

Programmeren in C#

Programmeren

K. Provoost

Schooljaar: 20 – 20

Naam:

Klas: AD 3^e graad



Dit boek dient als handboek voor de studierichting Applicatie- en Databeheer in de derde graad van het Vlaams secundair onderwijs. Het is bedoeld voor leerlingen die mogelijk nog nooit hebben geprogrammeerd en is zo opgebouwd dat het aan alle leerplandoelstellingen voldoet.

In het eerste deel behandelen we de basisstructuren zoals variabelen, keuzes, lussen (loops) en methoden. Daarna leren we hoe we efficiënter kunnen programmeren met object georiënteerd programmeren. Tot slot duiken we in de wondere wereld van grafische applicaties met WPF.

Ik heb geprobeerd de nadruk te leggen op aspecten van de programmeertaal die nuttig zijn voor je toekomstige (professionele) programmeercarrière. Hopelijk geeft dit boek je een stevige basis, zodat je ook vlot met andere programmeertalen zoals Java, C++ of Python aan de slag kunt.

Veel succes en plezier met programmeren!

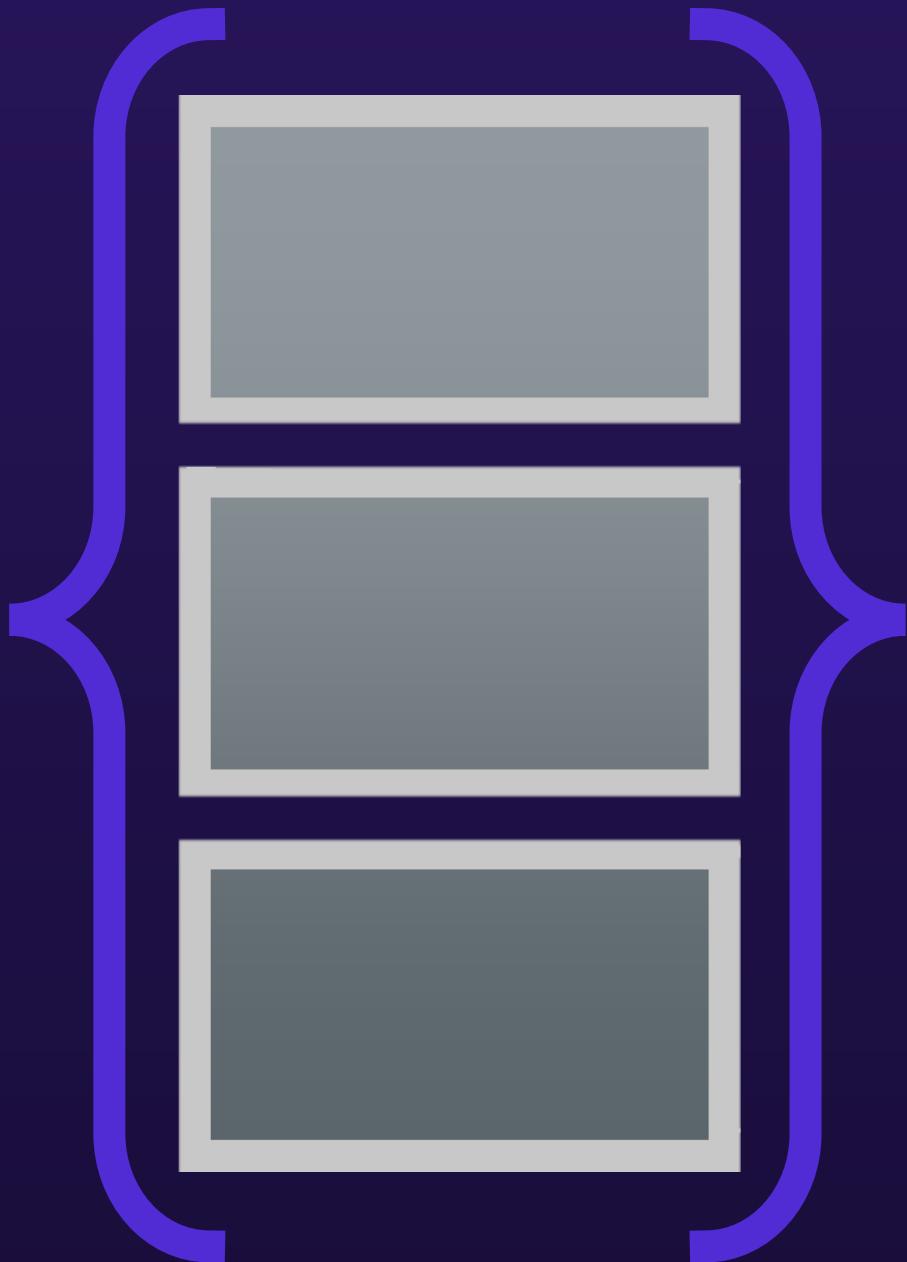


Deel 1: Gestructureerd programmeren	9
0 Inleiding	11
0.1 Principes van programmeren	11
0.2 Mijn eerste C#-toepassing	19
0.3 De opbouw van een programma	23
1 Algoritmes.....	25
1.1 Programmeeralgoritmes.....	25
1.2 Probleemoplossen denken	26
1.3 Rekenen in C#	31
2 Variabelen.....	33
2.1 Inleiding	33
2.2 Gegevenstypen	33
2.3 De naamgeving	35
2.4 Het bereik	36
2.5 Het declareren en initialiseren van variabelen.....	37
3 Operatoren.....	41
3.1 Rekenkundige operatoren	41
3.2 Incrementeer- en decrementeeroperatoren	41
3.3 Toekenningsoperatoren	42
3.4 Vergelijkingsoperatoren	42
3.5 Logische operatoren	43
3.6 Conditionele operatoren	44
4 Selecties	45
4.1 Bouwstenen van code	45
4.2 Selectie statements	46
5 Iteraties.....	51
5.1 Soorten iteraties	51
5.2 Voorwaardelijke herhaling	51
5.3 Begrensde herhaling	53
6 Foutencontrole.....	55
1.1 Soorten fouten.....	55
1.2 Debuggen: breakpoints gebruiken	57
1.3 Debuggen van syntaxfouten.....	57
1.4 Debuggen van Runtimefouten.....	59
1.5 Debuggen van Logische fouten.....	63

7	Methoden.....	65
7.1	Het belang van gestructureerde programmacode	65
7.2	Wat zijn methoden?.....	65
7.3	Procedures en functies	68
7.4	Parameters.....	69
7.5	Method overloading	70
Deel 2: Object georiënteerd programmeren.....		71
8	Basisprincipes van OOP.....	73
8.1	Object georiënteerd programmeren	73
9	Objecten.....	77
9.1	Wat zijn objecten?	77
9.2	Klassen en objecten	77
9.3	Boodschappen.....	78
9.4	Objecten aanmaken	79
9.5	Objecten gebruiken.....	81
9.6	Objecten opruimen	83
9.7	De willekeurige klasse Random.....	83
9.8	De tekenreeksklasse String	85
9.9	De tekenreeksmanipulatieklasse StringBuilder	88
9.10	String Interpolatie	89
10	Arrays	92
10.1	Wat zijn arrays?.....	92
10.2	Arrays creëren.....	93
10.3	Arrays gebruiken	93
10.4	Waarden van arrays tonen.....	94
10.5	Arrays van objecten	95
10.6	Arrays van arrays.....	96
10.7	Eigenschappen van arrays.....	99
10.8	Methoden van de klasse Array	99
10.9	Arrays, tekens en tekenreeksen.....	101
