MaxLeeftijd { get; }
4 }
5
6 internal class Olifant : Dier
7 {
8     public override int MaxLeeftijd
9     {
10         get
11         {
12             return 100;
13         }
14     }
15 }
```

Wanneer je een abstracte property maakt dien je ogenblikkelijk aan te geven of het om een readonly, writeonly, of property met get én set gaat:

- **public abstract int** Oppervlakte {**get**;}
- **public abstract int** GeheimeCode {**set**;}
- **public abstract int** GeboorteDatum {**get**; **set**;}

14.2.4 Een wereld met OOP: Pong en abstract

Dankzij **abstract** kunnen we nu een meer algemene klasse maken in Pong. Beeld je in dat je naast balltjes ook andere zaken op het scherm wilt tonen. Echter, niet alles moet als een gek over het scherm vliegen én is op de koop toe niet noodzakelijk een *child* van de *Balletje*-klasse.

We definiëren daarom een klasse die alle zaken zal voorstellen die “op het scherm” moeten getoond worden. Omdat we niet weten **HOE** die zaken getoond worden, zal dit een abstracte klasse worden waarbij we de *TekenOpScherm*-methode bewust niet implementeren:

```

1 internal abstract class SpelObject
2 {
3     public int X { get; set; }
4     public int Y { get; set; }
5     public abstract void TekenOpScherm();
6 }
```

We kunnen nu ons klasse *Balletje* hier van laten overerven en veranderen volgende zaken:

- We halen de X en Y properties uit de klasse (daar de parent deze al heeft gedefinieerd).
- We veranderen *TekenOpScherm* van een **virtual** naar een **override** versie, daar we nu de abstracte methode van de parent **moeten** implementeren. Merk op dat dit geen invloed heeft op de child-klassen van *Balletje*, die zullen nog steeds in staat zijn om de Update-versie van *Balletje* te override'n.

```

1 internal class Balletje:SpelObject
2 {
3     //...
4
5     public override void TekenOpScherm()
6     {
7         //...
8     }
9 }
```

Dankzij deze abstracte klasse hebben we nu een manier om bijvoorbeeld ook een scorebord in het spel te brengen:

```

1 internal class ScoreBoard: SpelObject
2 {
3     public ScoreBoard()
4     {
5         X = 5;
6         Y = 5;
7     }
8
9     public int ScoreSpeler1 { get; set; }
10    public int ScoreSpeler2 { get; set; }
11    public override void TekentOpScherm()
12    {
13        Console.BackgroundColor = ConsoleColor.Yellow;
14        Console.ForegroundColor = ConsoleColor.Black;
15        Console.SetCursorPosition(X, Y);
16        Console.WriteLine($"{ScoreSpeler1} - {ScoreSpeler2}");
17        Console.ResetColor();
18    }
19 }
```

In ons hoofdprogramma blijven we leven van de kracht van polymorfisme en gebruiken we een snufje **is** en **as** om zeker onze Balletje ook te update'n wanneer nodig. Onze lijst, hernoemd naar spelElementen zal nu SpelObject objecten bevatten:

```

1 List<SpelObject> spelElementen = new List<SpelObject>();
2
3 //Balletjes toevoegen...
4
5 //En nu het scoreboard
6 var score = new ScoreBoard()
7 spelElementen.Add();
8
9 while (true)
10 {
11     foreach(var spelObject in spelElementen)
12     {
13         //update enkel de balletjes
14         if(spelObject is Balletje)
15         {
16             (spelObject as Balletje).Update();
17         }
18
19         //spe
20     }
```

Indien nu een speler scoort dan kunnen we schrijven:

```
1 score.ScoreSpeler2++;
```

14.3 Zelf exceptions maken

We zijn ondertussen al gewend aan het opvangen van uitzonderingen met behulp van **try** en **catch**. Ook bij exception handling wordt overerving toegepast. De uitzonderingen die we opvangen zijn steeds objecten van het type `Exception` of van een afgeleide klasse. Denk maar aan de `NullReferenceException` klasse die werd overgeerfd van `Exception`.



Dat wil zeggen dat `Exceptions` ook maar “gewone klassen” zijn en dus ook aan alle andere regels binnen C# moeten voldoen. Zo ondersteunen ze polymorfisme (*soooooon!*), kan je ze in arrays plaatsen, enz.

Bijgevolg is het logisch dat je in je code **uitzonderingen zelf kunt maken en opwerpen**. Vervolgens kan je deze elders opvangen.

Een voorbeeld van een bestaand `Exception` type gebruiken. We gaan zelf een `Exception` object aanmaken (met `new`) en dit vervolgens opwerpen wanneer we een uitzondering opmerken. In dit geval wanneer getal de waarde 0 heeft:

```
1 static int ResultaatBerekening(int getal)
2 {
3     if (getal != 0)
4         return 100 / getal;
5     else
6         throw new DivideByZeroException("BOEM. ZWART GAT!");
7 }
8
9
10 static void Main(string[] args)
11 {
12     try
13     {
14         Console.WriteLine(ResultaatBerekening(0));
15     }
16     catch(DivideByZeroException e)
17     {
18         Console.WriteLine(e.Message);
19     }
20 }
```

De uitvoer zal zijn:

```
1 BOEM. ZWART GAT!
```

De lijn `throw new DivideByZeroException("BOEM. ZWART GAT!");` zorgt er dus voor dat we een eigen foutbericht verpakken en opwerpen.

14.3.1 Een eigen exception ontwerpen

Je kan ook eigen klassen overerven van `Exception` zodat je eigen uitzonderingen kan maken. Je maakt hiervoor gewoon een nieuwe klasse aan die je laat overerven van de `Exception`-klasse. Een voorbeeld:

```
1 internal class Timception: Exception
2 {
3     public override string ToString()
4     {
5         string extrainfo = "Exception Generated by Tim Dams:\n";
6         return $"{extrainfo}. {base.ToString()}";
7     }
8 }
```



Merk op dat we hier met `base.ToString()` ervoor zorgen dat ook de foutboodschap van het parent-gedeelte van de uitzondering wordt weergegeven.

Om deze exception nu zelf op te werpen gebruiken we het keyword `throw` gevolgd door een object van het type uitzondering dat je wenst op te werpen.

In volgende voorbeeld gooien we onze eigen exception op een bepaald punt in de code op en vangen deze dan op:

```
1 static void Main(string[] args)
2 {
3     try
4     {
5         TimsMethode();
6     }
7
8     catch (Timception e)
9     {
10        Console.WriteLine(e.ToString());
11    }
12 }
13 static public void TimsMethode()
14 {
15     //doe dingen
16     //...
17     //"when suddenly: a wild exception appears"
18     throw new Timception();
19 }
```

15 Associaties



Dit hoofdstuk is kort maar krachtig. Ik ga eigenlijk niets nieuws uitleggen. Ik ga zaken benoemen die je waarschijnlijk al toepaste, zonder te weten dat er daar ook een naam voor was.

We spreken over compositie (**compositie**) en aggregatie (**aggregation**) wanneer we een object in een ander object gebruiken. Beide termen zijn zogenaamde **associaties**. Ze beschrijven de relatie tussen 2 objecten. Denk bijvoorbeeld aan een object van het type Motor dat je gebruikt in een object van het type Auto. Afhankelijk of het *interne* object kan bestaan zonder het *omliggende* object bepaalt of het gaat om aggregatie of compositie:

- **Compositie:** Het interne object heeft geen bestaansreden zonder het omliggende object. Denk bijvoorbeeld aan een kamer in een huis. Als het huis verdwijnt, verdwijnt ook de kamer.
- **Aggregatie:** Beide objecten kunnen onafhankelijk van elkaar bestaan. Denk hierbij aan de motor in een auto. Wanneer de auto vernietigd wordt kan de motor gered worden en elders gebruikt worden. Een ander voorbeeld zijn de harde schijven in een computer.

Het lijdende voorwerp zal steeds het object zijn dat binnen het onderwerp zal geplaatst worden (*motor* in *auto*, *schijf* in *computer*).

15.1 Heeft een-relatie

Overerving konden we detecteren door de “is een”-relatie. **Een associatie daarentegen detecteren we met behulp van de “heeft een”-relatie tussen 2 klassen.** *Een mango heeft een pit. Een vliegtuig heeft een cockpit.* enz.

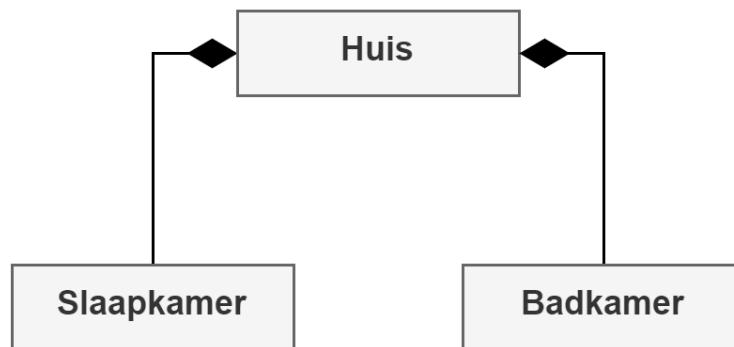
Je hoort ook ogenblikkelijk of het om een “heeft één” of “heeft meerdere”-relatie gaat. In het tweede geval (*heeft meerdere*) wil dit zeggen dat het omliggende object **een array van het interne object** in zich heeft. Wederom het voorbeeld van het boek: een boek heeft meerdere pagina’s. Dus in de klasse Boek zullen we een object van het type Pagina[] of List<Pagina> tegenkomen.



Een klassieke fout is overerving gebruiken wanneer je bijvoorbeeld de relatie tussen een boek en z'n pagina's wilt aanduiden. Een boek is géén pagina, ook niet omgekeerd. Er is dus geen sprake van overerving. Een boek **HEEFT** een pagina (of meerdere). Er dus sprake van een associati, namelijk aggregatie (je kan pagina's uit een boek scheuren en deze nog steeds doorgeven aan iemand anders).

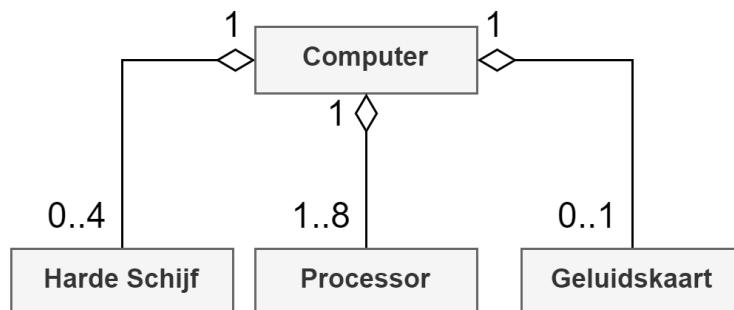
15.1.1 Associaties beschrijven

Compositie duiden we aan met een lijn die begint met een volle ruit aan de kant van de klasse die de objecten in zich heeft:



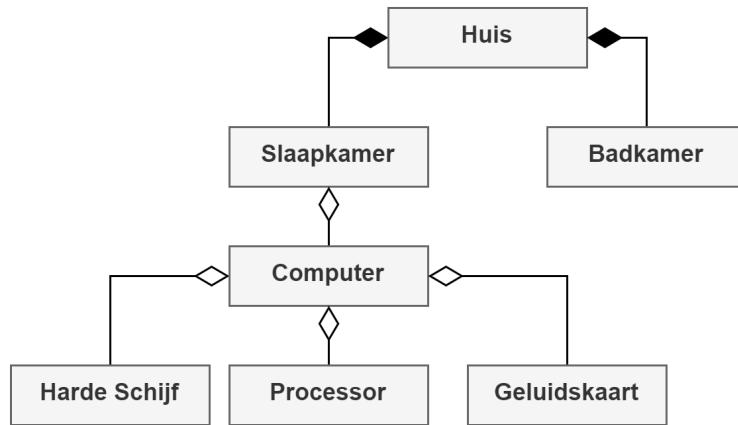
Figuur 15.1: Compositie: een huis heeft een slaapkamer en heeft een badkamer.

Aggregatie duiden we op exact dezelfde manier aan, maar de ruiten zijn niet gevuld. Optioneel duidt een getal aan iedere kant van de lijn de verhouding aan (zowel bij aggregatie als compositie), zodat we kunnen aangeven hoeveel (of geen) objecten het omliggende object kan hebben:



Figuur 15.2: Aggregatie: een computer heeft minstens 1 processor nodig, maar kan er tot 8 hebben. Ieder element kan echter ook op zichzelf bestaan.

Uiteraard zijn ook combinaties mogelijk. Stel je voor dat je een applicatie moet ontwerpen waarin je een reeks huizen moet bouwen, waarbij er in de slaapkamer steeds een computer moet gezet worden:



Figuur 15.3: Een computer kan je uit een brandend huis reden. De kamers van het huis zelf helaas niet.



Herinner je: overerving duiden we aan met een pijl die wijst naar de parent-klasse en duidt een “is een”-relatie aan.

15.1.2 Associatie in de praktijk

Het verschil tussen aggregatie en compositie is vooral van filosofische aard. In de praktijk zijn er weinig verschillen.

Eens kijken naar het voorbeeld van de computer en de harde schijf. We hebben twee klassen:

```

1 internal class PC
2 {
3 }
4 internal class HardeSchijf
5 {
6 }
  
```

Een *PC* heeft een *HardeSchijf*, dit wil zeggen dat we in de klasse *PC* een object (instantievariabele) van het type *HardeSchijf* zullen definiëren:

```

1 internal class PC
2 {
3     private HardeSchijf chardeSchijf;
4 }
  
```

In principe kunnen we nu zeggen dat we aggregatie hebben toegepast. Uiteraard moeten we nu deze HardeSchijf nog instantiëren anders zal deze de hele levensduur van ieder PC-object **null** zijn.

De instantie van een geaggregeerd object kan op verschillende manieren aangemaakt worden en is afhankelijk van wat je nodig hebt in je applicatie.



Associatie is net zoals overerving een belangrijk onderdeel van het OOP paradigma. Er is geen exacte oplossingsstrategie om associatie toe te passen: deze zal afhankelijk zijn van je specifieke probleem en oplossing. Staar je dus niet blind op deze voorbeelden: het is maar een greep uit de vele manieren waarmee je associateis kunt gebruiken.

15.1.2.1 Manier 1: Rechtstreeks de instantievariabele instellen

Wanneer we wensen dat iedere nieuwe PC ogenblikkelijk een interne harde schijf heeft dan kunnen we dit doen door ogenblikkelijk de instantievariabele een object te geven:

```

1 internal class PC
2 {
3     private HardeSchijf cHardeSchijf = new HardeSchijf();
4 }
```

15.1.2.2 Manier 2: Via de constructor(s)

Willen we echter bij het aanmaken van een nieuwe pc ook iets meer controle over wat voor harde schijf er wordt geïnstalleerd, dan kan dit ook via de constructors. We zouden dan bijvoorbeeld afhankelijk van bepaalde parameters in de (overloaded) constructors de schijf andere eigenschappen kunnen geven:

```

1 internal class PC
2 {
3     private HardeSchijf cHardeSchijf;
4
5     public PC(bool preinstallHD)
6     {
7         //enkel interne harde schijf indien klant voorinstallatie
8         // wenst
9         if(preinstallHD)
10             cHardeSchijf = new HardeSchijf();
11         else
12             cHardeSchijf == null;
13     }
}
```



De lijn `cHardeSchijf == null` is niet noodzakelijk, daar `cHardeSchijf` sowieso `null` zal zijn indien we niet in de `if` gaan.

Ik raad je toch aan dit altijd expliciet te doen. Hiermee zeg je nadrukkelijk: “als we via de overloaded constructor een PC aanmaken en er is geen preinstallatie vereist dan zit er geen harde schijf in de pc”. Het kan namelijk gebeuren dat voor we aan deze code komen er ondertussen iets voor heeft gezorgd dat `cHardeSchijf` alsnog een objectreferentie bevat. Door deze nu expliciet op `null` te zetten verwijderen we zeker de harde schijf als die er toch nog had ingezeten.

Heb je gezien hoe ik praat over deze preinstallatie alsof het om iets gaat dat in het echte leven gebeurt? Dit is bewust: het OOP paradigma draait om het feit dat het ons toelaat de realiteit zo dicht mogelijk te benaderen. Het helpt dan ook om je code (en probleemanalyse) steeds vanuit de context van de “echte wereld” te benaderen. Bijna ieder concept uit de echte wereld heeft een equivalent binnen C# als OOP-taal.

Het is een goede OOP oefening om af en toe in je omgeving eens rond te kijken, en wat je ziet vervolgens te vertalen naar een structuur van klassen, objecten en verbanden tussen die dingen (overerving, compositie, arrays en later ook nog polymorfisme en interfaces).

15.1.2.3 Manier 3: Properties

De vorige 2 voorbeelden waren eigenlijk voorbeelden van *compositie*. Wanneer de PC-objecten vernietigd worden (door de garbage collector) zullen ook de interne harde schijven verdwijnen.

Willen we echter via *aggregatie* de pc's bouwen, dan is het logischer dat we op een externe plaats de HardeSchijf objecten aanmaken. Nadat de PC werd aangemaakt zullen we de HardeSchijf in de PC plaatsen. We gebruiken hierbij properties om toegang tot de interne (geaggregateerde) variabele te verschaffen:

```
1 internal class PC
2 {
3     public HardeSchijf CHardeSchijf {get;set;}
4 }
```

Vervolgens kunnen we nu van buiten het object benaderen en er, als het ware, een nieuwe harde schijf in steken:

```
1 HardeSchijf mijnHardeSchijf = new HardeSchijf()  
2 PC mijnPC = new PC();  
3 mijnPC.CHardeSchijf = mijnHardeSchijf ;
```

Op deze manier hebben we nog steeds een referentie naar `mijnHardeSchijf` en zal de GC dit object dus niet verwijderen wanneer `mijnPC` wordt opgekust.



Kortom, nog steeds niets nieuws onder de zon. Alle manieren die ja al kende om met bestaande types objecten aan te maken gelden nog steeds. Compositie deed je al de hele tijd wanneer je bijvoorbeeld zei “een student heeft een geboortejaar” en dan een instantievariabele `int` geboortejaar aanmaakte. Het grote verschil is echter dat objecten moeten geïnstantieerd worden, wat niet moest met value-types en je dus iets vaker op `null` zal moeten controleren.

15.1.3 Associatie objecten gebruiken

Stel je voor dat de klasse `HardeSchijf` ook een auto-property `MaxCapacity` heeft. De klasse `PC` kan dankzij compositie dus nu ook die property gebruiken, zoals volgende voorbeeld toont:

```
1 internal class PC  
2 {  
3     private HardeSchijf cHardeSchijf = new HardeSchijf();  
4     public override string ToString()  
5     {  
6         return $"PC capaciteit HD: {cHardeSchijf.MaxCapacity} Gb";  
7     }  
8 }
```

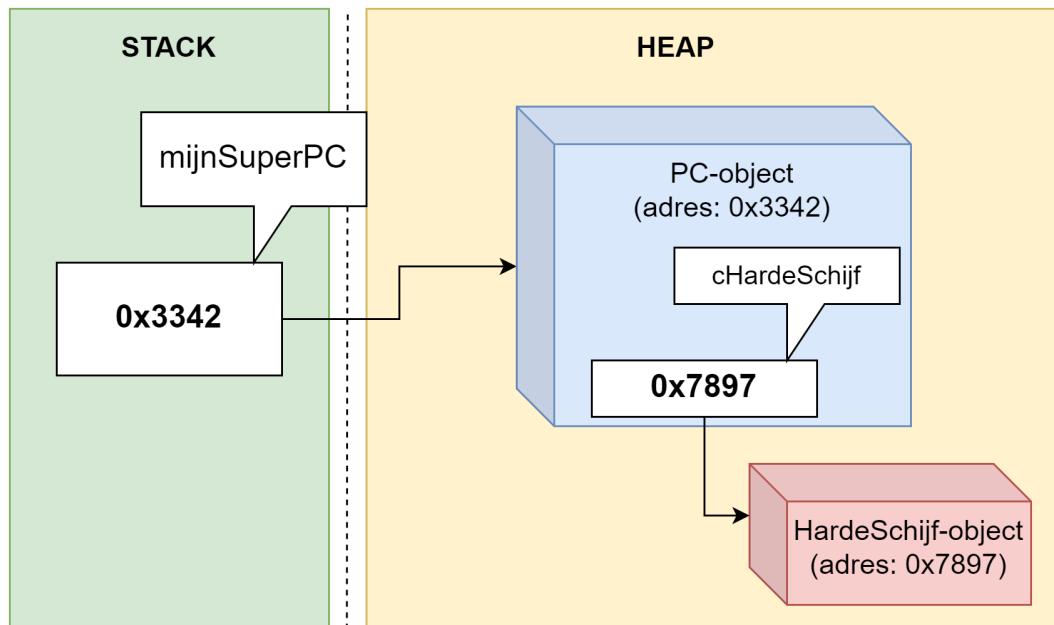
15.1.4 Associaties en referenties

Het moge duidelijk zijn: compositie/aggregatie en referenties horen samen. Maar hoe ziet dit er allemaal uit in het geheugen? Blij dat je het vraagt!

Wanneer we van voorgaande klasse een object aanmaken als volgt:

```
1 PC mijnSuperPC = new PC();
```

Dan zien we volgende “beeld”:



Figuur 15.4: Het is een erg nuttige *skill* indien je altijd dit soort tekening mentaal voorstelt, zodat je goed beseft waar welke informatie (en referenties) leven en wanneer die mogelijk door de GC gaan opgepeuzeld worden.

Compositie wil dus niet zeggen dat je in het geheugen grote *monolithische* objecten gaat hebben die het samengestelde object voorstellen. Nee, we blijven dankzij de kracht van referenties de boel apart houden.

Zoals je ziet is het belangrijk te beseffen dat bij compositie én aggregatie het *inner* object op zichzelf ergens zal gezet worden en dus niet *in* het parent-object komt. Alles dat we dus al wisten in verband met het doorgeven van referenties, de GC, enz. blijft dus nog steeds gelden.

Of zoals het hoofdstuk al begon: eigenlijk niets nieuws onder de zon!

15.1.4.1 NullReferenceException is een klassieke fout

Een veelvoorkomende fout bij compositie en aggregatie van objecten is dat je een intern object aanspreekt dat nooit werd aangemaakt. Je krijgt dan een `NullReferenceException`.

Het is dus zeker bij compositie en aggregatie een goede gewoonte om zoveel mogelijk te controleren op `null` telkens je het object gaat gebruiken:

```

1 public override string ToString()
2 {
3     string result = "Dit is een Intel i9.";
4     if(cHardeSchijf != null)
5         result += $"Capaciteit HD: {cHardeSchijf.MaxCapacity} Gb";
6     else
7         result += "Er is geen harde schijf aanwezig";
8     return result;
9 }
```

En uiteraard kan het ook nooit kwaad om alles in `try-catch` blokken te zetten, alleen is dat op detail-niveau niet werkbaar: je werkt met objecten en zal dus bijna de hele tijd code hebben waar `NullReferenceException` een potentieel gevaar is. Het is dus beter om vanaf de start je code zodanig te schrijven (met controles op `null`) dat er quasi geen uitzonderingen op `null` kunnen optreden.

15.2 “Heeft meerdere”- relatie

Wanneer een object meerdere objecten van een specifiek type heeft (denk maar aan een boek met pagina's) dan zullen we een array of een `List` als associatie-object gebruiken. Een voorbeeld:

```

1 internal class Pagina
2 {
3 }
4 internal class Boek
5 {
6     public Pagina[] AllePaginas {get;set;} = new Pagina[100];
7 }
```

Indien je nu een pagina wenst toe te voegen dan moet je ook deze individuele array-elementen nog instantiëren.

```

1 internal class Boek
2 {
3     public Pagina[] AllePaginas {get;set;} = new Pagina[100];
4     public void InsertPagina(Pagina paginaIn, int positie)
5     {
6         AllePaginas[positie] = paginaIn;
7     }
8 }
```

Een voorbeeld waarbij men vervolgens van buiten het object bestaande pagina's kan toevoegen:

```
1 Boek zieScherper = new Boek();
2 Pagina mijnDerdePagina = new Pagina();
3 zieScherper.InsertPagina(mijnDerdePagina, 2);
```

Of een voorbeeld met List:

```
1 internal class Boek
2 {
3     //SLECHT IDEE!
4     public List<Pagina> AllePaginas {get;set;} = new List<Pagina>();
5 }
```

Dit heeft als voordeel dat we de `Insert` methode van de `List`-klasse kunnen gebruiken en niet zelf nog moeten schrijven:

```
1 zieScherper.AllePaginas.Insert(new Pagina(), 5);
```

Dit voorbeeld met `List` is vanuit OOP-standpunt **geen goede oplossing**. Het vereist namelijk dat programmeurs die jouw klasse `Boek` gebruiken weten dat intern met een `List` wordt gewerkt.

We willen echter zo goed mogelijk een **blackbox** creëren, conform het abstractie-principe, die van buiten duidelijk en eenvoudig in gebruik is. Het is daarom beter om alsnog aan je `Boek` klasse een `Insert` methode toe te voegen. Dit geeft als extra verbetering dat we daarmee de `set` van onze lijst van pagina's **private** kunnen houden:

```
1 internal class Boek
2 {
3     public List<Pagina> AllePaginas {get; private set;} = new List<
4         Pagina>();
5     public void InsertPagina(Pagina paginaIn, int positie)
6     {
7         allPaginas.Insert(paginaIn, positie)
8     }
9 }
```

Pagina's voegen we nu als volgt toe:

```
1 zieScherper.InsertPagina(new Pagina(), 5);
```



Begrijp je nu waarom het geen goed idee is om een interne lijst gewoonweg via een property naar buiten beschikbaar te maken? Stel je voor dat het essentieel is dat de `AllePaginas` lijst NOoit leeggemaakt wordt. Jij als ontwikkelaar weet dit. Maar andere gebruikers van je klasse misschien niet. Zij kunnen echter zonder problemen `Clear()` via de property aanroepen, wat dus onverwachte gevolgen kan hebben!

15.3 Associatie of overerving?

Ik vertelde in het begin van dit hoofdstuk dat compositie en aggregatie een “heeft een”-relatie aanduiden, terwijl overerving een “is een”-relatie behelst. In de praktijk zal je v  l vaker compositie en aggregatie moeten gebruiken dan overerving. Associatie laat ons toe om 2 (of meer) totaal verschillende soorten zaken met elkaar te laten samenwerken, iets wat met overerving enkel kan indien beide zaken een “is een”-relatie hebben. Dit zien we ook in de echte wereld: de zaken rondom ons zullen vaker een compositie/aggregatie-relatie hebben dan een overervings-relatie.

Zoals je hopelijk beseft kan dus alles een compositieobject zijn in een ander object. Denk maar aan een Dictionary van klanten die je gebruikt in een klasse Winkel. Of wat te denken van de klasse Mens die uit een hele boel organen bestaat. Ieder orgaan is compositie-object in de klasse Mens, zoals 2 Nier-objecten, een Hersenen instantie, 1 Hart instantie enz. Iemand die in jouw Mens-simulator een nieuw hart nodig heeft kan dat dan dankzij manier 3, via een property ingeplant krijgen:

```
1 Mens patient = new Mens();
2 Mens donor = new Mens();
3 //Donor heeft een tragisch ongeluk en sterft
4 //Operatie start
5 patient.Hart = null; //vorig hart wordt "verwijderd"
6 patient.Hart = donor.Hart;
7 donor = null //donor wordt begraven
```



Let er wel op dat je niet overal compositie begint toe te passen alsof je de Dokter Frankenstein van C# bent. Hoe meer compositie (of aggregatie) je toepast in een klasse, hoe specifieker die soms wordt, en daardoor mogelijk minder herbruikbaar. Het is om die reden dat we verderop interfaces gaan ontdekken om ervoor te zorgen dat 2 of meerdere klassen minder “op/in elkaar gelijmd” zitten ten gevolge van bijvoorbeeld een nogal hechte compositie.

15.4 Het this keyword

Je zult in je zoektocht naar online antwoorden mogelijk al een paar keer het **this** keyword zijn tegengekomen. **Dit keyword kan je aanroepen in een object om de referentie van het object terug te krijgen.** Met andere woorden: het laat toe dat een object “zichzelf” kan aanroepen. Dat klinkt vreemd, maar heeft 3 duidelijke gebruiken:

- Het laat toe dat een object zichzelf kan meegeven als actuele parameter aan een methode.
- Het laat toe instantievariabelen en properties aan te roepen van het object die mogelijk dezelfde naam hebben als een lokale variabele.
- We kunnen een andere constructor vanuit een constructor aanroepen zoals reeds gezien (in hoofdstuk 11).

15.4.1 Aanroepen van instantievariabelen met zelfde naam

Wanneer je **this** gebruikt binnen een klasse, dan zal je zien dat bij het schrijven van de dot-operator je ogenblikkelijk de volledige interne structuur van de klasse kunt bereiken:

```
class Monster
{
    0 references
    public int Levens { get; set; }

    0 references
    public int Kracht { get; set; }

    0 references
    public void Aanval()
    {
        this.|
    }
}
```

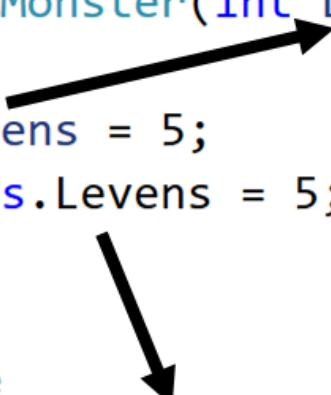
The screenshot shows a code editor with a tooltip displayed over the 'this.' prefix. The tooltip lists several members of the 'Monster' class, including 'Aanval', 'Equals', 'GetHashCode', 'GetType', 'Kracht', and 'Levens'. This demonstrates that the 'this' keyword provides access to all members of the class, regardless of their visibility (public, private, protected).

Figuur 15.5: Met this zie je letterlijk alles dat de klasse heeft aan te bieden, ongeacht de access modifiers.

Enerzijds ben je vrij om altijd **this** te gebruiken wanneer je eender wat van de klasse zelf wilt bereiken. Vooral in oudere code-voorbeelden zal je dat nog vaak zien gebeuren.

Anderzijds laat **this** ook toe om properties, methoden en instantievariabelen aan te roepen wanneer die mogelijk op de huidige plek niet aanroepbaar zijn omdat hun naam conflicteert met een lokale variabele dat dezelfde naam heeft:

```
class Monster
{
    0 references
    public Monster(int Levens)
    {
        Levens = 5;
        this.Levens = 5;
    }
}
1 reference
public int Levens { get; set; }
```



Figuur 15.6: Bij conflicterende namen binnen dezelfde scope zal this ons helpen om toch buiten de huidige methode aan een gelijknamig element te geraken.

De lijn `Levens = 5;` in de constructor zal de parameter zelf van waarde aanpassen (wat niet wordt aangeraden). Terwijl door **this** te gebruiken geraak je aan de property met dezelfde naam.



Merk op dat qua naamgeving de keuze van de formele parameter `Levens` in de constructor sowieso een ongelukkige keuze is in dit voorbeeld.

15.4.2 Object geeft zichzelf mee als parameter

Beeld je in dat je volgende Management klasse hebt die toelaat om Werknemer objecten te controleren of ze promoveerbaar zijn of niet. Het management van de firma heeft beslist dat werknemers enkel kunnen promoveren als hun huidige Rang lager is dan 10:

```

1 internal class Management
2 {
3     private const int MAXRANG = 10;
4     public static bool MagPromoveren(Werknemer toCheck)
5     {
6         return toCheck.Rang < MAXRANG;
7     }
8 }
```

Dankzij het **this** keyword kan je nu vanuit de klasse Werknemer deze externe methode aanroepen om zo te kijken of een object al dan niet kan promoveren:

```

1 internal class Werknemer
2 {
3     public int Rang { get; set; }
4     public bool IsPromoveerbaar()
5     {
6         return Management.MagPromoveren(this);
7     }
8 }
```

Op deze manier geeft het object waarop je `IsPromoveerbaar` op aanroeft zichzelf mee als actuele parameter aan `Management.MagPromoveren()`. Dit laat dus toe dat een werknemer zelf kan weten of hij of zij al dan niet kan promoveren:

```

1 Werknemer francis = new Werknemer();
2 if(francis.IsPromoveerbaar())
3 {
4     Console.WriteLine("Jeuj!");
5 }
```


16 Polymorfisme

En zo komen we eindelijk aan de vierde grote pijler van OOP. Weet je nog waar A,I en E voor stonden in **A PIE**? Nu gaan we dus de **P** van **Polymorfisme (polymorphism)** aanpakken.

De latijnse naam polymorfisme bestaat uit 2 delen: *poly* en *morfisme*, letterlijk dus “meerdere vormen”. En geloof het of niet, deze naam dekt de lading ongelooflijk goed.

Polymorfisme laat ons toe dat objecten kunnen behandeld worden als objecten van de klasse waar ze van overerven. Dit klinkt logisch, maar zoals je zo meteen zal zien zal je hierdoor erg krachtige code kunnen schrijven. Anderzijds zorgt polymorfisme er ook voor dat het **virtual** en **override** concept bij methoden en properties ook effectief werkt. Het is echter vooral de eerste eigenschap waar ik in dit hoofdstuk dieper op in ga.

16.1 De “is een”-relatie in actie

Dankzij overerving kunnen we “is een”-relaties beschrijven. Soms is het echter handig dat we alle child-objecten als dat van hun parent kunnen beschouwen. Beeld je in dat je een gigantische klasse-hiërarchie hebt gemaakt, maar uiteindelijk wil je wel dat alle objecten een bepaalde property aanpassen die ze gemeenschappelijk hebben. Zonder polymorfisme is dat een probleem.

Stel dat we een aantal van Dier afgeleide klassen hebben die allemaal op hun eigen manier een geluid voortbrengen:

```
1 internal abstract class Dier
2 {
3     public abstract string MaakGeluid();
4 }
5 internal class Paard: Dier
6 {
7     public override string MaakGeluid()
8     {
9         return "Hinnikhinnik";
10    }
11 }
12 internal class Varken: Dier
13 {
14     public override string MaakGeluid()
15     {
16         return "Oinkoink";
17     }
18 }
```

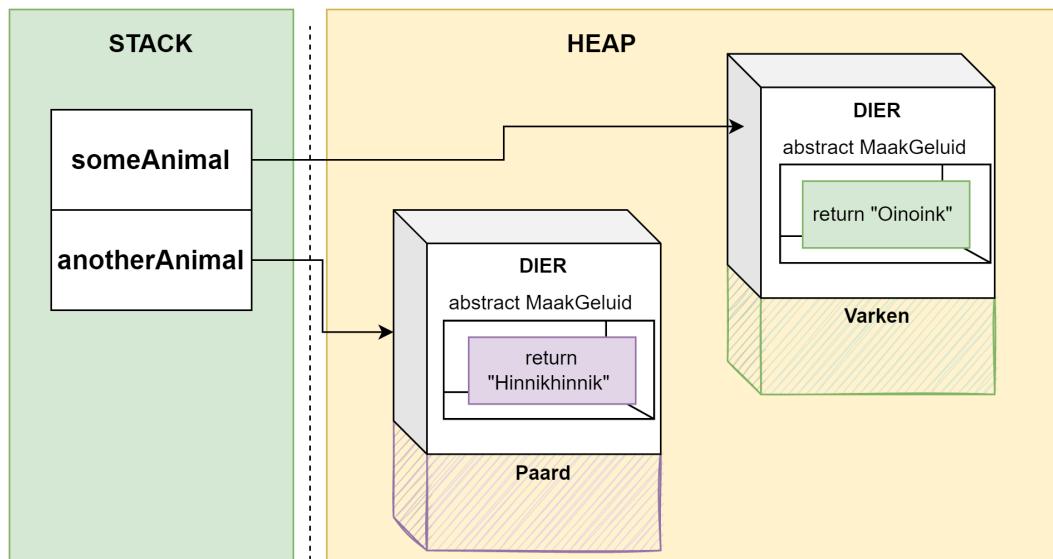
Met de hulp van polymorfisme kunnen we nu elders objecten van Paard en Varken in een variabele van het type Dier bewaren, maar toch hun eigen geluid laten reproduceren:

```

1 Dier someAnimal = new Varken();
2 Dier anotherAnimal = new Paard();
3 Console.WriteLine(someAnimal.MaakGeluid()); //Oinkoink
4 Console.WriteLine(anotherAnimal.MaakGeluid()); //Hinnikhinnik

```

Alhoewel er een volledig Varken en Paard object in de heap wordt aangemaakt (en blijft bestaan), zullen variabelen van het type Dier enkel die dingen kunnen aanroepen die in de klasse Dier gekend zijn. Dankzij **override** zorgen we er echter voor dat MaakGeluid wel die code uitvoert die specifiek bij het child-type hoort.



Figuur 16.1: Het gearceerde deel is niet bereikbaar voor de 2 variabelen in de stack daar deze van het type Dier zijn.



Het is belangrijk te beseffen dat `someAnimal` en `anotherAnimal` van het type Dier zijn en dus enkel die dingen kunnen die in Dier beschreven staan. Enkel zaken die **override** zijn in de child-klasse zullen met de specialisatie-code werken.

16.2 Objecten en polymorfisme

Kortom, polymorfisme laat ons toe om referenties (naar objecten van een child-type) toe te wijzen aan een variabele van het parent-type (**upcasting**).

Dit wil ook zeggen dat dit mag (daar alles overerft van `System.Object`):

```
1 System.Object mijnObject = new Varken();
```

Alhoewel `mijnObject` effectief een `Varken` is (in het geheugen), kunnen we enkel aan dat gedeelte dat in de klasse `System.Object` staat beschreven (`ToString`, `Equals` enz.). Als we het varken toch geluid willen laten maken, dan zal dat niet werken!

16.3 Arrays en polymorfisme

Arrays en lijsten laten heel krachtige code toe dankzij polymorfisme. Je kan een lijst van de basis-klasse maken en deze vullen met allerlei objecten van de basis-klasse **én de child-klassen**.

Een voorbeeld:

```
1 List<Dier> zoo = new List<Dier>();
2 zoo.Add(new Varken());
3 zoo.Add(new Paard());
4 foreach(var dier in zoo)
5 {
6     Console.WriteLine(dier.MaakGeluid());
7 }
```

We hebben nu een manier gevonden om onze objecten op de juiste momenten even als één geheel te gebruiken, zonder dat we verplicht zijn dat ze allemaal van hetzelfde type zijn!



Polymorfisme is een heel krachtig concept. Door een referentie naar een object te bewaren in een variabele van z'n basistype en, wanneer nodig, ze als 'zichzelf' te gebruiken wordt je code een pak eenvoudiger.

Vaak weet je niet op voorhand wat voor elementen je in je lijst wilt plaatsen. Via polymorfisme lossen we dit op. Stel dat je een `List<Person>` hebt waar echter elementen van subklassen (`Bakker`, `Student`, enz.) in terecht kunnen komen. Polymorfisme laat gewoon toe om ook deze elementen in die lijst te plaatsen.

16.3.1 Een wereld met OOP: Pong polymorfisme

Ik hadd al een klein tipje van de sluier gelicht toen ik overerving introduceerde in Pong. Met de hulp van overving konden we plots veel toffere balletjes ontwikkelen zoals het CentreerBalletje. Wel, met de hulp van polymorfisme kan dit op de koop toe zonder dat we ons hoofdprogramma moeten aanpassen. In principe kunnen we nog steeds werken met `List<Balletje>`, maar maakt het niet uit wat voor balletjes we er in plaatsen: polymorfisme zal de boel draaidende houden.

Beeld je in dat we naast CentreerBalletje ook nog de klassen InstabielBalletje (dat op random momenten z'n vectoren op 0 zet) en TeleportBalletje (dat elke 10 ticks naar een random plek op het scherm *teleporteert*) hebben. De code om deze balletjes te instantiëren en in de lijst te plaatsen wordt verrassend eenvoudig:

```
1 List<Balletje> veelBalletjes = new List<Balletje>();
2
3 veelBalletjes.Add(new Balletje());
4 veelBalletjes.Add(new CentreerBalletje());
5 veelBalletjes.Add(new Balletje());
6 veelBalletjes.Add(new InstabielBalletje());
7 veelBalletjes.Add(new TeleportBalletje());
```

We hebben nu 5 balletjes: 2 gewone, en een aantal van de 3 nieuwe types elk. Alle overige code verderop, waarin we de lijst doorlopen en de Update en TekenOpScherm aanroepen van ieder balletje, moet niet aangepast worden.

16.4 Polymorfisme in de praktijk

Beeld je in dat je een klasse `EersteMinister` hebt met een methode `Regeer` en je wilt een eenvoudig land simuleren.

De `EersteMinister` heeft toegang tot de ministers die hem kunnen helpen (inzake milieu, binnenlandse zaken (BZ) en economie). Zonder de voordelen van polymorfisme zou de klasse `EersteMinister` er zo kunnen uitzien (**slechte manier!**):

```
1 internal class EersteMinister
2 {
3     public MinisterVanMilieu Jansens {get;set;} = new
4         MinisterVanMilieu();
5     public MinisterBZ Ganzeweel {get;set;} = new MinisterBZ();
6     public MinisterVanEconomie VanCent {get;set;} = new
7         MinisterVanEconomie();
8
9     public void Regeer()
10    {
11        // ministers stappen binnen en zeggen wat er moet gebeuren
12        // Jansens: Problematiek aangaande bos dat gekapt wordt
13        Jansens.VerhoogBosSubsidies();
14        Jansens.OpenOnderzoek();
15        Jansens.ContacteerGreenpeace();
16
17        // Ganzeweel advies omtrent rel aan grens met Nederland
18        Ganzeweel.VervangAmbassadeur();
19        Ganzeweel.RoepTroepenmachtTerug();
20        Ganzeweel.VerhoogRisicoZoneAanGrens();
21
22        // Van Cent geeft advies omtrent nakende beurscrash
23        VanCent.InjecteerGeldInMarkt();
24        VanCent.VerlaagWerkloosheidsPremie();
25    }
26 }
```

Dit voorbeeld is gebaseerd op een briljante StackOverflow post waarin de vraag “*What is polymorphism, what is it for, and how is it used?*” wordt behandeld¹.

¹[stack overflow.com/questions/1031273/what-is-polymorphism-what-is-it-for-and-how-is-it-used](https://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-for-and-how-is-it-used)

De MinisterVanMilieu zou er zo kunnen uitzien (de methodenimplementatie mag je zelf verzinnen):

```
1 internal class MinisterVanMilieu
2 {
3     public void VerhoogBosSubsidies(){}
4     public void OpenOnderzoek(){}
5 }
```

De MinisterVanEconomie-klasse heeft dan weer heel andere publieke methoden. En de MinisterBZ ook weer totaal andere.

Je merkt dat de EersteMinister-klasse aardig wat specifieke kennis moet hebben van de vele verschillende departementen van het land. Bovenstaande code is dus zeer slecht en vloekt tegen het abstractie-principe van OOP: onze klasse moeten veel te veel weten van andere klassen, wat vermeden moet worden. Telkens er zaken binnen een specifieke ministerklasse wijzigen moet dit ook in de EersteMinister aangepast worden. **Dankzij polymorfisme en overerving kunnen we dit alles veel mooier oplossen!**

Ten eerste: We verplichten alle ministers dat ze overerven van de abstracte klasse Minister die maar 1 abstracte methode heeft Adviseer:

```
1 internal abstract class Minister
2 {
3     abstract public void Adviseer();
4 }
5
6 internal class MinisterVanMilieu:Minister
7 {
8     public override void Adviseer()
9     {
10         VerhoogBosSubsidies();
11         OpenOnderzoek();
12         ContacteerGreenpeace();
13     }
14     private void VerhoogBosSubsidies(){ ... }
15     private void OpenOnderzoek(){ ... }
16     private void ContacteerGreenpeace(){ ... }
17 }
18
19
20 internal class MinisterBZ:Minister {}
21 internal class MinisterVanEconomie:Minister {}
```

Ten tweede: Het leven van de EersteMinister wordt plots véél makkelijker. Hij kan gewoon de Adviseer methode aanroepen van iedere minister:

```

1 internal class EersteMinister
2 {
3     public MinisterVanMilieu Jansens {get;set;} = new MinisterVanMilieu
4         ();
5     public MinisterBZ Ganzeweel {get;set;} = new MinisterBZ();
6     public MinisterVanEconomie VanCent {get;set;} = new
7         MinisterVanEconomie();
8
9     public void Regeer()
10    {
11        Jansens.Adviseer();
12        Ganzeweel.Adviseer();
13        VanCent.Adviseer();
14    }
15 }
```

En ten derde: En we kunnen hem nog helpen door met een array of `List<Minister>` te werken zodat hij ook niet steeds de “namen” van z’n ministers moet kennen. Dankzij polymorfisme mag dit:

```

1 internal class EersteMinister
2 {
3     public List<Minister> AlleMinisters {get;set;} = new List<Minister>();
4     public EersteMinister()
5     {
6         AlleMinisters.Add(new MinisterVanMilieu());
7         AlleMinisters.Add(new MinisterBZ());
8         AlleMinisters.Add(new MinisterVanEconomie());
9     }
10    public void Regeer()
11    {
12        foreach (Minister minister in AlleMinisters)
13        {
14            minister.Adviseer();
15        }
16    }
17 }
```

En wie zei dat het regeren moeilijk was?!



Merk op dat dit voorbeeld ook goed gebruik maakt van **compositie**.

16.5 De **is** en **as** keywords

Dankzij polymorfisme kunnen we dus child en parent-objecten door elkaar gebruiken. De keywords **is** en **as** gaan ons helpen om door het bos van objecten het bos nog te zien.

16.5.1 Het **is** keyword

Het **is** keyword is een operator die je kan gebruiken om te weten te komen of:

- Een object van een bepaalde datatype is.
- Een object een bepaalde interface bevat (zie volgende hoofdstuk).

De **is** operator heeft twee operanden nodig en geeft een **bool** terug als resultaat. De linkse operator moet een variabele zijn, de rechtse een datatype. Bijvoorbeeld:

```
1 bool ditIs EenStudent = mijnStudent is Student;
```

16.5.1.1 **is** voorbeeld

Stel dat we volgende drie klassen hebben:

```
1 internal class Voertuig {}
2
3 internal class Auto: Voertuig{}
4
5 internal class Persoon {}
```

Een Auto **is** een Voertuig. Een Persoon **is géén** Voertuig.

Stel dat we enkele variabelen hebben als volgt:

```
1 Auto mijnAuto = new Auto();
2 Persoon rambo = new Persoon();
```

We kunnen nu de objecten met **is** bevragen of ze van een bepaalde type zijn:

```
1 if(mijnAuto is Voertuig)
2 {
3     Console.WriteLine("mijnAuto is een Voertuig");
4 }
5 if(rambo is Voertuig)
6 {
7     Console.WriteLine("rambo is een Voertuig");
8 }
```

De uitvoer zal worden: `mijnAuto is een Voertuig.`

Met polymorfisme wordt dit voorbeeld echter interessanter. Wat als we een hoop objecten in een lijst van voertuigen plaatsen en nu enkel met de auto's iets willen doen, dan kan dat:

```

1 List<Voertuig> alleMiddelen = new List<Voertuig>();
2 alleMiddelen.Add(new Voertuig());
3 alleMiddelen.Add(new Auto());
4 alleMiddelen.Add(new Voertuig());
5
6 foreach (var middel in alleMiddelen)
7 {
8     if(middel is Auto)
9     {
10         //Doe iets met het huidige voertuig
11     }
12 }
```

16.5.2 as keyword met voorbeeld

Wanneer we objecten van het ene naar het andere type willen omzetten dan doen we dit vaak met behulp van casting:

```

1 Student fritz = new Student();
2 Mens jos = (Mens)fritz;
```

Het probleem bij casting is dat dit niet altijd lukt. Indien de conversie niet mogelijk is zal een uitzondering gegenereerd worden en je programma zal crashen als je niet aan exception handling doet.

Het **as** keyword lost dit op. Het keyword zegt aan de compiler “**probeer dit object te converteren.**

Als het niet lukt, zet het dan op null in plaats van een uitzondering op te werpen.”

De code van daarnet herschrijven we dan naar:

```

1 Student fritz = new Student();
2 Mens jos = fritz as Mens;
```

Indien nu de casting niet lukt (omdat Student misschien geen childklasse van Mens blijkt te zijn) dan zal jos de waarde **null** krijgen.

We kunnen dan vervolgens schrijven:

```

1 Student fritz = new Student();
2 Mens jos = fritz as Mens;
3 if(jos != null)
4 {
5     //Doe Mens-zaken
6 }
```

16.5.3 Volgorde van bewerkingen met **is** en **as**

De **is** en **as** keywords worden gebruikt in logische expressie. Ze hebben dan ook een bepaalde volgorde wanneer ze verwerkt zullen worden. Onze bestaande volgorde van bewerkingen krijgt dus 2 nieuwe leden op lijn 4:

1. Logische NIET: !
2. Delen en vermenigvuldigen: *, /, %
3. Optellen en aftrekken: +, -
4. Relationale operators: <, <=, >, >= én **is, as**
5. Gelijkheid: ==, !=
6. Logische EN: &&
7. Logische OF: ||

16.6 Is, as en polymorfisme: een krachtige bende

Dankzij polymorfisme hebben we nu met de **is** en **as** keywords handige hulpmiddelen om meer “generieke” methoden te schrijven. Herinner je je nog de `Equals` methode die we schreven om 2 studenten te vergelijken toen we leerden dat alle klassen van `System.Object` overerfden? Laten we deze code er nog eens bijnemen en verbeteren:

```

1 //In de Student klasse
2 public override bool Equals(Object o)
3 {
4     Student temp = (Student)o;
5     return (Geboortejaar == temp.Geboortejaar && Voornaam == temp.
6             Voornaam);
7 }
```

De eerste lijn waarin we `o` casten naar een student kan natuurlijk mislukken. Het is dan ook veiliger om eerst te controleren of we wel mogen casten, voor we het effectief doen. Hierdoor schrijven we een minder foutgevoelige methode:

```

1 //In de Student klasse
2 public override bool Equals(Object o)
3 {
4     if(o is Student)
5     {
6         Student temp = o as Student;
7         return (Geboortejaar == temp.Geboortejaar && Voornaam == temp.
8                 .Voornaam);
9     }
10    return false;
11 }
```

Of we kunnen ook het volgende doen:

```

1 //In de Student klasse
2 public override bool Equals(Object o)
3 {
4     Student temp = o as Student;
5     if(temp != null)
6     {
7         return (Geboortejaar == temp.Geboortejaar && Voornaam == temp.
8                 .Voornaam);
9     }
10    return false;
11 }
```

Beide zijn geldige oplossingen.



De **is** en **as** keywords laten toe om meer dynamische code te schrijven. Mogelijk weet je niet op voorhand wat voor datatype je code zal moeten verwerken en wordt polymorfisme je oplossing. Maar dan? Dan komen **is** en **as** to the rescue!

Je met polymorfisme gevulde lijst van objecten van allerhande typen wordt nu beheersbaarder. Je kan nu met **is** een element bevragen of het van een bepaald type is. Vervolgens kan je met **as** het element tijdelijk ‘omzetten’ naar z’n effectieve type. Bijgevolg kan dit element dan doen dan wanneer hij kan in de vermomming is van z’n eigen basistype.

17 Interfaces

Interfaces in de echte wereld

De naam interface kan je letterlijk vertalen als “tussen vlakken”. Een interface is de verbinding tussen 2 systemen, van welke vorm ook. In de echte wereld gebruik je constant interfaces. Telkens je met de auto rijdt gebruik je een interface: namelijk een handvol handelingen om de auto te laten rijden (pedalen, stuur, enz.). Bijna alle auto's hanteren dezezelfde interface.

Van zodra je de interface kent en begrijpt kan je die overal gebruiken, zonder dat je moet weten wat er in het systeem intern juist gebeurt (als ik de gaspedaal indruw boeft het niet of ik op gas of elektrisch rijd, zolang de auto maar voortbeweegt).

Aan de achterkant van je computer (en ook in de pc zelf) zijn tal van hardware-interfaces. Afgesproken manieren om 2 systemen met elkaar te laten communiceren. Zo heb je de USB-aansluiting die toelaat dat een extern systeem met een usb-aansluiting met de computer kan communiceren. Maar ook de HDMI, audio, en andere aansluitingen hanteren interfaces. Zouden er rond deze zaken geen wereldwijde interfaces zijn afgesproken, dan zou je mogelijk telkens op een andere manier je externe harde schijf aan een computer moeten hangen.



Voor je dolenthousiast wordt, denkende dat je eindelijk grafische applicaties (GUI oftewel Graphical User Interface applicaties) gaat maken, moet ik je helaas teleurstellen. Dit hoofdstuk behandelt het programmeer-concept *interfaces* wat eigenlijk niets te maken heeft met User Interfaces. U weze gewaarschuwd.

Interfaces in OOP

Dit concept van interfaces uit de echte wereld heeft ook een OOP variant. Namelijk de interface tussen 2 (of meer) klassen. Door te beloven dat een klasse aan een bepaalde interface voldoet kunnen alle klassen die deze interface “kennen” met elkaar praten. **Een interface in OOP is een beschrijving van publieke methoden en properties die de klasse belooft te hebben.** Net zoals je een fotocamera kunt kopen die de HDMI en USB-interface heeft, zo ook kan je nu een klasse maken die bijvoorbeeld de interfaces ISecure en IStreamable heeft.



Interfaces zijn als het ware stempels die we op een klasse kunnen plakken om zo te zeggen “*deze klasse gebruikt interface xyz*”. Gebruikers van de klasse hoeven dan niet de hele klasse uit te spitten en weten dat alle klassen met interface xyz dezelfde publieke properties en methoden hebben.



Een interface is niet meer dan een belofte: het zegt enkel welke publieke methoden en properties de klassen bezit. **Het zegt echter niets over de effectieve code/implementatie van deze methoden en properties.**

17.0.1 Interfaces in C#

Een interface is dus eigenlijk als het ware een klein stukje papier waar je op zet “om aan deze interface te voldoen moet je zeker volgende methoden en properties hebben”. Kortom, een interfacebestand meestal een vrij klein bestand. Het is letterlijk de “Dit apparaat is USB 3.0 compatibel”-sticker.

Stel dat we deze interface kunnen we gebruiken in een spel vechtspel tussen karakters, waarin sommige van de klassen ook aan de Superhelden-interface moeten voldoen. Volgende code toont hoe we een interface definiëren in C#:

```
1 interface ISuperHeld
2 {
3     void SchietLasers();
4     int VerlaagKracht(bool isZwak);
5     int Power{get; set;}
6 }
```

Enkele opmerkingen hierbij zijn op z'n plaats:

- Het woord **class** wordt niet gebruikt, in de plaats daarvan gebruiken we **interface**.
- Het is een goede gewoonte om interfaces met een **I** te laten starten in hun naamgeving.
- Methoden en properties gaan niet vooraf van **public**: interfaces zijn van nature al publiek, dus alle methoden en properties van de interface zijn dat bijgevolg ook (uiteindelijk geldt dit niet voor andere methoden in de klassen, deze mogen nog steeds **private** zijn als dat nodig is).
- Er wordt geen code/implementatie gegeven: iedere methode eindigt ogenblikkelijk met een puntkomma.

Het is in de klassen waar we deze interface “aanhangen”, dat we nu vervolgens verplicht zijn deze methode en properties te implementeren.



Ook abstracte klassen kunnen één of meerdere interfaces hebben. In het geval van een abstracte klasse is deze niet verplicht de interface ook al te implementeren, en mag (delen van) de interface ook als abstract aangeduid worden.



Een interface is een beschrijving hoe een component een andere component kan gebruiken, zonder te zeggen hoe dit moet gebeuren. De interface is met andere woorden 100% scheiding tussen de methode/property-signatuur en de eigenlijke implementatie ervan.

17.0.1.1 Interface regels

Interfaces zijn als het ware standaarden waaraan een klasse moet voldoen, wil het kunnen zeggen dat het een bepaalde interface heeft. Standaarden impliceert dat er duidelijke afspraken nodig zijn. Bij C# interfaces zijn er enkele belangrijke regels:

- Je kan geen instantievariabelen declareren in een interface (dat hoort bij de implementatie).
- Je kan geen constructors declareren.
- Je kan geen access modifiers specificeren: alles is **public**¹.
- Je kan nieuwe types (bv. **enum**) in een interface declareren.
- Een interface kan niet overerven van een klasse, wel van één of meerdere interfaces.
- Een interface mag géén code bevatten.



De laatste regel, “een interface mag géén code bevatten”, is deels onwaar. Sinds C# 8.0 bestaan *default interface methods*. Dit zijn interface-methoden die standaardimplementaties bevatten. Deze implementaties worden gebruikt wanneer de methode niet is geïmplementeerd in de klasse die de interface overneemt. Ik behandel deze hier niet en raad je om interfaces te leren gebruiken waar ze voor bedoeld waren: property- en methodesignaturen zonder code.

¹In recentere versies van C# (sinds 8.0) is het nu wel toegestaan om **public** voor een methode of property te plaatsen. Dit verbeterd de leesbaarheid (daar we gewoon zijn dat het weglaten van een *access modifier* eigenlijk **private** betekent, wat bij interfaces dus niet zo is).

17.1 Interfaces en klassen

We kunnen nu aan klassen de stempel ISuperHeld geven zodat programmeurs weten dat die klasse gegarandeerd de methoden SchietLasers, VerlaagKracht en de property Power zal hebben.

Volgende code toont dit. We plaatsen de interface (of interfaces) die de klasse beloofd te hebben achter het dubbele punt bovenaan.