11	Klassen	103
11.1	Wat is een klasse?	103
11.2	Declaratie van de klasse.....	103
11.3	De klasse-omschrijving (body)	105
11.4	Access modifiers.....	106
11.5	Velden	107
11.6	Eigenschappen	107
11.7	Methoden	112

11.8	Constructors	118
12	Statische leden en hulpklassen	125
12.1	Het sleutelwoord static.....	125
12.2	Statische leden.....	125
12.3	Static vs. non-static.....	128
12.4	Static Random number generators	129
13	Enumeraties	131
13.1	Wat zijn enumeraties?.....	131
13.2	De bestaansreden van enumeraties.....	131
13.3	Enumeraties aanmaken.....	134
13.4	enum-variabelen gebruiken	136
14	Collectieklassen	139
14.1	Dynamische verzamelingen	139
14.2	De collectieklasse List	140
14.3	De collectieklasse Queue.....	141
14.4	De collectieklasse Stack.....	142
14.5	De collectieklasse Dictionary.....	143
15	Overerving en klasse-hiërarchie	147
15.1	Nog meer OOP-concepten.....	147
15.2	Overerving	147
15.3	Polymorfisme.....	158
15.4	Abstractie.....	160
16	Relaties tussen klassen	163
16.1	Associaties	163
16.2	Aggregatie en compositie	164
16.3	Compositie en aggregatie toepassen	166
16.4	Compositie- en aggregatie-objecten gebruiken	171
16.5	Code hergebruik: overerving of compositie?	171
17	Gevorderde OOP-concepten	173
17.1	Polymorfisme (bis).....	173
17.2	De oerkLASSE Object	180
18	Interfaces	185
18.1	Wat is een interface?.....	185
18.2	Interfaces en klassen	187
18.3	Interfaces en overerving.....	189
18.4	Interfaces en polymorfisme.....	191
18.5	Overzicht populairste .NET-interfaces.....	197

Deel 1: Gestructureerd programmeren



0.1 Principes van programmeren

0.1.1 Wat is programmeren

Programmeren, of *coderen*, is het schrijven van een computerprogramma. Zo'n programma bestaat uit een concrete reeks instructies, of **algoritmes**, die een computer kan uitvoeren. De programmacode die je ziet en bewerkt wordt de **broncode** genoemd.

Het is de taak van een softwareontwikkelaar of programmeur om een computerprogramma te schrijven. Daarvoor wordt eerst een uitgebreide analyse gedaan. Er wordt dus precies nagedacht over wat er gegeven en gevraagd is. Aan de hand van deze gegevens stelt de programmeur een stappenplan of algoritme op. Vervolgens worden de instructies vertaald naar een **programmeertaal**.

De instructies moeten worden opgebouwd volgens zeer strenge regels van de programmeertaal; de **syntax**. Zelfs kleine typfouten kunnen ervoor zorgen dat een computer niet weet welke taak of bewerking het moet uitvoeren. Het is dus erg belangrijk dat je de nodige aandacht speudeert aan het vermijden van typ- en structurele fouten!

Nochtans begrijpt een computer alleen **machinetaal**. Elke instructie van de programmeertaal moet dus eerst worden omgezet naar een hele reeks nulletjes (0) en eentjes (1), **bits** genaamd. Het omzetten van de broncode naar machinetaal noemt men **compilieren**. Deze taak wordt uitgevoerd door een assembler, compiler of interpreter. De programmeertaal C# maakt gebruik van een **compiler**.

Concreet kunnen we stellen dat programmeren meer is dan alleen het schrijven een programma. Het is namelijk ook een vorm **van probleemplossend denken**:

- Een **probleem** herleiden naar **eenvoudige delen**.
- De verschillende delen van het probleem **visualiseren**.
- De **oplossingen** formuleren in een eenvoudige taal die zelfs de computer kan begrijpen.



0.1.2 Soorten programmeertalen

Zoals je nu weet is een programmeertaal een formele taal waarin de opdrachten worden geschreven die een computer moet uitvoeren. Deze talen hebben een andere syntax en grammatica dan natuurlijke talen zoals Frans of Engels. Code die in een programmeertaal geschreven is, mag namelijk slechts op één manier “begrepen” worden door de computer.

Programmeertalen hebben trouwens zelf al een hele evolutie doorgemaakt. Er zijn op dit moment drie kenmerkende generaties van programmeertalen. Bij elke generatie is er een belangrijke vooruitgang in de werking van de programmeertaal gekomen:

- **1GL: Machine Code**

De instructies bestaan uit nulletjes en eentjes. Dergelijke talen staan ‘dichter bij de processor’.

```
11101001 10001010 10010101 00000000 00000000 11101001
00000000 00000000 00110011 11000000 10100000 11001100
00000000 11000011 10100001 11011111 10110000 01000000
11001100 10111001 10110000 00000000 00000000 00000000
01110100 00111001 10000011 00111101 11010111 10110000
00000000 01110011 00001010 10111000 11100010 00000000
11101000 11100011 11111111 11111111 11111111 01101000
00000000 00000000 01101010 01000000 11101000 10110110
00000000 000001011 11000000 01110101 000001010 10111000
00000000 00000000 11101000 11001001 11111111 11111111
11111111 00110101 11010111 10110000 01000000 00000000
10010111 00000000 11000000 11000011 10111001 10110000
00000000 000001011 11001001 01110100 00011001 11101000
00000000 00000000 10100011 11010111 10110000 01000000
11111000 00000000 01110011 10100101 10111000 11100010
```

- **2GL: Assembler**

Er wordt gebruik gemaakt van gemakkelijker te onthouden lettercodes in plaats van nulletjes en eentjes. Net als machinetaal is dit een *lagere programmeertaal*.

```
10 stack segment
11 dw 128 dup(0)
12 ends
13
14 code segment
15 start:
16 ; set segment registers:
17 mov ax, data
18 mov ds, ax
19 mov es, ax
20
21 ; add your code here
22
23 lea dx, msg
24 mov ah, 9
25 int 21h ; output string at ds:dx
26
27 ;line down
28 MOV AH, 2
29 MOV DL, 0Ah
30 INT 21h
31 MOV DL, 0Dh
32 INT 21h
33
34 mov string[4], "$" ;end of string is 4
35 xor ax, ax
36 mov bx, 1 ;the first integer
37 print1:
38     mov al, bl
39     add ax, 48
40     mov string[2], al
41     lea dx, string+2
42     mov ah, 9
43     int 21h
44     inc bx
45     cmp bx, 9
46     jne print1
47
48 end start
```

- **3GL: Procedurele- en objectgeoriënteerde talen**

De instructies door een compiler vertaald naar machinecode. Vanaf hier spreken we van een *hogere programmeertaal*.

```
// Hello World! program
namespace HelloWorld
{
    class Hello {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World!");
        }
    }
}
```

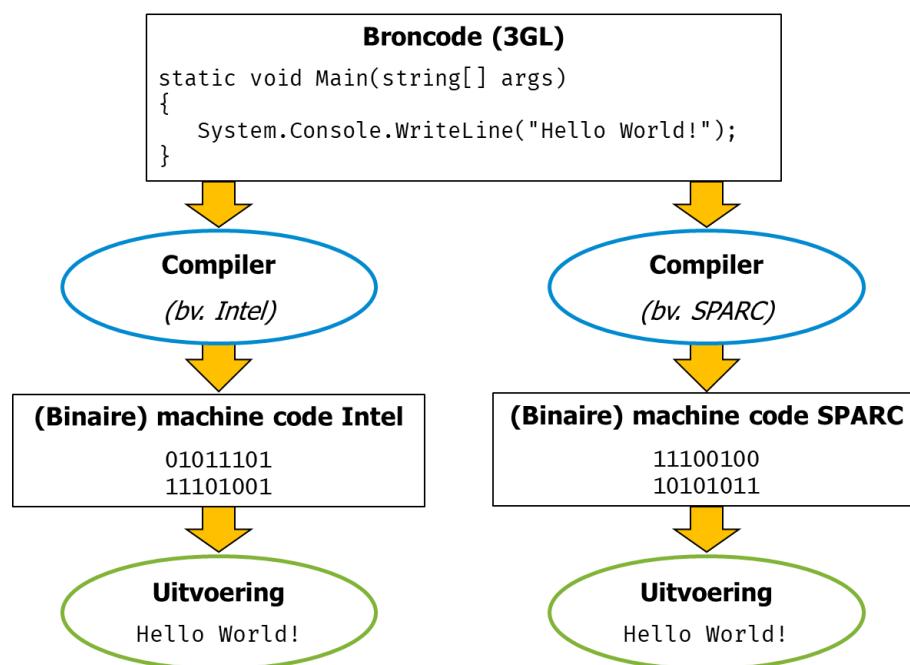
Vandaag de dag bestaan er heel wat verschillende programmeertalen, elk met hun eigen syntax:

- C, C++, C#, Java
- Visual Basic, VB.Net
- Python
- ...