```
1 internal class Zorro: ISuperHeld
2 {
3     public void RoepPaard(){...}
4     public bool HeeftSnor{get;set;}
5
6     public void SchietLasers() //interface ISuperHeld
7     {
8         Console.WriteLine("pewpew");
9     }
10
11    public int VerlaagKracht(bool isZwak)//interface ISuperHeld
12    {
13        if(isZwak)
14        {
15            return 5;
16        }
17        return 10;
18    }
19    public int Power {get;set;} //interface ISuperHeld
20 }
```

Zolang de klasse Zorro niet exact de interface inhoud implementeert zal deze klasse niet gecompileerd kunnen worden.

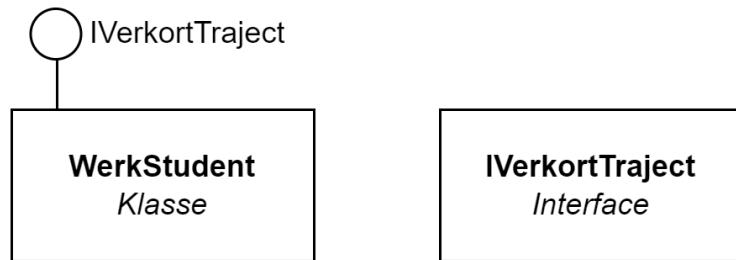
De klasse in dit voorbeeld blijft wel overerven van System.Object. Het is ook perfect mogelijk om een klasse te hebben die één overerft van een specifieke klasse én meerdere interfaces heeft:

```
1 internal class DarthVader: StarWarsCharacter, IForceUser, IPilot
```

17.1.1 Interfaces in UML

Een “lolly” op een klasse geeft aan dat deze een bepaalde interface heeft in UML notatie. In volgende tekening hebben we een klasse `WerkStudent` en een interface `IVerkortTraject`.

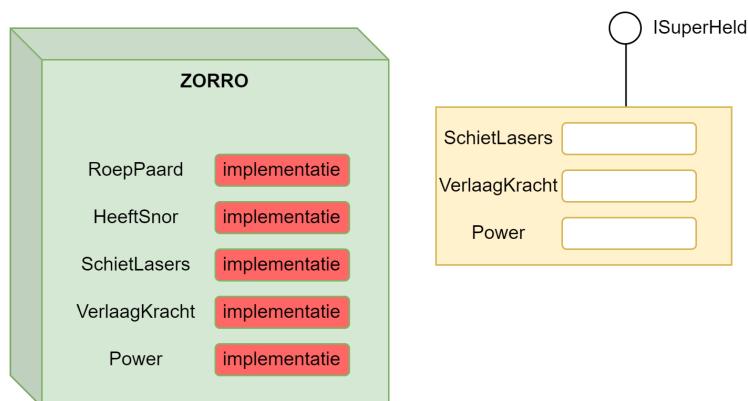
We gebruiken de UML notatie voor een interface om aan te geven dat de `Student` klasse de `IVerkortTraject` interface heeft:



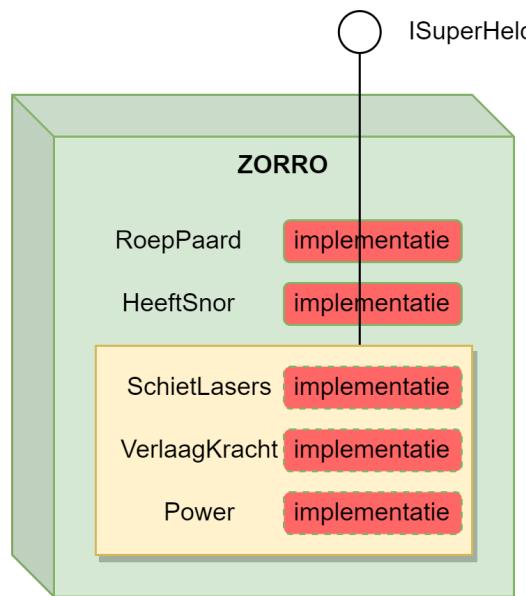
Figuur 17.1: Interface UML notatie.

17.1.1.1 Interfaces visualiseren

Een visuele manier om interfaces voor te stellen is de volgende. Eigenlijk is een interface als het ware een blad papier dat je bovenop je klasse kunt houden. Op het blad staan de methoden en properties beschreven die de interface moet hebben. Als je het blad mooi bovenop een klasse plaatst die de interface belooft te doen, dan zouden de gaten in het blad mooi bovenop de respectievelijke methoden en properties van de klasse passen.



Figuur 17.2: Het UML “lolly’tje” kan je als een haakje beschouwen waaraan de interface bengelt.



Figuur 17.3: Vervolgens kunnen we de interface met het haakje aan de klasse hangen.



Ik zei net: “*Volgende interface kunnen we gebruiken in een spel waarin sommige klassen superhelden zijn.*” Die zin impliceert toch overerving “sommige klassen **zijn** superhelden”?

Dat klopt, maar zoals we weten kan je maar van 1 klasse overerven. Beeld je in dat je een uitgebreide klasse-hiërarchie hebt gemaakt bestaande uit monsters, mensen, huizen en voertuigen. Deze 4 groepen hebben mogelijk geen gemeenschappelijke parent, maar toch willen we dat sommige monsters superhelden kunnen worden, net zoals sommige mensen EN zelfs enkele voertuigen (Transformers!).

Dankzij interface kunnen we als het ware een stukje de beperking dat je maar van 1 klasse kunt overerven opvangen. Sommige klassen ZIJN een voertuig MAAR OOK een Superheld. Met andere woorden, klassen kunnen meerdere interfaces implementeren.

Merk wel op dat de interface NIET de implementatie bevat van wat een superheld juist doet. Het gaat enkel beloven dat de klasse bepaalde methoden en properties heeft.

17.1.2 Lampje wederom to the rescue

Wanneer je in VS een klasse schrijft die een bepaalde interface moet hebben, dan kan je die snel implementeren. Je schrijft de klasse-signatuur en klikt er dan op: links verschijnt het *lampje* waar je vervolgens op kunt klikken en kiezen voor “Implement interface”. En presto!



Figuur 17.4: Als het lampje niet ogenblikkelijk verschijnt kan je ook altijd rechterklikken op het rood onderstreepte woord en kiezen voor “Quick Actions and Refactorings...”.

Merk op dat VS de nieuwere EBM syntax hier hanteert bij properties. Meer informatie hierover vind je in de appendix.

17.2 Het is keyword met interfaces

We kunnen **is** gebruiken om te weten of een klasse een specifieke interface heeft. Dit laat ons toe om code te schrijven die weer een beetje meer polyvalent wordt.

Stel dat we volgende klassen hebben waarbij de Boek klasse de **IVerwijderbaar** interface implementeert:

```
1 interface IVerwijderbaar{ ... };
2 internal class Boek: IVerwijderbaar { ... };
3 internal class Persoon { ... };
```

We kunnen nu met **is** objecten bevragen of ze de interface in kwestie hebben:

```
1 Persoon tim = new Persoon();
2 Boek gameOfThrones = new Boek();
3
4 if(gameOfThrones is IVerwijderbaar)
5 {
6     Console.WriteLine("Ik kan Game of Thrones verwijderen");
7 }
8 if(tim is IVerwijderbaar)
9 {
10    Console.WriteLine("Ik kan Tim verwijderen");
11 }
```

De output zal worden: Ik kan Game of Thrones verwijderen.

Net zoals bij onze voorbeelden over polymorfisme en **is** zal de kracht van interfaces pas zichtbaar worden wanneer we met arrays of lijsten van objecten werken. Indien deze lijst een bont allegaartje objecten bevat, allemaal met specifieke parents én interfaces, dan kunnen we weer met **is** bijvoorbeeld alle objecten benaderen die een bepaalde interface hebben:

```
1 foreach(var persoon in WerkNemers)
2 {
3     if(persoon is IManager)
4     {
5         //...
6     }
7 }
```

17.2.1 Meerder interfaces

Een nadeel van overerving is dat een klasse maar van 1 klasse kan overerven. Een klasse mag echter wel meerdere interfaces met zich meedragen:

```
1 interface ISuperHeld{...}
2 interface ICoureur{...}
3 internal class Man {...}
4
5 internal class Zorro:Man, ISuperHeld
6 {...}
7
8 internal class Batman:Man, ISuperHeld, ICoureur
9 {...}
```



Merk op dat de volgorde belangrijk is: eerst plaats je de klasse waarvan wordt overgeerfd, dan pas de interface(s).

Ook mogen interfaces van elkaar overerven:

```
1 interface IGod:ISuperHeld
2 { }
```



In kleine projecten lijken interfaces wat overkill, en dat zijn ze vaak wel. Van zodra je een iets complexer project krijgt met meerdere klassen die onderling met elkaar allerlei zaken moeten doen, dan zijn interfaces je dikke vrienden! Je hebt misschien al over de *SOLID programmeerprincipes* gehoord?

And if not, niet erg.

Samengevat zegt SOLID dat we een bepaalde hoeveelheid abstractie inbouwen enerzijds (zodat we niet de gore details van klassen moeten kennen om er mee te programmeren) anderzijds dat er een zogenaamde ‘separation of concerns’ (SoC) moet zijn (ieder deel/klasse/module van je code heeft een specifieke opdracht).

Met interfaces kunnen we volgens de SOLID principes programmeren: het boeit ons niet meer wat er in de klasse zit, we kunnen gewoon aan de interfaces van een klasse zien wat hij kan doen. Handig toch!

17.3 Interfaces in de praktijk

In het vorige hoofdstuk gaf ik een voorbeeld van een klasse `EersteMinister` die enkele `Minister`-klassen gebruikte om hem of haar te helpen.

Een nadeel van die aanpak is dat al onze Ministers maar 1 “job” kunnen hebben: ze erven allemaal over van `Minister` en kunnen nergens anders van overerven (geen *multiple inheritance* is toeestaan in C#). Je wordt uiteraard niet geboren als Minister. Het zou dus handig zijn dat ook andere mensen Minister kunnen worden, zonder dat ze hun bestaande expertise moeten weggooien.

Via interfaces kunnen we dit oplossen. Een `Minister` gaan we dan eerder als een “bij-job” beschouwen en niet de hoofdreden van een klasse.

We definiëren daarom eerst een nieuwe interface `IMinister`:

```
1 interface IMinister
2 {
3     void Adviseer();
4 }
```

Vanaf nu kan eender *wie* die deze interface implementeert de `EersteMinister` advies geven. Hoera! En daarnaast kan die klasse echter ook nog tal van andere zaken doen. Beeld je in dat een CEO van een bedrijf ook minister bij de `EersteMinister` wilt zijn, zoals deze:

```
1 internal class Ceo
2 {
3     public void MaakJaarlijkseOmzet()
4     {
5         Console.WriteLine("Geld!!!!");
6     }
7     public void OntslaDepartement()
8     {
9         Console.WriteLine("You're all fired!");
10    }
11 }
```

Nu we de interface `IMinister` hebben kunnen we deze klasse aanvullen met deze interface zonder dat de bestaande werking van de klasse moet aangepast worden:

```
1 internal class Ceo: IMinister
2 {
3     public void Adviseer()
4     {
5         Console.WriteLine("Vrijhandel is essentieel!");
6     }
7     //gevolgd door de reeds bestaande methoden
```

De CEO kan dus z'n bestaande job blijven uitoefenen maar ook als Minister optreden.

Ook de EersteMinister moet aangepast worden om nu met een lijst van IMinister ipv Minister te werken. Dankzij polymorfisme is dat erg eenvoudig!

```

1 internal class MisterEersteMinister
2 {
3     public void Regeer()
4     {
5         List<IMinister> AlleMinisters = new List<IMinister>();
6         AlleMinisters.Add(new Ceo);
7         foreach (IMinister minister in AlleMinisters)
8         {
9             minister.Adviseer();
10        }
11    }
12 }
```

De eerder beschreven MinisterVanMilieu, MinisterBZen MinisterVanEconomie dienen ook niet meer van de abstracte klasse Minister over te eren en kunnen gewoon de interface implementeren. Enkel lijn 1 moet hierbij aangepast worden:

```

1 internal class MinisterVanMilieu:IMinister
2 {
3     public void Adviseer()
4     {
5         VerhoogBosSubsidies();
6         OpenOnderzoek();
7         ContacteerGreenpeace();
8     }
9
10    private void VerhoogBosSubsidies(){ ... }
11    private void OpenOnderzoek(){ ... }
12    private void ContacteerGreenpeace(){ ... }
13 }
14 }
```

En bij deze hebben we dankzij interfaces, compositie en polymorfisme, ervoor gezorgd dat eender wie Minister kan worden zonder dat die daarvoor z'n bestaande job moet opzeggen. OOP laat ons echt toe de realiteit zo dicht mogelijk te benaderen!

17.4 Bestaande interfaces in .NET

De bestaande .NET klassen gebruiken vaak interfaces om bepaalde zaken uit te voeren. Zo heeft .NET tal van interfaces gedefinieerd (bv. `IEnumerable`, `IDisposable`, `IList`, `IQueryable` enz.) waar je zelfgemaakte klassen mogelijk aan moeten voldoen indien ze bepaalde bestaande methoden wensen te gebruiken.

Een typisch voorbeeld is het gebruik van de `Array.Sort` methode. Hier wordt het echte nut van interfaces erg duidelijk: de ontwikkelaars van .NET kunnen niet voorspellen hoe andere ontwikkelaars hun bibliotheken gaan gebruiken. Via interfaces geven ze als het ware kijntlijnen en vanaf dan moeten de ontwikkelaars zelf maar bepalen hoe hun nieuwe klassen zullen samenwerken met die van .NET.

17.4.1 Sorteren met `Array.Sort` en de `IComparable` interface

Een veelgebruikte .NET interface is de `IComparable` interface. Deze wordt gebruikt indien .NET bijvoorbeeld een array van objecten wil sorteren. Bij wijze van demonstratie zal ik demonstreren waarom deze interface erg nuttig kan zijn.

17.4.1.1 Stap 1: Het probleem

Indien je een array van objecten hebt en je wenst deze te sorteren via `Array.Sort` dan dienen de objecten de `IComparable` interface te hebben.

We willen een array van landen kunnen sorteren op grootte van oppervlakte.

Stel dat we de klasse Land hebben:

```
1 internal class Land
2 {
3     public string Naam {get;set;}
4     public int Oppervlakte {get;set;}
5     public int Inwoners {get;set;}
6 }
```

We plaatsen 3 landen in een array:

```
1 Land[] eurolanden = new Land[3];
2 eurolanden[0] = new Land() {Naam = "België", Oppervlakte = 5,
3                             Inwoners = 2000};
4 eurolanden[1] = new Land() {Naam = "Frankrijk", Oppervlakte = 7,
5                             Inwoners = 2500};
6 eurolanden[2] = new Land() {Naam = "Nederland", Oppervlakte = 6,
7                             Inwoners = 1800};
```

Wanneer we nu zouden proberen de landen te sorteren:

```
1 Array.Sort(eurolanden);
```

Dan treedt er een uitzondering op: `InvalidOperationException: Failed to compare two elements in the array.` Dit is erg logisch: .NET heeft geen flauw benul hoe objecten van het type `Land` moeten gesorteerd worden. Moet dit alfabetisch volgens de `Naam` property, of van groot naar klein op aantal `Inwoners`? Enkel jij als ontwikkelaar weet momenteel hoe er gesorteerd moet worden.

17.4.1.2 Stap 2: IComparable onderzoeken

We kunnen dit oplossen door de `IComparable` interface in de klasse `Land` te implementeren. We bekijken daarom eerst de documentatie van deze interface². De interface is beschreven als:

```
1 interface IComparable
2 {
3     int CompareTo(Object obj);
4 }
```



OPGELET: Deze interface bestaat al in .NET en mag je dus niet opnieuw in code schrijven!

Daarbij moet de methode een `int` teruggeven als volgt:

Waarde	Betekenis
Getal kleiner dan 0	Huidig object komt voor het obj dat werd meegegeven.
0	Huidig object komt op dezelfde positie als obj.
Getal groter dan 0	Huidig object komt na obj.

De `Array.Sort` methode zal werken tegen deze `IComparable` interface om juist te kunnen sorteren. Het verwacht dat de klasse in kwestie een `int` teruggeeft volgens de afspraken van de tabel hierboven.

²Zie <https://msdn.microsoft.com/system.icomparable>

17.4.1.3 Stap 3: IComparable in Land implementeren

We zorgen er nu voor dat Land deze interface implementeert. Daarbij willen we dat *de landen volgens oppervlakte worden gesorteerd*:

```

1  internal class Land: IComparable
2  {
3      public int CompareTo(object obj)
4      {
5          Land temp = obj as Land;
6          if(temp != null)
7          {
8              if(Oppervlakte > temp.Oppervlakte)
9                  return 1;
10             if(Oppervlakte < temp.Oppervlakte)
11                 return -1;
12             return 0;
13         }
14     else
15         throw new NotImplementedException("Object is not a Land")
16         ;
17     }
18 }
```

Nu zal de Sort werken:

```
1 Array.Sort(eurolanden);
```

De Sort()-methode kan nu ieder object bevragen via de CompareTo()-methode en zo volgens een eigen interne sorteeralgoritme de landen in de juiste volgorde plaatsen.

Stel dat vervolgens nog beter willen sorteren: *we willen dat landen met een gelijke oppervlakte, op hun aantal inwoners gesorteerd worden*:

```

1  public int CompareTo(object obj)
2  {
3
4      Land temp = obj as Land;
5      if(temp != null)
6      {
7          if(Oppervlakte > temp.Oppervlakte) return 1;
8          if(Oppervlakte < temp.Oppervlakte) return -1;
9          if(this.Inwoners > temp.Inwoners) return 1;
10         if(this.Inwoners < temp.Inwoners) return -1;
11     }
12   else
13     throw new ArgumentException("Object is not a Land");
14 }
15 }
```

Ik laat jou de code schrijven wat er moet gebeuren indien het aantal inwoners én de oppervlakte dezelfde is. Misschien kan je dan sorteren volgens de Naam van het land.



De bestaande datatypes in .NET hebben allemaal de `IComparable` interface ingebakken. Zo ook dus de gekende primitive datatypes. `string` dus ook en laat dus toe om bijvoorbeeld snel te weten welke van 2 string alfabetisch eerst komt, als volgt:

```
1 return this.Naam.CompareTo(temp.Naam);
```

Kortom, voeg dit achteraan de eerder geschreven vergelijkingen in je `Land`-klasse om uiteindelijk de `Naam` te gebruiken als sorteerelement.

17.4.2 List sorteren

Indien je een `List<Land>` zou willen sorteren in plaats van een array van `Land` dan kan dit ook. Nog steeds vereisen we dat je klasse de `IComparable` interface gebruikt. We kunnen nu de ingebouwde `Sort`-methode van de `List` klasse gebruiken. Stel dat je een lijst van landen hebt genaamd `landLijst`, deze sorteren kan dan heel eenvoudig als volgt:

```
1 landLijst.Sort();
```

Zo simpel!

Merk op dat we hier de lijst zelf sorteren. Er wordt dus geen nieuwe lijst teruggegeven zoals bij `Array.Sort()` het geval is.

Indien je toch liever `Array.Sort` gebruikt dan kunnen we een andere, handige, ingebouwde `List`-methode gebruiken, namelijk `ToArray()`, als volgt:

```
1 landLijst = Array.Sort(landLijst.ToArray());
```

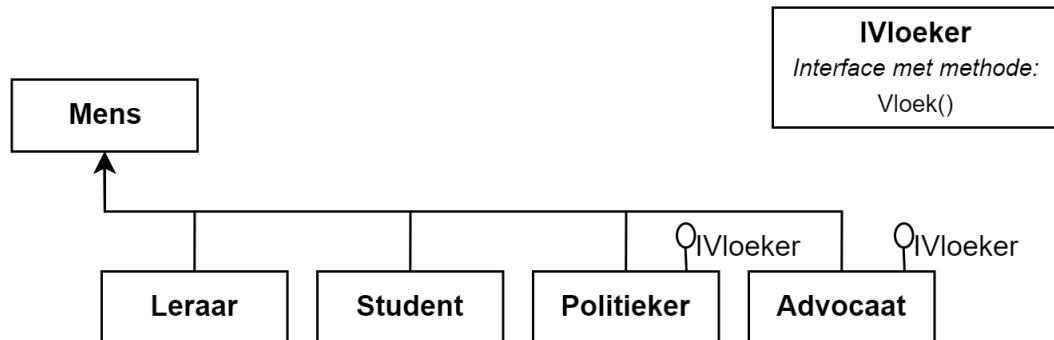
17.5 Alles samen : Polymorfisme, interfaces en is/as

De eigenschappen van polymorfisme en interfaces combineren kan tot zeer krachtige code resulteren. Wanneer we dan ook nog eens de **is** en **as** keywords gebruiken is het hek helemaal van de dam. Als afsluiter van deze lange reis in OOP-land zal ik daarom een voorbeeld geven waarin de verschillende OOP-concepten samenkommen om ... vloekende mensen op het scherm te tonen.

17.5.1 Vloekende mensen: Opstart

Het idee is het volgende: mensen kunnen spreken. Leraars, studenten, politieker, en ja zelfs advocaten zijn mensen. Echter, enkel politiekers en advocaten hebben ook de interface **IVloeker** die hen toelaat eens goed te vloeken. Brave leerkrachten en studenten doen dat niet (*kuch*). We willen een programma dat lijsten van mensen bevat waarbij we de vloekers kunnen doen vloeken zonder complexe code te moeten schrijven.

We hebben volgende klasse-structuur:



Figuur 17.5: Klasse-schema van de vloekende mensen.

Als basis klasse Mens hebben we:

```

1 internal class Mens
2 {
3     public void Spreek()
4     {
5         Console.WriteLine("Hoi!");
6     }
7 }
  
```

Voorts definiëren we de interface **IVloeker** als volgt:

```

1 interface IVloeker
2 {
3     void Vloek();
4 }
  
```

We kunnen nu de nodige child-klassen maken:

1. De niet-vloekers: Leraar en Student
2. De vloekers: Advocaat en Politieker

```
1 internal class Leraar:Mens {} //moet niets speciaal doen
2
3 internal class Student:Mens{} //ook studenten doen niets speciaal
4
5 internal class Politieker: Mens, IVloeker
6 {
7     public void Vloek()
8     {
9         Console.WriteLine("Godvermiljaardedju, zei de politieker");
10    }
11 }
12
13 internal class Advocaat: Mens, IVloeker
14 {
15     public void Vloek()
16     {
17         Console.WriteLine("SHIIIIIT, zei de advocaat");
18     }
19 }
```

17.5.2 Vloekende mensen: Het probleem

We maken een array van mensen aan waarin we van iedere type een vertegenwoordiger plaatsen (uiterraad had dit ook in een `List<Mens>` kunnen gebeuren):

```
1 Mens[] mensjes = new Mens[4];
2 mensjes[0] = new Leraar();
3 mensjes[1] = new Politieker();
4 mensjes[2] = new Student();
5 mensjes[3] = new Advocaat();
6 for(int i = 0; i < mensjes.Length; i++)
7 {
8     //NOW WHAT?
```

Het probleem: hoe kan ik in de array van studenten, leraren, advocaten en politiekers **enkel de vloekers laten vloeken?**

17.5.3 Oplossing 1: **is** to the rescue

De eerste oplossing is door gebruik te maken van het **is** keyword. We zullen de array doorlopen en steeds aan het huidige object vragen of dit object de **IVloeker** interface bezit, als volgt:

```

1 for(int i = 0; i<mensjes.Length; i++)
2 {
3     if(mensjes[i] is IVloeker)
4     {
5         //NOW WHAT?
6     }
7     else
8     {
9         mensjes[i].Spreek();
10    }
11 }
```

Vervolgens kunnen we binnen deze **if** het huidige object tijdelijk omzetten (casten) naar een **IVloeker** object en laten vloeken:

```

1 if(mensjes[i] is IVloeker)
2 {
3     IVloeker tijdelijk = (IVloeker)mensjes[i];
4     tijdelijk.Vloek();
5 }
```

17.5.4 Oplossing 2: **as** to the rescue

Het **as** keyword kan ook een toffe oplossing geven. Hierbij zullen we het object proberen om te zetten via **as** naar een **IVloeker**. Als dit lukt (het object is verschillend van **null**) dan kunnen we het object laten vloeken:

```

1 for(int i = 0; i<mensjes.Length; i++)
2 {
3     IVloeker tijdelijk = mensjes[i] as IVloeker;
4     if(tijdelijk != null)
5     {
6         tijdelijk.Vloek();
7     }
8     else
9     {
10        mensjes[i].Spreek();
11    }
12 }
```



Hopelijk hebben voorgaande voorbeelden je een beetje kunnen doen proeven van de kracht van interfaces. Gedaan met ons druk te maken wat er allemaal in een klasse gebeurt. Werk gewoon ‘tegen’ de interfaces van een klasse en we krijgen de ultieme black-box revelatie! See what I did there?

18 Bestandsverwerking

Op vraag van velen is het nu tijd om één van de meest gebruikte .NET namespaces te bekijken: **System.IO**¹.



Vergeet zeker niet bovenaan je code **using System.IO;** toe te voegen indien je ook maar één voorbeeld uit dit hoofdstuk wilt kunnen maken.

De System.IO namespace is een zeer uitgebreide bibliotheek die alle methoden bevat die je nodig hebt om input en output (I/O) operaties te verrichten. Dit betekent dat je met behulp van deze namespace bestanden en folders (ook wel *mappen* of *directories* genoemd) kunt uitlezen, schrijven, maken en verwijderen, en zo voort.

Het zal je met andere woorden toelaten om toegang te krijgen tot, onder andere, het lokale bestandssysteem.

...met alle voor-en nadelen als gevolg. Je zal namelijk rechtstreeks:

- **Wijzigingen op je harde schijf** kunnen aanbrengen. Het is belangrijk om voorzichtig te zijn wanneer je werkt met bestanden en folders. Eén kleine fout kan leiden tot het verwijderen van belangrijke bestanden of gegevensverlies. En ja, ook ik heb dit al meegemaakt.
- **Niet altijd de juiste gebruikersrechten hebben** om bepaalde I/O bewerkingen uit te voeren. Dit hangt af van de rechten die je gecompileerde programma heeft binnen het besturingssysteem. Zorg ervoor dat je controleert of je voldoende rechten hebt voordat je probeert een bestand te lezen of te schrijven.

¹Zoals steeds, kijk zeker eens naar de officiële documentatie op learn.microsoft.com/en-us/dotnet/api/system.io.



Enkele tips voor je begint

Om de voorgaande waarschuwing te benadrukken, nog 2 belangrijke tips:

1. **Back-ups:** Backups maken is altijd belangrijk. Maar de voorbije 17 hoofdstukken heb je normaal gezien nooit code geschreven die effectief zaken kon kapot maken op je computer. Daar komt dus vanaf nu verandering in. Tijd dus voor die wekelijkse backup!
2. **Try-catch gebruiken:** Eigenlijk hebben we voorlopig maar beperkt aan exception handling moeten doen. 99% van de tijd wisten we heel goed welke uitzonderingen konden optreden en schreven we onze code er naar. Echter, vanaf nu zal je programma ook zaken benaderen *buiten het programma*. Tot aan dit hoofdstuk was enkel de gebruikersinput iets dat van buiten kwam. We waren meestal zelf de eindgebruiker, en gingen uit van foutloze invoer (*ik weet het, naïef*). Bestanden luisteren echter niet zo goed. Het kan dus goed zijn dat een bestand toch niet op die plaats staat waar je dacht dat het stond, of dat je toch niet de juiste rechten hebt. Kortom, **exception handling zal vanaf nu essentieel worden.**

18.1 Bestands- en folderlocaties

Ieder bestand en folder op je harde schijf wordt gedefinieerd door een unieke locatie, **path** genoemd. Als je een bestand genaamd `mijnData.txt` hebt in de “temp”-folder van je c-schijf, dan is het **full path** van dit bestand:

```
1 c:\temp\mijnData.txt
```

Folders hebben ook een path, in het vorige voorbeeld is het full path van de temp-folder:

```
1 c:\temp\
```

In C# zullen deze paths altijd als **string** worden verwerkt.



Let er op dat een path **niet hoofdlettergevoelig** is. Je kan dus geen 2 bestanden met de naam “mijnData.txt” en “MijnData.TXT” in dezelfde folder hebben. Zowel Windows als Mac OS hebben een niet hoofdlettergevoelige bestandsstructuur.



Bij MacOS werkt men met *forward slashes* in plaats van *backward slashes*.

18.1.1 Bestandslocatie zonder path

Als je geen specifiek path aangeeft wanneer je een bestand wilt gebruiken, gaat je programma ervan uit dat het zich in de folder van het programma zelf zich bevindt. Voor de meeste projecten is dit meestal de \bin\Debug-folder tijdens het ontwikkelen en testen.

Volgende voorbeeld (we lopen even al vooruit) zal de tekst “Het einde is nabij” wegschrijven naar een bestand “doem.txt” op één van volgende locaties:

- Lijn 1: Naar de plek waar het programma wordt uitgevoerd.
- Lijn 2: Naar de temp-folder.

```
1 File.WriteAllText("doem.txt", "Het einde is nabij");
2 File.WriteAllText(@"c:\temp\doem.txt", "Het einde is nabij");
```

In het tweede geval is belangrijk te controleren of je wel schrijfrechten hebt voor die folder. Heb je die niet dan zal je een **UnauthorizedAccessException** krijgen. We gebruiken dus best exception handling wanneer we met bestanden werken.

```
1 try
2 {
3     File.WriteAllText("doem.txt", "Het einde is nabij");
4 }
5 catch(UnauthorizedAccessException)
6 {
7     Console.WriteLine($"Geen schrijfrechten!")
8 }
```

18.1.2 System.IO.Path

Het is niet altijd duidelijk of je applicatie op een Mac of Windows zal uitgevoerd worden. Je kan er dus maar beter rekening mee houden dat je applicatie soms met andere paths zal werken dan je gewend bent (Mac OS werkt bijvoorbeeld niet met een “c:”-schijf notatie). Het is daarom veiliger om te werken met de **System.IO.Path**-klasse, die ons in staat stelt om op een platformonafhankelijke manier met paden en bestandsnamen te werken.

Stel dat we een bestandsnaam willen samenstellen uit verschillende delen, dan gebruiken we hier de erg nuttige **Combine-methode** voor:

```
1 string folder = "data";
2 string bestand = "dagboek.txt";
3 string fullPath = Path.Combine(directory, filename);
4 Console.WriteLine(fullPath);
```

Afhankelijk van het besturingssystemen zal de output dus verschillend zijn. Op Windows:

```
1 data\dagboek.txt
```

Op Mac OS wordt dit echter:

```
1 data/dagboek.txt
```

De **Path**-klasse heeft ook nog tal van nuttige methode zoals: **ChangeExtension**, **GetDirectoryName**, **GetFileNameWithoutExtension**, **GetFullPath**, enz. Met

behulp van de `GetRandomFileName`-methode kan je een willekeurige folder- of bestandsnaam verkrijgen. Dit is handig als je een tijdelijke bestand wil aanmaken en zeker wil zijn dat de naam niet al bestaat. Een andere handige methode is `GetTempPath`, deze geeft je het path naar de temp-folder van de huidige gebruiker. Het is een goede gewoonte om tijdelijke werkbestanden voor je applicatie in deze folder te plaatsen. En het is nog toffer als je deze ook verwijderd wanneer ze niet meer nodig zijn.

18.1.3 Speciale folders

Soms wil je bestanden opslaan in speciale folders zoals op het bureaublad. Hiervoor kun je de `Environment.GetFolderPath`-methode gebruiken. Deze methode vereist een parameter van het type `Environment.SpecialFolder`, wat een ingebouwde enum is. Deze enum bevat een hele hoop *gekende* locaties van Windows-folders, zoals het bureaublad, de temp-folder, "Mijn documenten"-folder, enz. Dit zorgt ervoor dat je bestand altijd op de juiste plek komt, ongeacht de gebruikersomgeving.

In dit voorbeeld plaatsen we ons onheilspellende bericht op het bureaublad van de huidige gebruiker:

```
1 string desktopPath = Environment.GetFolderPath(Environment.  
    SpecialFolder.Desktop);  
2 string fullPath = Path.Combine(desktopPath, "doem.txt");  
3 File.WriteAllText(fullPath, "Het einde is nabij");
```

18.1.4 Controleren dat folder of bestand bestaat

Het is een goede gewoonte om steeds te controleren of een bestand of folder al bestaat, voor je ermee gaat werken. Soms wil je een bestaand bestand zeker niet overschrijven. Of wil je een folder aanmaken als deze nog niet bestaat. Controleer dit dus altijd in je programma én vraag aan de gebruiker wat te doen bij twijfel. Overschrijf of verwijder nooit een bestand of folder zonder de gebruiker hierover te waarschuwen. Zeker wanneer het om een bestand gaat dat niet door jouw programma beheerd wordt.

Het controleren of een bestand bestaat doe je met de `File.Exists`-methode:

```
1 if(File.Exists(desktopPath))  
2 {  
3     //werk met bestand  
4 }  
5 else  
6 {  
7     Console.WriteLine("Dit bestand bestaat niet.");  
8 }
```

Het controleren van het bestaan van een folder doe je met `Directory.Exists()`:

```
1 if(Directory.Exists(tempPath))  
2 {  
3     //werk met folder  
4 }
```



Om de voorbeeldcode in dit hoofdstuk behapbaar te houden, zullen we niet telkens overal deze controle doen.

18.1.5 Folder of bestand aanmaken

Na de controle kan je dan beslissen om het bestand of folder aan te maken. Dat kan met de `File.Create` en `Directory.CreateDirectory`-methoden:

```
1 if (!Directory.Exists(tempPath))  
2 {  
3     Directory.CreateDirectory(tempPath);  
4 }
```

en

```
1 if (!File.Exists(desktopPath))  
2 {  
3     File.Create(desktopPath);  
4 }
```

18.2 Schrijven en lezen

De locaties en paths van bestanden kennen is... interessant. Het wordt natuurlijk pas echt nuttig wanneer we de inhoud van bestanden kunnen uitlezen en aanpassen. Daarvoor zijn we hier natuurlijk.

Er is geen vaste manier om dit te doen. Alles hangt af van je specifieke probleem. We zullen daarom enkele veel gebruikte *use-cases* bekijken.

18.2.1 StreamWriter en StreamReader

18.2.1.1 StreamWriter

Met behulp van een `StreamWriter` kan je data naar een bestand (of een andere *streambron*) wegschrijven. Het gebruik ervan is verrassend eenvoudig.

```
1 StreamWriter writer = new StreamWriter("doeeeeem.txt", true);
2 writer.WriteLine("Game over!");
```

Inderdaad, net zoals je bij de `Console`-klasse `WriteLine` hebt, heb je dat ook hier om tekst naar een bestand te schrijven.

Wanneer we een `writer`-object aanmaken, geven we ook een tweede parameter (`true`) mee. Dit geeft aan dat de tekst achter de bestaande inhoud van het bestand moet worden toegevoegd (*to append*). Indien je `false` meegeeft dan zal de inhoud van het bestand verwijderd worden en start je van een nieuw, leeg bestand.

18.2.1.2 StreamReader

En ja, uiterraard is er dus ook een manier om tekst uit een bestand te lezen. De `StreamReader` heeft onder andere een `ReadLine` methode die dit toelaat. De extra moeilijkheid bij het uitlezen van bestanden is dat we goed moeten nakijken of er nog iets uit te lezen valt. Dit kun je enigszins vergelijken met het uitlezen van een array, want ook daar moeten we opletten dat we niet voorbij de grenzen van de array gaan.

Volgende voorbeeld gebruikt een `StreamReader` om ons tekstbestand lijn per lijn op het scherm te tonen:

```
1 StreamReader reader = new StreamReader("doeeeeem.txt");
2 string regel;
3
4 while ((regel = reader.ReadLine()) != null)
5 {
6     Console.WriteLine(line);
7 }
```

De conditie op lijn 4 verdient wat toelichting. We doen hier 2 zaken:

1. We lezen de volgende lijn uit met behulp van ReadLine.
2. Wanneer we aan het einde van het bestand zijn zouden we **null** terugkrijgen uit ReadLine.
Daarom dat we vervolgens hierop controleren. Enkel als we dus effectief nog tekst uitlezen mogen we nogmaals in de loop gaan.

18.2.2 using alternatief

In een ver verleden (hoofdstuk 10) hadden we het al even over **using** en hoe het gebruikt wordt om aan te geven dat de compiler ook een specifieke **namespace** mag gebruiken. Wel, het **using** keyword heeft ook nog een alternatief gebruik in C#.

Het **using** keyword kan ook worden gebruikt om een bepaald stuk code binnen een codeblok te definiëren waarin een object wordt aangemaakt, gebruikt en vervolgens op de juiste manier wordt verwijderd zodra het blok is uitgevoerd.

Dit is vooral handig bij het werken met bestanden. Het zal ervoor zorgen dat het bestand correct wordt vrijgegeven aan het besturingssysteem, zelfs wanneer er fouten optreden. Beeld je in dat je programma crasht wanneer je net naar een bestand schrijft: hierdoor bestaat de kans dat het bestand *gelockt* wordt waardoor andere applicaties niet aan het bestand aan kunnen. Niet handig.

De vorm van een **using**-codeblock ziet er als volgt uit:

```
1 using (resource aanmaken dat netjes moet worden opgeruimd)
2 {
3
4 }
```

Volgend voorbeeld, waarin we een StreamWriter gebruiken, toont de werking ervan:

```
1 using (StreamWriter writer = new StreamWriter("doem.txt"))
2 {
3     writer.WriteLine("Het einde is nabij!");
4     writer.WriteLine("Hou vol!");
5 }
```

Wanneer de accolades van lijn 5 worden bereikt, zal het object dat in lijn 1 werd aangemaakt, proper afgesloten worden. C# zal de *Dispose*-methode² van het *writer*-object aanroepen zodat het bestand terug vrijgegeven wordt.

²De *Dispose* methode wordt aangeroepen om bestanden, folders of andere *resources* explicet vrij te geven wanneer ze niet langer nodig zijn. Dit voorkomt geheugenlekken en verhoogt de efficiëntie van de toepassing. De methode zit ingebakken in veel klassen die werken met zaken zoals bestanden, netwerkverbindingen, etc. Al deze klassen implementeren de *IDisposable* interface en hebben daarom de *Dispose* methode.

18.2.3 Uitgewerkt voorbeeld

Dankzij `StreamReader` en `StreamWriter` hebben we nu reeds een goede greep op werken met bestanden. Laten we eens alles combineren tot een functioneel programma. We maken een dagboek-programma. Telkens het programma opstart zal het de reeds bestaande dagboek-schrijfsels tonen. Vervolgens kan de gebruiker een nieuwe tekst toevoegen. Nadien sluit het programma af. We gebruiken `DateTime.Now` om ieder schrijfsel van een duidelijke tijd en datum te voorzien:

```

1  string dagboekPath = "dagboek.txt";
2  if (!File.Exists(dagboekPath))
3  {
4      File.Create(dagboekPath);
5  }
6
7 // dagboek tonen
8 Console.WriteLine("Dagboek:");
9 using (StreamReader reader = new StreamReader(dagboekPath))
10 {
11     string regel;
12     while ((regel = reader.ReadLine()) != null)
13     {
14         Console.WriteLine(regel); //
15     }
16 }
17
18 //dagboek schrijven
19 Console.WriteLine("\nGeef je volgende dagboekschrijfsel:");
20 string entry = Console.ReadLine();
21
22 using (StreamWriter writer = new StreamWriter(dagboekPath, true))
23 {
24     writer.WriteLine($"{DateTime.Now}");
25     writer.WriteLine(entry);
26 }
```

18.3 Binaire bestanden

Totnogtoe werkten we enkel met tekstbestanden om strings uit te lezen. Uiteraard hoeft dit niet, maar het is wel eenvoudig. Soms wil je echter ook binaire bestanden aanmaken en verwerken. Dat kan met de `BinaryWriter` en `BinaryReader`. Dit is echter iets complexer.

De extra moeilijkheid is het feit dat we nu niet meer beperkt zijn tot het wegschrijven van strings. We kunnen perfect een bestand aanmaken dat een opeenvolging van waardes bevat met allemaal verschillende datatypes. Vooral bij het uitlezen zal dit die extra complexiteit én foutgevoeligheid geven.

18.3.1 BinaryWriter

De `BinaryWriter` is nog relatief eenvoudig. De werking is zeer gelijklopend met de `TextWriter`. Er zijn wel enkele verschillen.

18.3.1.1 Bestand openen met `File.Open`

Om een bestand te openen met een `BinaryWriter` gebruiken we de `File`-klasse. We moeten daarbij ook expliciet aangeven hoe het bestand moet aangemaakt worden (indien het niet bestaat):

```
1 FileStream fs = File.Open("bondData.dat", FileMode.Create);
2 using (BinaryWriter writer = new BinaryWriter(fs))
3 {
4     //...
5 }
```

Het tweede argument bij de `Open`-methode is een enum die verschillende mogelijkheden heeft:

- `Append`: Opent een bestaand bestand en plaatst de schrijfpositie aan het einde van het bestand, zodat nieuwe data aan het einde worden toegevoegd. Als het bestand niet bestaat, wordt er een `FileNotFoundException` gegenereerd.
- `Create`: Maakt een nieuw bestand aan. Als het bestand al bestaat, wordt het overschreven.
- `CreateNew`: Maakt een nieuw bestand aan. Als het bestand al bestaat, wordt er een `IOException` gegenereerd.
- `Open`: Opent een bestaand bestand. Als het bestand niet bestaat, zal er een `FileNotFoundException` worden gegenereerd.
- `OpenOrCreate`: Opent een bestaand bestand als het bestaat. Als het bestand niet bestaat, wordt er een nieuw bestand aangemaakt.
- `Truncate`: Opent een bestaand bestand en snijdt de inhoud af, waardoor het bestand leeg wordt. Als het bestand niet bestaat, wordt er een `FileNotFoundException` gegenereerd.

Toegegeven, het is niet altijd erg intuïtief welke modus je juist nodig zal hebben. Alles hangt af van het specifieke probleem dat je wenst op te lossen.



Figuur 18.1: Dankzij IntelliSense hoeft je gelukkig niet de 6 FileMode mogelijkheden uit het hoofd te leren.

18.3.1.2 Schrijven met Write

Het andere verschil met de `TextWriter` is dat je nu alleen een `Write`-methode gaat gebruiken. Deze methode accepteert elk datatype. En dat is het grote verschil: **met een `BinaryWriter` kunnen we elk datatype wegschrijven!**

In volgende voorbeeld schrijven we 3 verschillende zaken weg, een `string`, een `int` en uiteindelijk een `bool`:

```

1 var fs = File.Open("bondData.dat", FileMode.Create);
2 using (BinaryWriter writer = new BinaryWriter(fs))
3 {
4     writer.Write("Bond");
5     writer.Write(7);
6     writer.Write(true);
7 }
```

Dit genereert een bestand van 10 bytes. Als we dit bestand vervolgens op binair niveau zouden bekijken, zouden we ontdekken dat de grootte van het bestand exact gelijk is aan de individuele groottes van de drie variabelen:

- De string Bond bestaat uit 4 karakters, elk 1 byte groot, plus 1 extra byte om de lengte van de string aan te geven. Dus in totaal 5 bytes.
- De integer vereist 4 bytes.
- De boolean vereist slechts 1 byte.

Stel nu dat we een complexer voorbeeld hebben waarbij we nog meer typen data willen wegschrijven, zoals een `double` en een `char`:

```

1 var fs = File.Open("bondDataAdvanced.dat", FileMode.Create);
2 using (BinaryWriter writer = new BinaryWriter(fs))
3 {
4     writer.Write("Gadget");
5     writer.Write(42);
6     writer.Write(false);
7     writer.Write(3.14159);
8     writer.Write('A');
9 }
```

In dit voorbeeld zou de bestandsgrootte als volgt worden berekend:

- De `string` is 6 karakters lang + 1 byte voor de lengte: 6 bytes + 1 byte = 7 bytes.
- De integer beslaat 4 bytes.
- De boolean beslaat 1 byte.
- De `double` beslaat 8 bytes.
- De `char` beslaat 1 bytes.
- De totale bestandsgrootte zal daarom 21 bytes zijn.

18.3.2 BinaryReader

Om binaire bestanden uit te lezen hebben we een `BinaryReader`-object nodig. Deze klasse heeft aller `ReadX`-methoden, waarbij *X* het datatype aangeeft dat je wilt uitlezen.

Bij het schrijven hoefden we nog niet echt na te denken over de volgorde. Bij het lezen is dit uiteraard cruciaal. Het is essentieel dat we weten in welke volgorde de data in het bestand staat. Als we bijvoorbeeld eerst een `int` en daarna een `char` hebben weggeschreven, dan moeten we bij het uitlezen exact die volgorde aanhouden. Anders zal de binaire informatie uit het bestand op een verkeerde manier worden verwerkt, wat kan leiden tot onjuiste data of zelfs een crash.

De `Read`-methode van de `BinaryReader` heeft varianten voor ieder primitief datatype beschikbaar in C#. Zo zijn er onder andere: `ReadBoolean`, `ReadByte`, `ReadChar`, `ReadDecimal`, `ReadDouble`, `ReadInt16` (voor `short`), `ReadInt32` (voor `int`), `ReadInt64` (voor `long`), `ReadSingle` (voor `float`), `ReadString`, `ReadUInt16` (voor `ushort`), `ReadUInt32` (voor `uint`).

Hierdoor kunnen we specifiek aangeven welk type data we willen inlezen en kan de `BinaryReader` de bytes correct interpreteren.

Als we dus het `bondData.dat` bestand vervolgens willen uitlezen dan moet dit als volgt:

```
1 var fs = File.Open("bondData.dat", FileMode.Open);
2 using (BinaryReader reader = new BinaryReader(fs))
3 {
4     string naam = reader.ReadString();
5     int code = reader.ReadInt32();
6     bool leeftNog = reader.ReadBoolean();
7     Console.WriteLine($"{naam} ({code}). Leeft nog = {leeftNog}");
8 }
```

Merk op dat we deze keer de modus `FileMode.Open` hanteren bij het openen van het bestand.



Test gerust eens wat er zou gebeuren als je een van de `Read`-methode van volgorde zou veranderen. Meestal zal je een uitzondering krijgen omdat de methoden de in te lezen bytes niet begrijpen en kunnen omzetten naar het verwachte datatype. Als we in het voorgaande voorbeeld lijn 4 en 5 zouden omwisselen dan crasht onze applicatie met een `EndOfStreamException`.

18.3.3 Binaire inhoud tonen

De `File`-klasse heeft een handige methode `ReadAllBytes` waarmee je snel de binaire inhoud van een bestand kunt bekijken. Deze methode is nuttig wanneer je de exacte gegevens van een bestand wilt inspecteren.

De `ReadAllBytes` methode zal een array van `byte` teruggeven die we vervolgens kunnen overlopen in een loop. Als handigheidje gebruiken een string formatter `X2` (zie hoofdstuk 4) om de bytes als hexadecimale waarden af te drukken:

```
1 byte[] inhoud = File.ReadAllBytes("bondData.dat");
2
3 foreach (byte b in inhoud)
4 {
5     Console.WriteLine($"{b:X2} ");
6 }
```

Dit geeft volgende output:

```
1 04 42 6F 6E 64 07 00 00 00 01
```



De eerste byte (04) geeft de lengte van de string aan die volgt, 4 dus. De volgende 4 bytes, 42 6F 6E 64 zijn de Unicode waarden voor de letters “b, o, n, d”. Vervolgens hebben we 4 bytes om het getal 7 voor te stellen (07 00 00 00). Finaal hebben we nog de byte-waarde 01 die de `bool` op `true` voorstelt.



Vond je het vreemd dat 7 binair als 07 00 00 00 werd voorgesteld?

Dit komt doordat het getal 7 wordt opgeslagen als een 4-byte **little-endian** getal. In little-endian-notatie wordt de minst significante byte (**least significant byte** of LSB) eerst opgeslagen. Voor het getal 7 betekent dit dat de hexadecimale waarde 07 in de eerste byte komt, gevolgd door drie nullen omdat de overige bytes geen bijdrage leveren aan de waarde van het getal.

Als we daarentegen het getal 1000 willen voorstellen, 3E8 hexadecimaal, die we in little-endian volgorde opslaan als E8 03 00 00. Hier wordt de LSB E8 (de laagste byte) als eerste byte opgeslagen, gevolgd door 03 en daarna twee nullen om de 4-byte structuur te vervullen, aangezien 3E8 in 32-bits binaire vorm wordt opgeslagen.

18.4 De FileInfo klasse

De FileInfoklasse heeft 2 specifieke doelen:

1. Ze laat toe om **meer informatie over bestanden** te verkrijgen, zoals bijvoorbeeld het moment waarop het bestand werd aangemaakt.
2. Anderzijds kan je er eenvoudig **bestanden met kopiëren, verplaatsen en verwijderen**.

We kunnen deze klasse gebruiken door eerst een object te instantiëren, waarbij we aan de constructor het path meegeven naar het te gebruiken bestand:

```
1 FileInfo fileInfo = new FileInfo(pathNaarBestand);
```



Er is uiteraard ook een DirectoryInfo klasse met een soortgelijke werking, die we verderop bespreken.

18.4.1 Meer informatie over een bestand

Nadat het fileInfo object werd aangemaakt krijg je via een hele resem properties toegang tot detail-informatie over het bestand in kwestie, zoals:

Methode	Info	Voorbeeldoutput
Name	Bestandsnaam	“temp.txt”
FullName	Volledige pad van het bestand	c:\temp\temp.txt
Extension	Bestandsextensie	.txt”
Length	Bestandsgrootte in bytes	21
CreationTime	DateTime object met datum en tijd waarop het bestand is aangemaakt	11/06/2024 10:17:21
LastAccessTime	Laatste keer dat het bestand is geopend	idem
LastWriteTime	Laatste keer dat het bestand is gewijzigd	idem
Exists	boolean die aangeeft of het bestand bestaat	true

Het gebruik van deze properties wijst zichzelf uit (uiteindelijk zijn dit allemaal read-only properties):

```

1 FileInfo info = new FileInfo("bondData.dat");
2 if (info.Exists)
3 {
4     Console.WriteLine($"Bestandsnaam: {info.Name}");
5     Console.WriteLine($"Bestandsgrootte: {info.Length}/1024 KB");
6 }
```

18.4.2 Kopiëren, verplaatsen en verwijderen

Van zodra je een `FileInfo`-object hebt, krijg je beschikking over tal van handige methoden. We gaan de 3 nuttigste (`CopyTo`, `MoveTo` en `Delete`) eens tonen in een domme demo:

```

1 FileInfo info = new FileInfo("bondData.dat");
2 if(info.Exists)
3 {
4     fileInfo.CopyTo("supermanData.dat");
5     fileInfo.MoveTo("bond2Data.dat");
6     fileInfo.Delete();
7 }
```

Als we deze code uitvoeren zullen er 3 zaken gebeuren, op voorwaarde dat het bestand `bondData.dat` beschikbaar:

- Lijn 4: Een tweede bestand `supermanData.dat` wordt aangemaakt en zal dezelfde informatie als het originele bestand bevatten.
- Lijn 5: Het bestand `bondData.dat` wordt hernoemd naar `bond2Data.dat`.
- Lijn 6: Het originele bestand wordt verwijderd. Ook al werd het hernoemd.
- Lijn 7: Als we nu de folder zouden bekijken waar de applicatie werd uitgevoerd, dan zouden we enkel nog een bestand met de naam “`supermanData.dat`” zien staan.

18.4.3 File of FileInfo

De `System.IO` namespace is een nogal verwarringende klasse. Je kan dezelfde zaken op verschillende manieren doen. Er zijn verschillende Readers en Writers. Soms gebruik je static-methoden, soms object-methoden. Wat is het nu? Lig er niet te hard van wakker! Je bent nog maar aan het prille begin van je C# carrière en zal de komende jaren zeker beter aanvoelen wanneer je welke oplossingsstrategie moet toepassen.

Toch willen we kort toelichten wat het verschil is tussen `File` en `FileInfo`. Je hebt gezien dat ik ze beide doorheen dit hoofdstuk door elkaar gebruikte. Beide klassen hebben veel gelijkaardige functionaliteiten, maar de `File`-klasse is een **static klasse**. Terwijl `FileInfo` dat niet is.

De `File` vereist dus niet dat je telkens een object aanmaakt wanneer je snel iets met een bestand wenst te doen. Bij `FileInfo` doen we dit uiteraard wel, waarbij we het path naar het te gebruiken bestand meegeven.

En hier ligt dan ook direct een groot verschil: de `FileInfo` heeft bewust een **Refresh-methode**, omdat het niet kan garanderen dat alle ingelezen informatie later in de code nog relevant is. `File` zal steeds *instaan* met het betrokken bestand werken. `FileInfo` doet dit enkel tijdens de constructie en bij een Refresh.

Als je meerdere bewerkingen op een bestand wilt doen na elkaar is `FileInfo` aangeraden, daar je anders meerdere keren de file expliciet zou openen en sluiten met de `File` klasse. Wil je echter maar *kort en krachtig* iets met het bestand doen (bv. controleren of het bestaat met `File.Exists`) dan gebruik je beter de `File` klasse. Maar toegegeven, dit is geen harde wet. Ik zou daarom momenteel aanbevelen: gebruik wat je zelf het prettigst vindt. Van zodra je professionele code moet beginnen schrijven waarbij performantie en veiligheid belangrijk is, dan wordt het tijd om *in-depth* na te denken over het gebruik van specifieke klassen.

18.5 DirectoryInfo klasse

Uiteraard is er ook een `DirectoryInfo` klasse. En net zoals `FileInfo` een tegenhanger in de vorm van de `File` klasse heeft, zo is er ook de `Directory` klasse. Ook hier is `Directory` een **static klasse**, en `DirectoryInfo` niet. De uitleg van zonet over het verschil blijft dus ook hier gelden.

Deze klasse geeft dus meer informatie over een folder en het gebruik is identiek aan de `FileInfo`-klasse. Eerst moet er weer een object van aan gemaakt worden:

```
1 DirectoryInfo dirInfo = new DirectoryInfo(@"c:\temp");
```

Wederom kunnen we de typische properties (`LastAccessTime`, `CreationTime`, enz.) en methoden (`Create`, `Delete`, `MoveTo` enz.) aanroepen.

18.5.1 GetFiles en GetDirectories

De DirectoryInfo-klasse heeft nog 2 erg nuttige methoden om te bekijken welke elementen in de folder staan. GetFiles en GetDirectories geven een array van respectievelijk FileInfo en DirectoryInfo objecten terug. Vervolgens kunnen we deze arrays van paths gebruiken om bijvoorbeeld deze bestanden te verwijderen.

Volgende code toont hoe je kunt visualiseren welke elementen zich in de c:\temp-folder bevinden:

```
1 DirectoryInfo tempInfo = new DirectoryInfo(@"C:\temp");
2
3 if(tempInfo.Exists)
4 {
5     var bestanden = tempInfo.GetFiles();
6     var folders = tempInfo.GetDirectories();
7
8     foreach (var folder in folders)
9     {
10         Console.WriteLine($"Folder:{folder.Name}");
11     }
12     foreach (var bestand in bestanden)
13     {
14         Console.WriteLine(bestand.Name);
15     }
16 }
```

18.5.1.1 Filteren op bestanden

De GetFiles-methode aanvaardt enkele handige parameters die je toelaten om naar specifieke bestanden te zoeken.

Ten eerste kan je een **searchPattern** als **string** meegeven. Hierbij kan je met behulp van de asterisk (*) en het vraagteken angeven bepaalde zaken maar te zoeken.

Volgende voorbeeld zal alle bestanden die extensie “.txt” hebben teruggeven:

```
1 var tekstBestanden= tempInfo.GetFiles("*.txt");
```

En deze zal alle bestanden teruggeven wiens bestandsnaam start met “Tim” en dan nog 1 teken bevat. De extensie maakt niet uit:

```
1 var timBestanden= tempInfo.GetFiles("Tim?.*");
```

18.5.1.2 Zoeken in subfolders

Via een tweede argument bij `GetFiles` kan je ook angeven om niet enkel in de huidige folder te zoeken, maar ook in de subfolders. Volgende voorbeeld zal alle bestanden zoeken die eindigen op “.txt”, inclusief in de subfolders:

```
1 var gevonden = tempInfo.GetFiles("*.txt", SearchOption.AllDirectories);
```

18.5.1.3 Recursief een folderstructuur verwerken

Stel dat je alle subfolders en bestanden in die subfolders wilt oplijsten, inclusief folders in subfolders, en zo voort. Hiervoor bestaat geen ingebouwde .NET methode. Je zal dit dus zelf moeten oplossen, waarbij we een **recursie**-structuur zullen moeten aanmaken. Een recursieve methode roept zichzelf terug aan tot aan een bepaalde voorwaarde wordt voldaan. In dit geval wanneer er geen nieuwe subfolders meer worden gedetecteerd in de huidige folder.

Volgende methode zal telkens in de huidige folder, die je meegeeft als argument, alle bestanden en folders van oplijsten. Het zal vervolgens zichzelf aanroepen met als argument telkens één van de subfolders in de huidige folder:

```
1 static void ToonFoldersEnBestanden(string path)
2 {
3     foreach (string bestand in Directory.GetFiles(path))
4     {
5         Console.WriteLine(bestand);
6     }
7
8     foreach (string folder in Directory.GetDirectories(path))
9     {
10        Console.WriteLine(folder);
11        ToonFoldersEnBestanden(folder);
12    }
13 }
```

De eerste **foreach** lijst de bestanden op. De tweede loop zal de folders tonen en ook zichzelf *recursief aanroepen*.



Een klassieke fout bij recursie is een methode schrijven zonder **stopvoorwaarde**. Gelukkig hebben we dat probleem hier niet: de methode stopt met zichzelf aanroepen als er geen subfolders meer zijn in de huidige folder.

Pas wel op: als je deze methode uitvoert op bijvoorbeeld “c:”, zal er waarschijnlijk heel veel tekst op je scherm verschijnen. Dat kan lang duren.

18.6 Klassen serialiseren

Uiteraard weet je nu genoeg om informatie uit je klassen naar een bestand te schrijven en vice versa. Zowel de `BinaryWriter` en `TextWriter` laten in principe toe om je objectinhoud te bewaren. Bij de `TextWriter` moeten we dan een hoop data dan de hele tijd converteren van en naar **string**, wat totaal niet handig werkt. Bij `BinaryWriter` moeten we dan weer goed uitkijken dat we de data in de juiste volgorde inlezen als dat we ze in de eerste instantie hadden weggeschreven.

Telkens een introductie begint zoals de vorige paragraaf, dan weet je dat er een betere oplossing is. Inderdaad, er zit in C# een ingebakken manier om objecten te **serialiseren** naar een bestand. Het woord serialiseren dekt de lading: we gaan de inhoud van een object in serie, achter elkaar bewaren enwegschrijven. Uiteraard zullen we ook het omgekeerde proces bekijken, namelijk **deserialiseren**.



Waarom wil je objecten kunnen serialiseren naar een bestand? Eenvoudig: het laat je toe om de huidige staat van je programma naar een bestand weg te schrijven en later terug op te halen. Je maakt letterlijk een *savepoint* van je programma en geeft je gebruiker de mogelijkheid om op een later moment vanaf dat punt verder te werken.

18.6.1 JSON

Serialiseren naar een binair bestand resulteert in een zeer compact, maar onleesbaar bestand. Dit type bestand is uiterst efficiënt wat betreft opslag en snelheid bij het inlezen. Echter, het aanpassen van een binair bestand is uiterst complex en het biedt geen garantie dat dit bestand nadien nog correct kan worden gedeserialiseerd naar een object. Bovendien is kennis van het exacte binaire formaat vereist om enige aanpassing te maken, wat handmatig werken vrijwel onmogelijk maakt.

Serialiseren naar een tekstbestand geeft daarentegen een zeer leesbaar en dus makkelijker aanpasbaar bestand. Het biedt eveneens het voordeel dat je het bestand eenvoudig kan openen, wijzigen en opslaan met behulp van een eenvoudige teksteditor. We moeten echter nauwkeurig specificeren welk datatype welke string vertegenwoordigt en structuren consequent aanhouden om misverstanden te vermijden.

Het **JSON**-bestandsformaat (JavaScript Object Notation) combineert het beste van beide werelden. We gaan niet alle details van JSON in dit boek bespreken, daar de essentie ervan zeer eenvoudig is. Een JSON-bestand is ogenblikkelijk herkenbaar en leesbaar:

```

1  {
2      "student": {
3          "naam": "barry",
4          "leeftijd": 25,
5          "uitgeschreven": true
6      }
7  }
```



JSON is de spirituele opvolger van XML. Alhoewel dit bestandsformaat nog steeds populair is, zien we toch dat meer en meer applicaties met JSON beginnen werken. XML-bestanden zijn door de grote hoeveelheid tags net iets minder leesbaar dan JSON-bestanden. Zeg nu zelf:

```

1 <student>
2     <naam>Barry</naam>
3     <leeftijd>25</leeftijd>
4     <uitgeschreven>true</uitgeschreven>
5 </student>
```

Met JSON kun je complexe datastructuren representeren zoals arrays en geneste objecten (denk maar aan associaties). Bovendien maakt JSON gebruik van een sleutel-waarde-notatie, wat bijdraagt aan de leesbaarheid. Hier is een meer geavanceerd voorbeeld waarbij we **vierkante haken gebruiken om een array van data te beschrijven**:

```

1  {
2      "school": {
3          "naam": "AP Hogeschool",
4          "locatie": "Antwerpen",
5          "studenten": [
6              {
7                  "naam": "Barry",
8                  "leeftijd": 25,
9                  "uitgeschreven": true
10             },
11             {
12                 "naam": "Anna",
13                 "leeftijd": 22,
14                 "uitgeschreven": false
15             }
16         ]
17     }
18 }
```

18.6.2 Serialiseren in C# naar JSON

Om klassen in C# te serialiseren naar JSON-bestanden, kun je gebruik maken van de `System.Text.Json` namespace.

Objecten serialiseren is verrassend eenvoudig en intuïtief.

1. **Voeg de benodigde namespace toe:** Zonder deze 3 namespaces kan je uiteraard niets doen in deze sectie:

```
1 using System.Text.Json;
2 using System.Text.Json.Serialization;
3 using System.IO;
```

2. **Definieer je klasse:** In principe kan je eender welke klasse serialiseren. Oefen echter eerst met kleine, niet complexe klassen. Probeer zeker eerst associaties te vermijden. Dit is trouwens de eerste keer dat je zal ontdekken waarom properties zo belangrijk zijn: enkel de publieke *zijde* van een object wordt geserialiseerd. Wil je dus private instantievariabelen ook bewaren dan zal je deze via een property beschikbaar moeten maken. Zorg er ook voor dat je klasse **public**:

```
1 public class Student
2 {
3     public string Naam { get; set; }
4     public int Leeftijd { get; set; }
5     public bool Uitgeschreven { get; set; }
6 }
```

3. **Serialiseer de klasse:** Met behulp van de **static** klasse **JsonSerializer** kunnen we nu eenvoudig een object omzetten naar zijn JSON-voorstelling, van het type '**string**'. Je roept gewoon de `Serialize` methode aan en geeft het te serialiseren object mee als argument:

```
1 var student = new Student
2         { Naam = "Barry", Leeftijd = 25, Uitgeschreven = true };
3 string jsonString = JsonSerializer.Serialize(student);
4 Console.WriteLine(jsonString); //ter controle
```

4. **Schrijf naar een bestand:** Finaal kunnen we onze bestaande kennis van de `File.WriteAllText`-methode gebruiken om de JSON-voorstelling naar een bestand weg te schrijven. Merk op dat een goede gewoonte is om het bestand een ".json"-extensie te geven.

```
1 File.WriteAllText("studentdata.json", jsonString);
```

Als we het bestand in een teksteditor zouden openen dan zouden we volgende **string** zien:

```
1 {"Naam": "Barry", "Leeftijd": 25, "Uitgeschreven": true}
```



Je zal in veel documentatie en online bronnen vaak zien dat men een andere namespace gebruikt om met JSON-bestanden te werken in C#. Tot recent was de Newtonsoft.Json namespace de geijkte manier. Deze bibliotheek is door een externe firma, Newtonsoft, ontwikkelt (merk op dat het volledig opensource is!) De .NET ontwikkelaars hebben echter veel tijd en moeite in hun System.Text.Json namespace gestoken, waardoor er nu een ingebouwde .NET oplossing is. Hierdoor is het aangeraden om nu te werken met de Microsoft oplossing, deze kan quasi alles wat de Newtonsoft-oplossing kan en het zal niet lang meer duren voor het meer zal kunnen.

18.6.3 Deserialiseren in C# naar JSON

De omgekeerde weg: deserialiseren.

Om data uit een JSON-bestand te laden, gebruiken we de Deserialize-methode van de JsonSerializer. We veronderstellen dat de klasse onveranderd is gebleven en dat we nog steeds de nodige namespaces voorzien. De methode is *generic*, dus we moeten meegeven welk datatype we verwachten. Dit is logisch: de methode krijgt een **string** en kan niet raden bij welke klasse deze data hoort. voor hetzelfde geld zijn er meerdere klassen met de naam Student en de properties Naam, Leeftijd en Uitgeschreven:

```
1 string jsonText = File.ReadAllText("studentdata.json");
2 Student ingeladen = JsonSerializer.Deserialize<Student>(jsonText);
```

18.6.4 Serialisatiegedrag bijsturen

Soms zijn er zaken in een klasse die niet direct als JSON kunnen worden geserialiseerd. Denk aan private instantievariabelen en read-only properties. Deze zaken zijn niet beschikbaar voor de buitenwereld. In dergelijke gevallen kunnen we het serialisatiegedrag aanpassen door attributen te gebruiken.



Attributen zijn kleine codeblokjes die worden toegevoegd aan onze klasse om bepaalde eigenschappen van de klasse aan te passen. Bijvoorbeeld, we kunnen een attribuut gebruiken om een property te laten overslaan bij het serialiseren. Attributen zijn herkenbaar aan de tekst tussen vierkante haken boven een klasse-element. Merk op dat attributen niets met arrays te maken hebben. Ze zijn een C# manier om je code als het ware meta-informatie te geven die door de compiler of andere bibliotheken kan gebruikt worden.

18.6.4.1 JsonIgnore

Er zijn tal van JSON-gerelateerde attributen beschikbaar om dus het *serialisatiegedrag* bij te sturen. Stel dat we in voorgaande klasse een property hebben die niet mag geserialiseerd worden, dan plaatsen we het `JsonIgnore` attribuut boven die property:

```
1 public class Student
2 {
3     public string Naam { get; set; }
4     public int Leeftijd { get; set; }
5     [JsonIgnore]
6     public bool Uitgeschreven { get; set; }
7 }
```

Als we nu een `Student`-object zouden serialiseren zoals voorheen dan krijgen we volgende JSON:

```
1 {"Naam": "Barry", "Leeftijd": 25}
```

Ook bij het deserialiseren zou `Uitgeschreven` genegeerd worden, zelfs als die in het JSON-bestand zou voorkomen.

18.6.4.2 JsonPropertyName

Dit attribuut laat ons toe om de naam van een property aan te passen wanneer deze wordt geserialiseerd. Dit kan handig zijn als de naam van de property niet exact overeenkomt met de naam die we in het JSON-bestand willen gebruiken:

```
1 public class Student
2 {
3     [JsonPropertyName("VolledigeNaam")]
4     public string Naam { get; set; }
5 }
```

Als we hier een object zouden van serialiseren zou dit volgende JSON geven:

```
1 {
2     "VolledigeNaam": "Barry"
3 }
```

18.6.4.3 JsonInclude

Soms is het belangrijk dat bepaalde private informatie ook geserialiseerd wordt. Met het `JsonInclude` kan je dat aanduiden:

```
1 public class Student
2 {
3     public string Naam { get; set; }
4     [JsonInclude]
5     private int Leeftijd=20;
6 }
```

Beeld je in dat we nog een methode hebben die de leeftijd geregeld zal veranderen, dan nog zal de juiste waarde van `Leeftijd` bewaard worden (en dus niet op 20 blijven). De JSON zal er als volgt uit zien:

```
1 {"Naam": "Barry", "Leeftijd": 20}
```

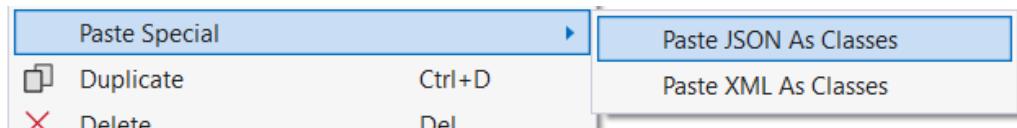


Er zijn nog tal van attributen om het serialisatiegedrag te verbeteren. Deze zijn vaak echter een stuk complexer en worden daarom niet in dit basis handboek behandeld. **Opgelet:** controleer steeds goed of je de juiste attributen hebt opgezocht. Je zal op het internet zowel `Newtonsoft.Json` als `System.Text.Json` attributen vinden, en beide zijn vaak n^et niet hetzelfde.

18.6.5 Onbekende JSON deserialiseren

Als afsluiter toon ik graag een *verborgen feature* van VS die mij al veel tijd heeft bespaard. Stel dat je een stuk JSON hebt van elders³ dat je in je code wilt kunnen deserialiseren naar een object. Het JSON-bestand is echter vrij complex en gebruikt bijvoorbeeld allerlei geneste objecten (door associatie) en arrays, etc. Kortom, hier manueel de juiste klasse(n) voor schrijven voor je verder kan is veel werk. Zoals je al vermoedde kan je dit heel eenvoudig oplossen.

- Stap 1: kopieer de JSON-teks naar het klembord.
- Stap 2: voeg een nieuw, leeg klasse-bestand toe aan je project. verwijder alles in dit bestand behalve de namespace-definitie en bijhorende accolades.
- Stap 3: en nu de magie! Kies in het menu bovenaan voor Edit, dan “Paste Special” en final voor “Paste JSON as Classes”. BOEM!



Figuur 18.2: Deze feature krijgt van mij de prijs voor “Meest onderschatte eigenschap van VS”

Visual Studio heeft nu voor je de nodige klassen geschreven die exact overeen komen met de JSON die jij wilt kunnen deserialiseren. Handig toch?!

³Een typische use-case is wanneer je met een online webapi *praat* die met JSON antwoordt.

19 Conclusie

Je hebt het gehaald! Volgens m'n statistieken zal je nu in 1 van volgende 2 staten zijn:

1. Het scheelt niet veel of je droomt in klassen en objecten. Overal waar je kijkt zie je toepassingen van polymorfisme, interfaces en overerving. Je begrijpt nu waarom zoveel mensen graag software ontwikkelen. Je hebt de smaak te pakken en er ligt een ongelooflijk scala aan mogelijkheden voor je klaar. Bekijk zeker enkele aanbevelingen op de volgende pagina die je na dit boek kan ontdekken. Ook in de appendix zal je nog enkele interessante, gevorderde concepten kunnen ontdekken.
2. Je pakt een traantje weg. Je had zo gehoopt nu alles van OOP te kunnen, maar het is alleen maar verwarrender geworden. Dat is jammer, maar niets aan te doen. Bij sommigen komt de klik niet altijd direct. Hopelijk heb je toch iets geleerd uit dit boek en begrijp je waarom zoveel mensen, zoals ik, zo enthousiast over OOP zijn. Blijven oefenen is de boodschap!

En moet je dit boek nu ongelooflijk nuttig, slecht of briljant vinden: iedere review helpt. Je doet me er een ongelooflijke dienst mee als je een review plaatst op de website waar je dit boek kocht!

Ik wens je alvast veel succes met de verdere ontwikkeling van je programmeer-expertise en denk er aan: gebruik nooit **goto**!

Tim Dams

Zomer 2024



19.1 En nu? Ken ik nu alles van C#/NET ?

Helaas niet. Maar je hebt wel een erg goede basis gelegd. Vanaf dit punt kan je tal van richtingen uitgaan, afhankelijk van je interesses:

- **Geavanceerde C# concepten:** je zou je verder kunnen verdiepen in “de taal C#”. Denk maar aan leren werken met **async** en **events**. Maar ook het wonderlijke **Linq** is iets dat je in bijna alle .NET geledingen zal kunnen gebruiken.
- **Desktop-applicaties:** Totnogtoe hebben we enkel oersaaie Console-applicaties gemaakt. Uiteraard kan je ook heel eenvoudig -met de kennis die je nu hebt- zogenaamde bureaublad-applicaties maken. Neem zeker eens een kijkje wat **WPF** en **UWP** je te bieden heeft. Je zal je even moeten inwerken in **eventgebaseerd**-programmeren en **XAML** en vanaf dan ben je vlot vertrokken!
- **Mobiele applicaties:** Zogenaamde *native* Android of iPhone applicaties ontwikkelen gaat niet met C#. Merk wel op dat dankzij je nieuwe C# kennis je vlot de native programmeertalen van Android (*Java*) en iOS (*Swift*) kan leren. Binnen de .NET-familie bestaat er echter wel het nieuwe **.NET MAUI**-framework. Dit krachtige framework (de opvolger van Xamarin) laat je toe om in C# crossplatform-apps te ontwikkelen. Je zal met 1 *codebase* kunnen compileren naar zowel Windows, Android, iPhone, enz. Bekijk zeker ook eens de Comet toolkit om erg modern-ogende apps te maken met .NET MAUI.
- **Web-ontwikkeling:** Ook .NET heeft een zogenaamde back-end stack waar aardig wat grote bedrijven op draaien. Deze technologie-stack bevat tal van belangrijke technologieën zoals **APS.NET**, **Entity Framework**. En als je genoeg hebt van altijd maar in Javascript te werken, dan moet je zeker eens een kijkje nemen in de jongste .NET-telg **Blazor**, die je toelaat om C# te schrijven in je HTML!
- **Game development:** Wil je eerder de Sid Meiers, John Romeros en Gabe Newells van deze wereld achteraan gaan en games beginnen ontwikkelen? Steeds meer games - zeker in de *indie-wereld* - worden nu ontwikkeld in **Unity**, een op C# gebaseerde game-engine. Maar bekijk zeker ook eens **Monogame**, een C# bibliotheek waar onder andere Stardew Valley in is ontwikkeld. Monogame is een zogenaamde crossplatform bibliotheek en kan je games compileren naar Mac, Windows, Linux, Android, Nintendo Switch, Playstation 4, XboxOne, enz. **Godot** is een andere, laagdrempelige, manier om in C# games te ontwikkelen.
- **Azure en de cloud:** en wil je echt ontdekken dat je nog niet veel kent van .NET, dan moet je eens kijken naar wat er allemaal onder de **Azure**-tak van Microsoft te vinden is. Azure is de verzamelnaam voor alle cloud-gebaseerde technologieën & services van Microsoft, waarin .NET - en dus ook C# - een belangrijk onderdeel is.
- **Gevorderde programmeerconcepten:** **Design Patterns**, Dependency Injection, **SOLID** programming, enz. zijn allemaal taal-agnostische programmeerconcepten. Wat wil zeggen dat je ze kan toepassen op je programmeerproblemen, onafhankelijk van de programmeertaal die je hanteert. Je zal namelijk ontdekken dat bepaalde problemen vaak herleid kunnen worden tot een specifieke groep van problemen. Voor deze problemen hebben slimmere mensen dan ik “oplossings-recepten” (design patterns) uitgedokterd.

Appendix: Handig om weten

1 Visual Studio snippets

Bepaalde code zal je vaak opnieuw schrijven. Er zitten in VS tal van shortcuts om deze typische lijnen code sneller te schrijven. Schrijf een van volgende stukken code en druk dan 2x op de [tab]-toets:

- cw : schrijft Console.WriteLine();
- **for**
- **foreach**
- **while**
- dowhile
- **switch**
- **//**: automatisch methode commentaar blok
- propfull: full property
- prop: auto-property
- ctor: constructor
- **try**: geeft een try-catch blok

2 Regions

Je kan delen van je code in handige inklapbare secties zetten door deze als regions aan te duiden, als volgt:

```
1 #region My Epic code
2 Console.WriteLine("I am the greatest!");
3 Console.WriteLine("Echt waar!");
4 #endregion
```

Je zal vanaf dan in Visual Studio rechts van de start van de region een minnetje zien waar je op kunt klikken om de hele region tot aan #endregion in te klappen. De code zal nog steeds gecompileerd worden, maar je bladspiegel is weer wat ordelijker geworden.

3 String.Format()

String interpolatie met het \$-teken is een nieuwe C# aanwinst. Je zal echter geregeld documentatie en online code tegenkomen die nog met `String.Format` werkt (ook zijn er nog zaken waar het te verkiezen is om `String.Format` te gebruiken i.p.v. 1 van vorige manieren). Om die reden bespreken we dit nog in dit boek.

`String.Format` is een ingebouwde methode die string-interpolatie toelaat op een iets minder intuïtieve manier, als volgt:

```
1 string result = String.Format("Ik ben {0} en ik ben {1} jaar.", naam,  
leeftijd);
```

Het getal tussen de accolades geeft aan welke parameter op die plek moet komen. 0 betekent de eerste, 1 betekent de tweede, enzovoort. De eerste parameter is naam, de tweede is leeftijd.

Volgende code zal een ander resultaat geven:

```
1 string result = String.Format("Ik ben {1} en ben {1} jaar.", naam,  
leeftijd);
```

Namelijk: Ik ben 13 en ik ben 13 jaar oud.



Je kan deze vorm van formateren ook toepassen in `Console.WriteLine` zonder dat je expliciet `String.Format` hiervoor moet aanroepen:

```
1 Console.WriteLine("Gratis formateren. {0} maal hoera voor  
.NET!", 3);
```

4 out en ref keywords

Parameters kun je op twee manieren aan een methode doorgeven: by value (via de waarde) of by reference (via het geheugenadres). Welke manier gebruikt wordt, hangt af van het datatype. Primitieve datatypes zoals int en double worden by value doorgegeven. Arrays worden by reference doorgegeven.

Je kunt primitieve datatypes ook by reference doorgeven. Dit geeft de methode directe toegang tot de variabele, in plaats van een kopie. Dit kan handig zijn, maar kan ook ongewenste bugs veroorzaken. Wees dus voorzichtig.

4.1 Parameters by reference doorgeven

Je kan parameters op 2 manieren by reference doorgeven aan een methode:

- Indien de actuele parameter **reeds een waarde** heeft dan kan je het **ref** keyword gebruiken. Dit gebruik je dus voor in/out-parameters.
- Indien de actuele parameter **pas in de methode een waarde** krijgt toegekend dan wordt het **out** keyword gebruikt. Dit gebruik je dus voor out-parameters.

4.2 ref

Je plaatst het **ref** keyword in de methode signatuur voor de formele parameter dat *by reference* moet meegegeven worden. Vanaf dan heeft de methode toegang tot de originele parameter en dus niet tot de kopie. Je dient ook expliciet het keyword voor de actuele parameter bij de aanroep van de methode te plaatsen:

```

1 static void VerhoogWaarde(ref int getal)
2 {
3     getal++;
4 }
5
6 static void Main(string[] args)
7 {
8     int eerste = 1;
9     Console.WriteLine(eerste); //er verschijnt 1 op het scherm
10    VerhoogWaarde(ref eerste); //let op het ref keyword!
11    Console.WriteLine(eerste); //er verschijnt 2 op het scherm
12 }
```

4.3 out

Door het **out** keyword te gebruiken geven we expliciet aan dat we beseffen dat de parameter in kwestie pas binnen de methode een waarde zal toegekend krijgen. Wat ik hier toon:

```
1 static void GeefWaarde(out int getal)
2 {
3     getal = 5;
4 }
5
6 static void Main(string[] args)
7 {
8     int eerste;
9     GeefWaarde(out eerste);
10    Console.WriteLine(eerste); //er verschijnt 5 op het scherm
11 }
```

5 Foute invoer opvangen met TryParse

Vaak wil je de invoer van de gebruiker verwerken/omzetten naar een getal. Denk maar aan volgende applicatie:

```
1 Console.WriteLine("Geef je leeftijd");
2 string invoer = Console.ReadLine();
3 int leeftijd = int.Parse(invoer);
4 leeftijd += 10;
5 Console.WriteLine($"Over 10 jaar ben je {leeftijd} jaar oud");
```

Deze applicatie zal falen indien de gebruiker iets invoert dat niet kan geconverteerd worden naar een **int**. We lossen dit op met behulp van TryParse.

5.1 Werking TryParse

De primitieve datatypes **int**, **double**, **float** enz. hebben allemaal een TryParse methode. Je kan deze gebruiken om de invoer van een gebruiker **te proberen om te zetten**, als deze niet lukt dan kan je dit ook weten zonder dat je programma crasht door een exception op te werpen.

De werking van TryParse is als volgt:

```
1 bool gelukt = int.TryParse(invoer,out int leeftijd);
```

De methode TryParse probeert de string in de eerste parameter (invoer) om te zetten naar een int. Als dit lukt, wordt het resultaat opgeslagen in de variabele int leeftijd. Let op dat we out voor de parameter moeten zetten, zoals eerder uitgelegd.

Het return resultaat van de methode is **bool**: indien de conversie gelukt is dan zal deze **true** teruggeven, anders **false**.

We kunnen nu onze applicatie herschrijven en minder foutgevoelig maken voor slechte invoer van de gebruiker:

```
1 Console.WriteLine("Geef je leeftijd");
2 string invoer = Console.ReadLine();
3 bool gelukt = int.TryParse(invoer, out int leeftijd);
4 if (gelukt)
5 {
6     leeftijd += 10;
7     Console.WriteLine($"Over 10 jaar ben je {leeftijd} jaar oud");
8 }
9 else
10 {
11     Console.WriteLine("Geen geldige invoer gegeven!");
12 }
```

5.2 TryParse en loops

Daar TryParse een **bool** teruggeeft kunnen we deze ook gebruiken in loops als logische expressie. Volgende applicatie zal aan de gebruiker een komma getal vragen en pas verder gaan indien de gebruiker een geldige invoer heeft gegeven:

```
1 double temperatuur;
2 string invoer = "";
3 do
4 {
5     Console.WriteLine("Geef temperatuur");
6     invoer = Console.ReadLine();
7 } while (! double.TryParse(invoer, out temperatuur));
8
9 //enkel verdergaan van zodra temperatuur een geldige waarde heeft
//gekregen
```



Let er op dat de scope hier van belang is: **invoer** en **temperatuur** moet gekend zijn buiten de loop waar technisch gezien ook de TryParse zal gebeuren.

6 Operator overloading

Stel, je hebt volgende klasse:

```
1 internal class Kassa
2 {
3     public int Totaal {get;set;}
4     public int Bouwjaar {get;set;}
5 }
```

Je maakt even later twee kassa's aan met de nodige informatie:

```
1 Kassa benedenKassa = new Kassa(){Totaal = 50, Bouwjaar = 1981};
2 Kassa bovenKassa = new Kassa(){Totaal = 40, Bouwjaar = 2000};
```

Even later wordt besloten dat beide kassa's moeten samengevoegd worden tot een gloednieuwe kassa voor beide verdiepingen samen. Bedoeling is dat het totale geld in beide kassa's opgeteld in de nieuwe kassa moet gezet worden. Het bouwjaar van de nieuwe kassa moet het bouwjaar van de oudste van de 2 originele kassa's zijn. Je zou willen schrijven:

```
1 Kassa nieuw = benedenKassa + bovenKassa;
```

Uiteraard heeft C# geen flauw benul hoe de **+ operator** moet toegepast worden op objecten van klassen die je zelf geschreven hebt.

6.1 Operator overloading to the rescue

Je kan in een klasse bestaande operators (+,-,*, enz.) **overloaden**. Dit betekent dat je aan C# vertelt hoe een operator moet werken voor objecten van die klasse.

Stel dat je de + wilt overladen in je klasse dan voeg je volgende methode toe:

```
1 internal class Kassa
2 {
3     public int Totaal {get;set;}
4     public int Bouwjaar {get;set;}
5
6     public static Kassa operator+ (Kassa a, Kassa b)
7     {
8         //Zie verder
9     }
10 }
```

Laten we deze syntax even bekijken:

- Operator overloading methoden zijn altijd **static**.
- Het returntype is idealiter het type van de klasse zelf (logisch: twee kassa's optellen geeft een nieuwe kassa).

- **operator**+ geeft aan welke operator je wenst te overladen. Zie verderop met een link naar alle operators die je kan overladen.
- Als je een operator hebt die twee operanden gebruikt (zoals de +), dan heeft de methode ook twee parameters nodig van hetzelfde type als de klasse. Dit zijn de twee elementen (operanden) die je wilt optellen met de operator.

Bekijk zeker eens de officiële documentatie¹ om te zien welke operators je allemaal kan overladen.
Tip: het zijn er veel!

6.2 De operator beschrijven

Vervolgens moeten we nu beschrijven hoe de operator moet werken. Finaal zal de methode een nieuw object moeten teruggeven waarin het resultaat van de operatie zit.

In het voorbeeld dat we maken, willen we dus het volgende:

```
1 public static Kassa operator+ (Kassa a, Kassa b)
2 {
3     Kassa resultaat = new Kassa()
4     {
5         Totaal = a.Totaal+b.Totaal,
6         Bouwjaar = a.Bouwjaar
7     };
8
9     if(a.Bouwjaar < b.Bouwjaar)
10    {
11        resultaat.Bouwjaar = b.Bouwjaar;
12    }
13    return resultaat;
14 }
```

Zoals je ziet maken we een nieuw object **resultaat** waarin we de som van de twee meegegeven kassa's hun totalen plaatsen, alsook het bouwjaar van de oudste van de 2 kassa's.

¹Zie docs.microsoft.com/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators

7 Expression bodied members

Wanneer je methoden, constructors of properties schrijft waar **exact 1 expressie** (*1 lijn code die een resultaat teruggeeft*) nodig is dan kan je gebruik maken van de **expression bodied member syntax** (EBM).

Deze is van de vorm:

```
1 member => expression
```

Dankzij EBM kan je veel kortere code schrijven.

Ik toon telkens een voorbeeld hoe deze origineel is en hoe deze naar EBM syntax kan omgezet worden.

7.1 Methoden en EBM

Origineel:

```
1 public void ToonGeboortejaar(int geboortejaarIn)
2 {
3     Console.WriteLine(geboortejaarIn);
4 }
```

Met EBM:

```
1 public void ToonGeboortejaar(int geboortejaarIn)
2 => Console.WriteLine(geboortejaarIn);
```

Nog een voorbeeld, nu met een return. Merk op dat we return niet moeten schrijven:

```
1 public int GeefGewicht()
2 {
3     return 4 * 34;
4 }
```

Met EBM:

```
1 public int GeefGewicht() => 4 * 34;
```

7.2 Constructors en EBM

Ook constructors die maar 1 expressie bevatten kunnen korter nu. Origineel:

```

1 internal class Student
2 {
3     public int Geboortejaar {get;set;}
4     public Student(int geboorteJaarIn)
5     {
6         Geboortejaar = geboorteJaarIn;
7     }
8 }
```

Met EBM wordt dit:

```

1 internal class Student
2 {
3     public int Geboortejaar {get;set;}
4     public Student(int geboorteJaarIn) => Geboortejaar =
      geboorteJaarIn;
5 }
```

7.3 Full Properties met EBM

Properties worden een soort mengeling tussen full en auto-properties:

```

1 private int name;
2 public int Name
3 {
4     get => name;
5     set => name = value;
6 }
```

7.4 Read-only properties met EBM

Bij read-only properties hoeft het **get** keyword zelfs niet meer getypt te worden bij EBM:

```

1 private int name;
2 public int Name => name;
```



Uiteraard had voorgaande zelfs nog koper geweest met behulp van een auto-property.

8 Generics

8.1 Generieke methoden

Vaak schrijf je methoden die hetzelfde doen, maar waarvan enkel het type van de parameters en/of het returntype verschilt. Stel dat je een methode hebt die de elementen in een array onder elkaar toont. Je wil dit werkende hebben voor arrays van het type **int**, **string**, enz. Zonder generics moeten we dan per type een methode moeten schrijven:

```

1 public static void ToonArray(int[] array)
2 {
3     foreach (var i in array)
4     {
5         Console.WriteLine(i);
6     }
7 }
8
9 public static void ToonArray(string[] array)
10 {
11     foreach (var i in array)
12     {
13         Console.WriteLine(i);
14     }
15 }
```

Dankzij generics kunnen we nu het deel dat **generiek** moet zijn aanduiden (in dit geval met T) en onze methode eenmalig definiëren. We gebruiken hierbij de < > aanduiding die aan de compiler vertelt “dit stuk is een generiek type”:

```

1 public static void ToonArray<T>(T[] array)
2 {
3     foreach (T item in array)
4     {
5         Console.WriteLine(item);
6     }
7 }
```

Vanaf nu kun je eender welk soort array aan deze ene methode geven en de array zal naar het scherm afgedrukt worden:

```

1 int[] getallen= {1,2,4};
2 string[] namen = {"tim", "ali", "marie", "fons"};
3 ToonArray(getallen);
4 ToonArray(namen);
```

8.2 Generic types

We kunnen niet alleen generieke methoden schrijven, maar ook eigen klassen én interfaces definiëren die generiek zijn. In het volgende codevoorbeeld is te zien hoe een eigen generic class in C# gedefinieerd en gebruikt kan worden. Merk het gebruik van de aanduiding T, deze geeft weer aan dat hier een type (zoals **int**, **double**, Student, enz.) zal worden ingevuld tijdens het compileren.

De typeparameter <T> wordt pas voor de specifieke instantie van de generieke klasse of type ingevuld bij het compileren. Hierdoor kan de compiler per instantie controleren of alle parameters en variabelen die in samenhang met het generieke type gebruikt worden wel kloppen.

De afspraak is om .NET een T te gebruiken indien het type nog dient bepaald te worden. Dit is niet verplicht maar wordt aanbevolen als je maar 1 generiek type nodig hebt.

We wensen nu een klasse te maken die de locatie in X,Y,Z coördinaten kan bewaren. We willen echter zowel **float**, **double** als **int** gebruiken om deze X,Y,Z coördinaten in bij te houden:

```
1 internal class Locatie<T>
2 {
3     public T X {get;set;}
4     public T Y {get;set;}
5     public T Z {get;set;}
6 }
```

We kunnen deze klasse nu als volgt gebruiken:

```
1 var plaats = new Locatie<int>();
2 plaats.X = 34;
3 plaats.Y = 22;
4 plaats.Z = 56;
5
6 var plaats2 = new Locatie<double>();
7 plaats2.X = 34.5;
8 plaats2.Y = 22.2;
9 plaats2.Z = 56.7;
10
11 var plaats3 = new Locatie<string>();
12 plaats3.X = "naast de kerk";
13 plaats3.Y = "links van de bakker";
14 plaats3.Z = "onder het hotel";
```



Merk op dat het keyword **var** hier handig is: het verkort de ellenlange stukken code waarin we toch maar gewoon het datatype herhalen dat ook al rechts van de toekenningsoperator staat.

8.3 Een complexere generieke klasse

Voorgaand voorbeeld is natuurlijk maar de tip van de ijsberg. We kunnen bijvoorbeeld volgende klasse maken die we kunnen gebruiken met eender welk type om de meetwaarde van een meting in op te slaan. Merk op hoe we op verschillende plaatsen in de klasse het element T gebruiken als een datatype:

```

1 internal class Meting<T>
2 {
3     public T Waarde {get;set;}
4     public Meting(T waardein)
5     {
6         Waarde = waardein;
7     }
8 }
```

Een voorbeeldgebruik van dit nieuwe type kan zijn:

```

1 var m1 = new Meting<int>(44);
2 Console.WriteLine(m1.Waarde);
3 var m2 = new Meting<string>("slechte meting");
4 Console.WriteLine(m2.Waarde);
```

8.4 Meerdere types in generics

Zoals reeds eerder vermeld is de T aanduiding enkel maar een afspraak. Je kan echter zoveel T-parameters meegeven als je wenst. Stel dat je bijvoorbeeld een klasse wenst te maken waarbij 2 verschillende types kunnen gebruikt worden. De klassedefinitie zou er dan als volgt uit zien:

```

1 internal class DataBewaarder<Type1, Type2>
2 {
3     public Type1 Waarde1 {get;set;}
4     public Type2 Waarde2 {get;set;}
5     public DataBewaarder(Type1 w1, Type2 w2)
6     {
7         Waarde1 = w1;
8         Waarde2 = w2;
9     }
10 }
```

Een object aanmaken zal nu als volgt gaan:

```

1 DataBewaarder<int, string> d1 = new DataBewaarder<int, string>(4, "Ok
");
```

8.5 Constraints

We willen soms voorkomen dat bepaalde types wel of niet gebruikt kunnen worden in je zelfgemaakte generieke klasse.

Stel bijvoorbeeld dat je een klasse schrijft waarbij je de `CompareTo()` methode wenst te gebruiken. Dit gaat enkel indien het type in kwestie de `IComparable` interface implementeert. We kunnen als **constraint** (*beperking*) dan opgeven dat de volgende klasse enkel kan gebruikt worden door klassen die ook effectief die interface implementeren (en dus de `CompareTo()`-methoden hebben). We doen dit in de klasse-definitie met het nieuwe `where` keyword. We zeggen dus letterlijk: “waar *T* overerft van *IComparable*”:

```

1 internal class Wijziging<T> where T : IComparable
2 {
3     public T VorigeWaarde {get; set;}
4     public T Huidigwaarde {get; set;}
5     public Wijziging(T vorig, T huidig)
6     {
7         VorigeWaarde = vorig;
8         Huidigwaarde = huidig;
9     }
10
11    public bool IsGestegen()
12    {
13        return Huidigwaarde.CompareTo(VorigeWaarde) > 0;
14    }
15 }
```

Volgende gebruik van deze klasse zou dan True op het scherm tonen:

```

1 Wijziging<double> w = new Wijziging<double>(3.4, 3.65);
2 Console.WriteLine(w.IsGestegen());
```

8.6 Mogelijke constraints

Verschillende zaken kunnen als constraint optreden. Naast de verplichting dat een bepaalde interface moet worden geïmplementeerd kunnen ook volgende constraints gelden (bekijk de online documentatie voor meer informatie hierover):

- Enkel value types.
- Enkel klassen.
- Moet default constructor hebben.
- Moet overerven van een bepaalde klasse.

9 Records & structs

9.1 Records

Sinds C# 9.0 is het ook mogelijk om zogenaamde record-klassen te maken. Erg vaak schrijf je klassen die niet meer moeten doen dan wat data eenmalig wegschrijven en onthouden, dat je dan vervolgens via readonly getters kunt uitlezen, zoals:

```

1 internal class Student
2 {
3     public Student(string naam, int geboorteJaarIn, bool
4                     isIngeschreven)
5     {
6         Naam = naam;
7         Geboortejaar = geboorteJaarIn;
8         IsIngeschreven = isIngeschreven;
9     }
10    public string Naam {get;}
11    public int Geboortejaar {get;}
12    public bool IsIngeschreven {get;}
13 }
```

Wanneer je een dergelijke klasse nodig hebt kan dit sinds C# 9.0 vereenvoudigd geschreven worden als een record:

```

1 public record Student
2 {
3     public string Naam { get; init; }
4     public int Geboortejaar { get; init; }
5     public bool IsIngeschreven { get; init; }
6 }
```

Het **init** keyword geeft aan dat deze auto-property eenmalig kunnen geset worden bij het aanmaken van het record via de object initializer syntax:

```

1 Student eenNieuweStudent = new Student
2             {   Naam = "Tim",
3                 Geboortejaar = 1981,
4                 IsIngeschreven = false
5             };
```

Er zijn nog tal van extra's die je krijgt met records (o.a. eenvoudig objecten vergelijken) maar die ga ik niet bespreken.