0.1.3 Van programmeertaal tot machinetaal

Zoals je ondertussen al weet kent de computer alleen de machinetaal. Iedere instructie die hij uitvoert, is eigenlijk een binair getal dat is opgeslagen in het werkgeheugen. De processor haalt dit getal uit het werkgeheugen en voert de instructie uit. **Deze binaire codes en de overeenkomstige instructies zijn specifiek voor iedere processor(familie)**. Zo heeft een processor van *Intel* een andere instructieset dan een *SPARC* processor. Beide zijn op binair niveau dus **niet compatibel**. Dit wilt zeggen dat processoren elkaars binaire codes niet kunnen gebruiken.

Sommige **hogere programmeertalen** zoals C, C#, Java, ... zijn wel **overdraagbaar**. Dat wil zeggen dat een programma geschreven in die taal onafhankelijk is van het type processor dat nadien de instructies zal uitvoeren. De programmacode wordt nadien vertaald naar de juiste binaire instructies voor die specifieke processor.



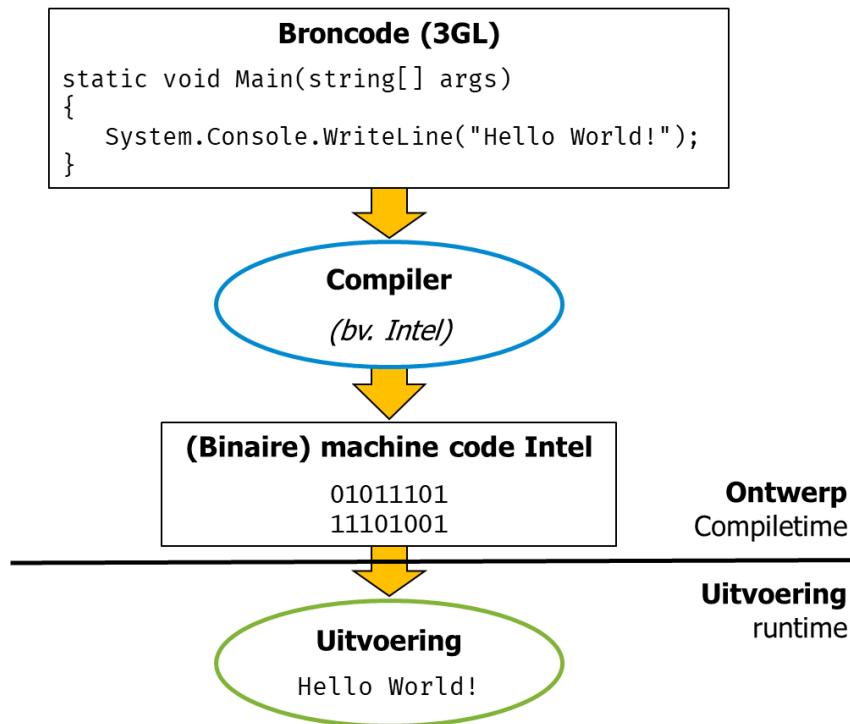
Op basis van dit vertaalmoment worden programmeertalen ingedeeld in twee groepen:

- Gecompileerde programmeertalen
- Geïnterpreteerde programmeertalen

0.1.3.1 Gecompileerde programmeertalen

Bij compileerde programmeertalen wordt de broncode weggeschreven in een tekstbestand. Deze broncode wordt vervolgens vertaald naar de binaire objectcode die wordt weggeschreven in een uitvoerbaar binair bestand. Men noemt dit proces **compilieren** en dit wordt gedaan door een **compiler**.

Nadien wordt de binaire code van het bestand ingeladen en uitgevoerd door de processor. Iedere processor heeft zijn eigen compiler die de programmacode kan omzetten in de juiste binaire codes voor de processor (Vaes, 2014). De vertaalslag wordt dus gedaan **voor de uitvoering van het programma**.



Voordelen:

- **Snelle uitvoering** van de programma's omdat binaire code rechtstreeks wordt uitgevoerd.
- De broncode is **overdraagbaar**. Je kan programma's in één taal schrijven en door verschillende machines laten uitvoeren.
- De broncode is binair en dus **goed beschermd** omdat het moeilijk aangepast of gebruikt kan worden door anderen.

Nadelen:

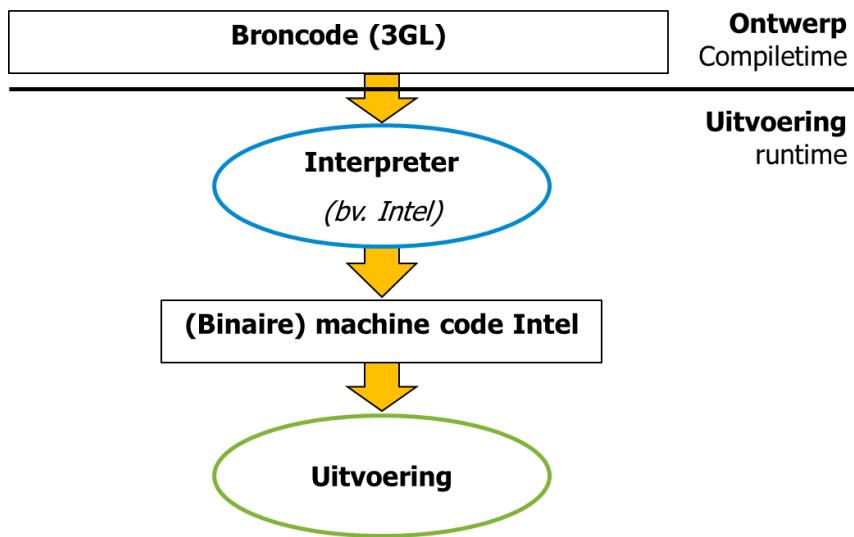
- Voor elk type processor moet een afzonderlijk binair bestand (objectcode) gemaakt worden. De uitvoerbare programma's zijn niet overdraagbaar en dus **processor-afhankelijk**.
- Voor elk besturingssysteem zoals Windows, Linux, MacOS, ... moet het programma afzonderlijk gecompileerd worden omdat de interactie met het besturingssysteem telkens anders is. Zowel de broncode als de objectcode zijn dus **besturingssysteem-afhankelijk**.
- Programma's moeten altijd eerst gecompileerd worden vooraleer ze getest kunnen worden. Na iedere aanpassing volgt er terug een compilatie. **Debugging en testing** vraagt dus altijd extra stap en is daardoor **omslachtig en tijdrovend**.

0.1.3.2 Geïnterpreteerde programmeertalen

Bij geïnterpreteerde programmeertalen wordt de vertaalslag gedaan **tijdens de uitvoering** van het programma. De broncode wordt ook hier opgeslagen in een tekstbestand en tijdens de uitvoering van het programma worden de programmaregels stap voor stap geïnterpreteerd en uitgevoerd. Er is dus geen intermediair bestand met de objectcode.

Het **interpreteren** wordt in dit geval gedaan door een **interpreter**.

Script-talen zoals Javascript, Visual Basic Script, ... zijn over het algemeen geïnterpreteerde talen. In dit geval is het bijvoorbeeld de internetbrowser die dienst doet als de interpreter (Vaes, 2014).



Voordelen:

- De programmacode kan **snell aangepast** en **onmiddellijk geëvalueerd** worden.
- Programma's zijn **onmiddellijk overdraagbaar** omdat de programmacode onafhankelijk is van de processor en het besturingssysteem. De vertaling gebeurt immers door de interpreter. Dit maakt dit soort talen uitermate geschikt voor de verspreiding via het internet. Er is slechts één broncode die rechtstreeks kan dienen voor verschillende platformen.

Nadelen:

- De programma's werken **traag** omdat alle programmastappen telkens weer geïnterpreteerd moeten worden.
- Het is **moeilijk** om de broncode **te beschermen** tegen illegaal gebruik. De programma's bestaan uit tekstbestanden die andere kunnen kopiëren en aanpassen.

0.1.4 De programmeertaal C#



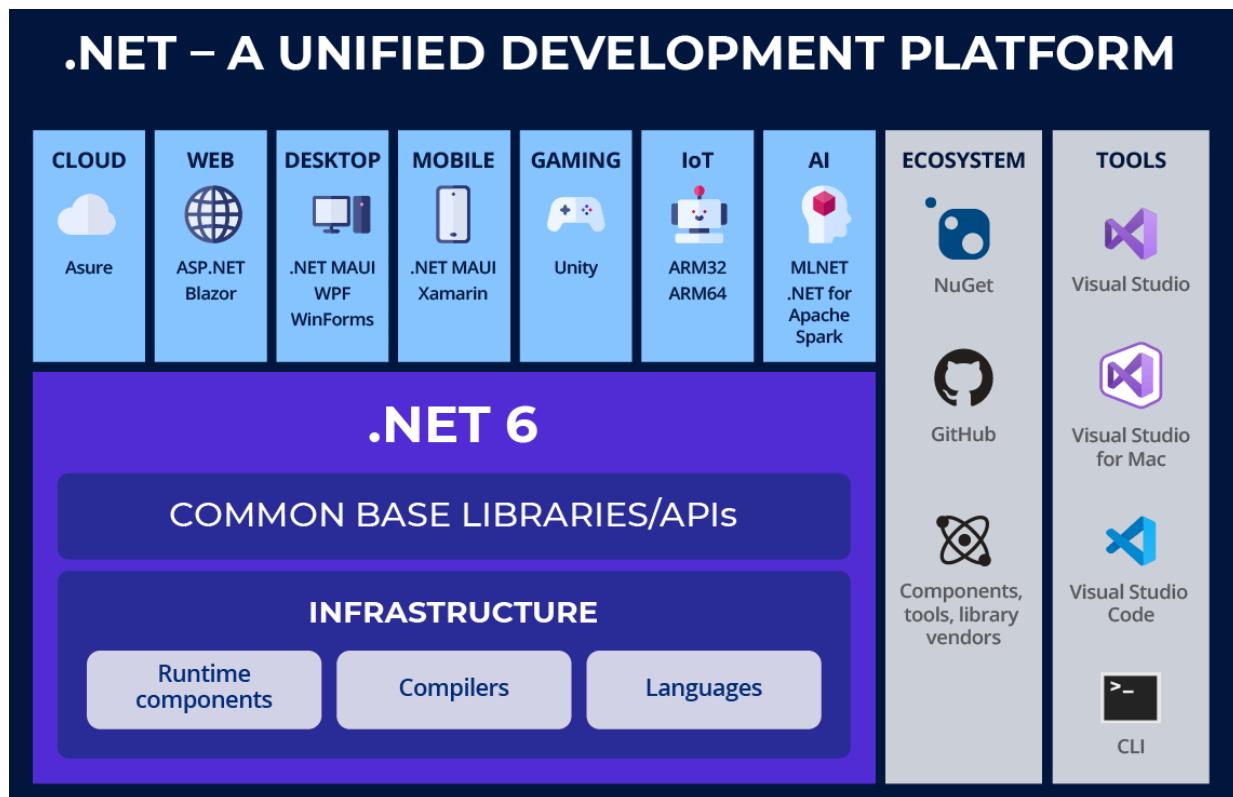
C# wordt uitgesproken als "C-Sharp". Het is een **objectgeoriënteerde** programmeertaal gemaakt door **Microsoft** onder leiding van Anders Hejlsberg in het jaar 2000. C# heeft wortels in de C-familie en de taal ligt qua syntax en gebruik dicht bij andere populaire talen zoals C++ en Java. De eerste versie werd uitgebracht in 2002. De nieuwste versie, C# 11, werd uitgebracht in november 2022.

C# kent veel **doeleinden** en kan gebruikt worden voor:

- Mobiele applicaties (Xamarin);
- Desktop-applicaties (.NET core, WPF, .NET MAUI);
- web applicaties (Blazor);
- Videogames (Unity);
- AI/machine learning (ML.NET);
- Database-applicaties;
- ...

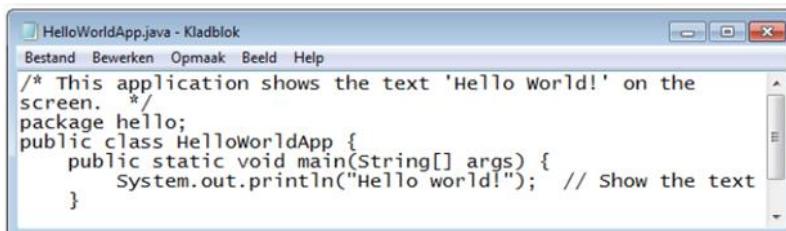
0.1.5 Het .NET Framework

C# is een onderdeel van het **.NET Framework**. Een framework (of raamwerk) staat ten behoeve van de naadloze samenwerking van applicaties en bibliotheken geschreven in verschillende programmeertalen voor het maken van applicaties, in dit geval voor Windowsapplicaties. Dit zeggen dat het niet gelimiteerd is tot alleen de programmeertaal C#. Meerdere talen zoals F#, VB.NET, ... kunnen er immers op worden gebruikt om applicaties te maken.



0.1.6 De ontwikkelomgeving Visual Studio (Code)

Programmacode kan geschreven worden in eender welke tekstverwerker zoals kladblok, notepad++, ...

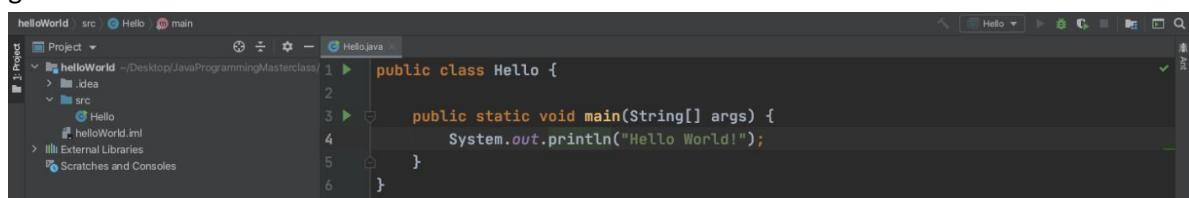


De bovenstaande afbeelding toont de uitvoering van het programma ‘Hello World’, geschreven in de programmeertaal Java. Je merkt echter dat hier weinig ondersteuning wordt aangeboden waardoor je als het ware de taal al bijna “van buiten” moet kennen.

Om **efficiënter** te werken, bestaan er echter **speciale programma’s** die verschillende taken bij het schrijven van programmacode kunnen automatiseren en ondersteunen. Zo’n ontwikkelomgeving staat ook wel gekend als een **Integrated Development Environment (IDE)**.

De voorkeur gaat dus uit naar werken met een Integrated Development Environment (IDE) en dit omwille van de volgende ingebouwde **ondersteuningsfuncties**:

- **Editor:** een schrijfruimte om de broncode van het programma te schrijven en aan te passen.
- **Compiler:** het programma wordt eerst gecontroleerd en vervolgens uitgevoerd als het geen fouten detecteert tijdens de opbouw ervan.
- **Debugger:** helpt om fouten in het programma te diagnosticeren, op te sporen en te verhelpen.
- **Documentatie** van de gebruikte programmeertaal en de bijbehorende API's (meestal in de vorm van bibliotheken) om uitleg aan de ontwikkelaar te verschaffen over de code die ze wensen te gebruiken, zijn reeds geïntegreerd in het programma. Een Application Programming Interface (API) is een software-interface die het mogelijk maakt dat twee applicaties met elkaar kunnen communiceren. Een bibliotheek is een verzameling code die door programma's kan worden gebruikt.



 De bovenstaande afbeelding toont ons opnieuw het programma 'Hello World' geschreven in de programmeertaal *Java*, maar ditmaal in de IDE genaamd *IntelliJ*.

Voor de meeste programmeertalen bestaan er meerdere IDE's:

- **C# en C++:** Visual Studio, Visual Studio Code, Rider IDE, ...
- **Python:** PyCharm, Pydev, IDLE, ...
- **Java:** IntelliJ IDEA, Eclipse IDE, Apache NetBeans, ...
- ...

0.1.6.1 Visual Studio of Visual Studio Code?



Visual Studio: Dit is een uitgebreide IDE die bekend staat als een "heavyweight" IDE. Het is ontwikkeld door Microsoft en is voornamelijk gericht op .NET-ontwikkeling (C#, F#, VB.NET) en ondersteunt ook andere talen zoals C++, Python, enzovoort. Het biedt een breed scala aan functies, waaronder GUI-ontwerptools, debugging, profiling en meer.



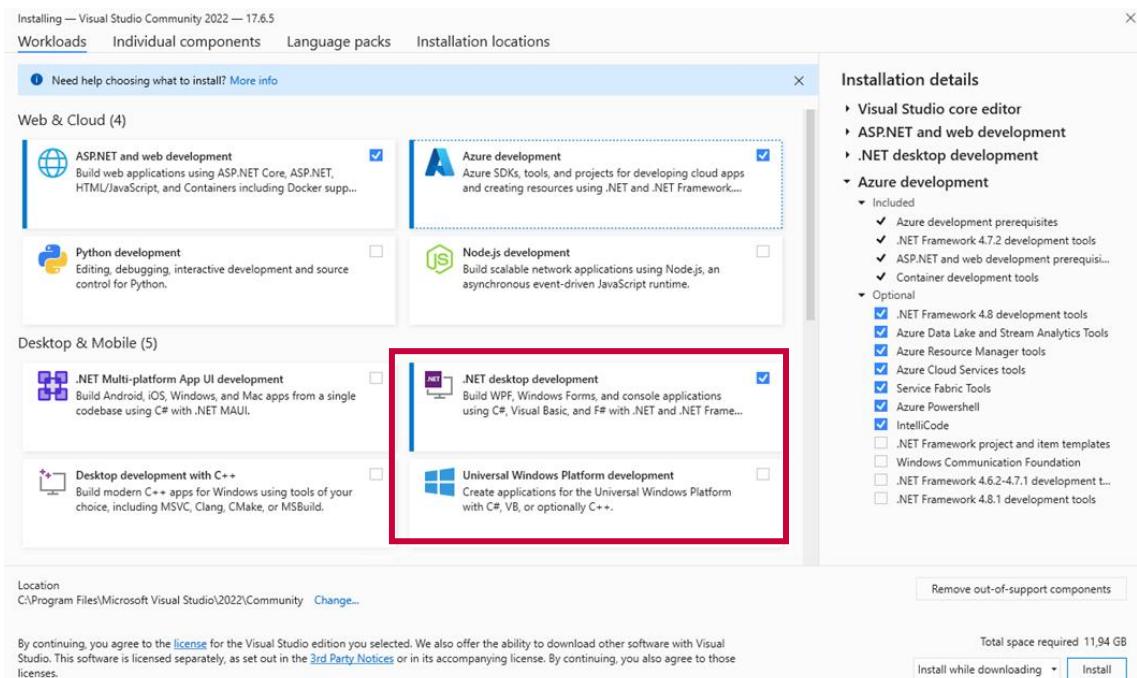
Visual Studio Code: Dit is een lichtgewicht, op tekst gebaseerde code-editor die bedoeld is voor een breed scala aan programmeertalen. Het is ontwikkeld door Microsoft, maar is open source en platformonafhankelijk, wat betekent dat het op Windows, macOS en Linux kan draaien.

Kortom, als je op zoek bent naar een zware IDE met uitgebreide functionaliteit voor .NET-ontwikkeling, is Visual Studio de betere keuze. Als je echter op zoek bent naar een lichtgewicht, aanpasbare code-editor die voor veel programmeertalen kan worden gebruikt, dan is Visual Studio Code waarschijnlijk de betere optie. Het komt vaak neer op de specifieke behoeften van het project en persoonlijke voorkeuren van de ontwikkelaar. **In deze cursus werken we echter met de IDE Visual Studio.**

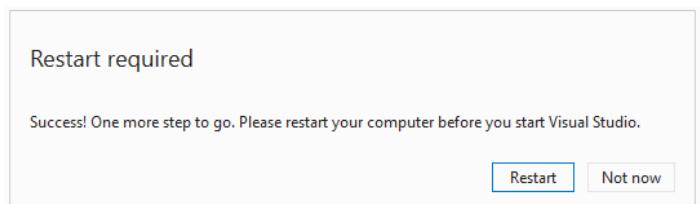
0.1.6.2 Visual Studio installen

Je kan het programma downloaden via <https://visualstudio.microsoft.com/downloads/>. Let op, er zijn verschillende versies! Kies voor de **gratis versie ‘Community’**.

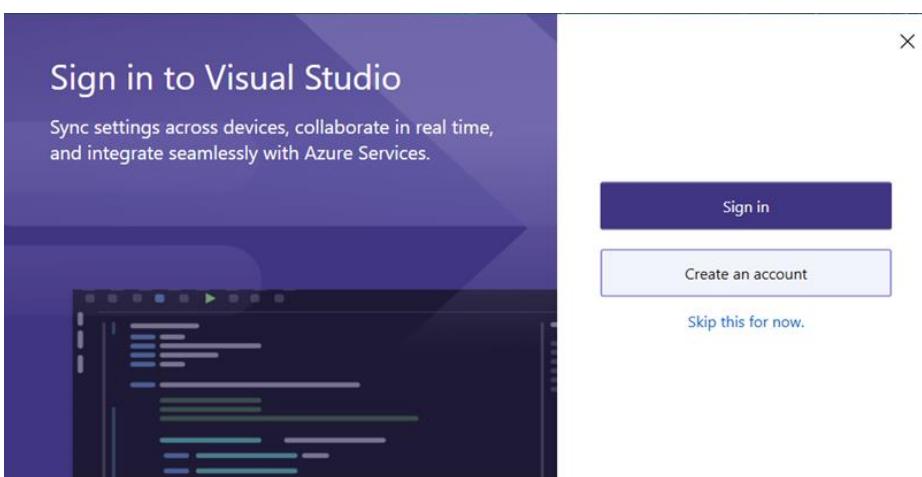
1. Download de Community-versie.
2. Selecteer de ‘.NET desktop development’ + ‘Universal Windows Platform development’ workload en klik vervolgens links onderaan op ‘Install’.



3. Herstart je computer van zodra de download compleet is.



4. Meld je na de installatie aan met je TAB-account en je kan beginnen met programmeren in Visual Studio.

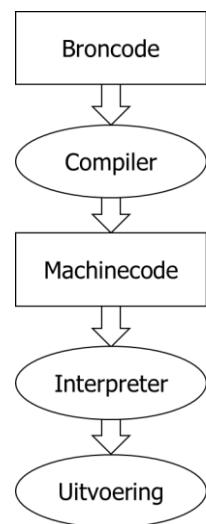


0.2 Mijn eerste C#-toepassing

Wanneer je een nieuwe programmeertaal leert, is het een beetje een traditie om het eerste programma de tekst “Hello World” te laten tonen op het scherm.

Het schrijven van een C#-programma gebeurt in 3 fasen:

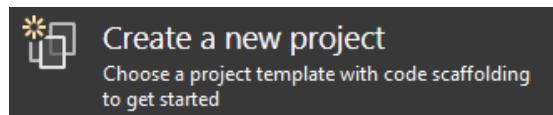
1. Het maken van de **broncode**.
2. Het compileren van de broncode tot **machinecode**.
(In het .NET-framework is er nog een tussenstap: IL- code)
3. Het **uitvoeren** van het programma.



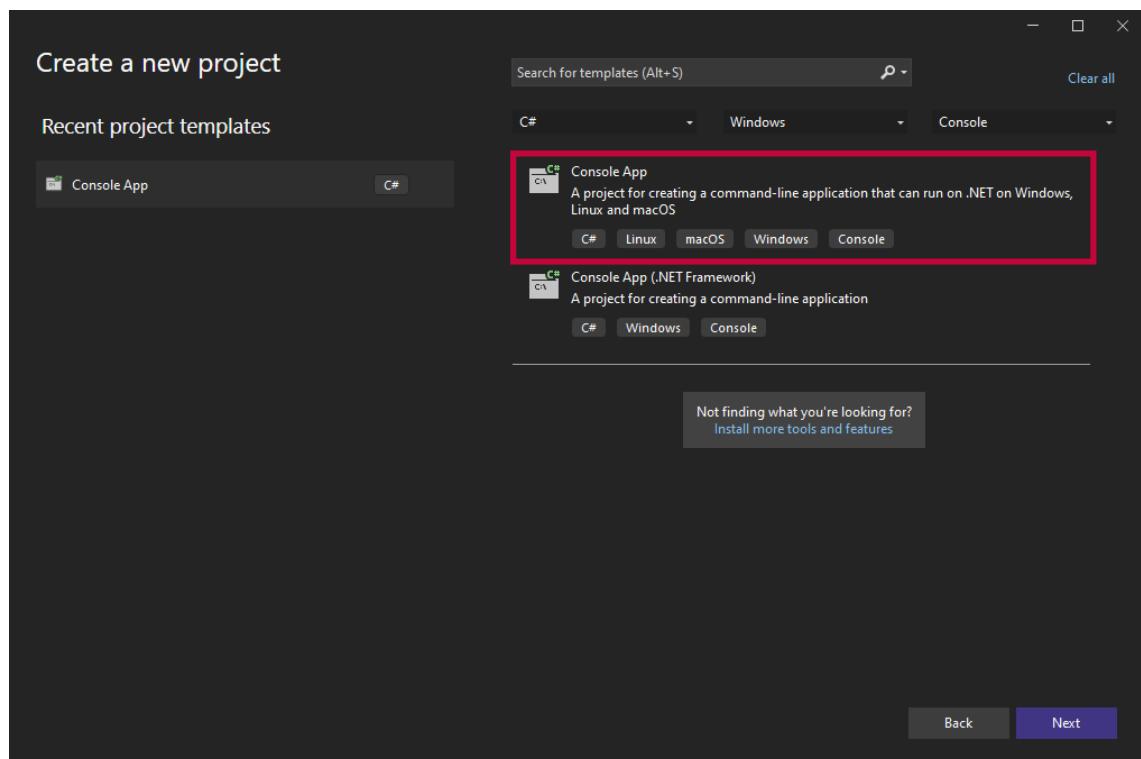
0.2.1 Het maken van de broncode

Voordat we beginnen met het schrijven van de broncode, moeten we eerst een **nieuw project** aanmaken waarin we het programma wensen te maken.

1. Open het programma Visual Studio.
2. Selecteer ‘Create a new Project’ in het startscherm van Visual Studio.



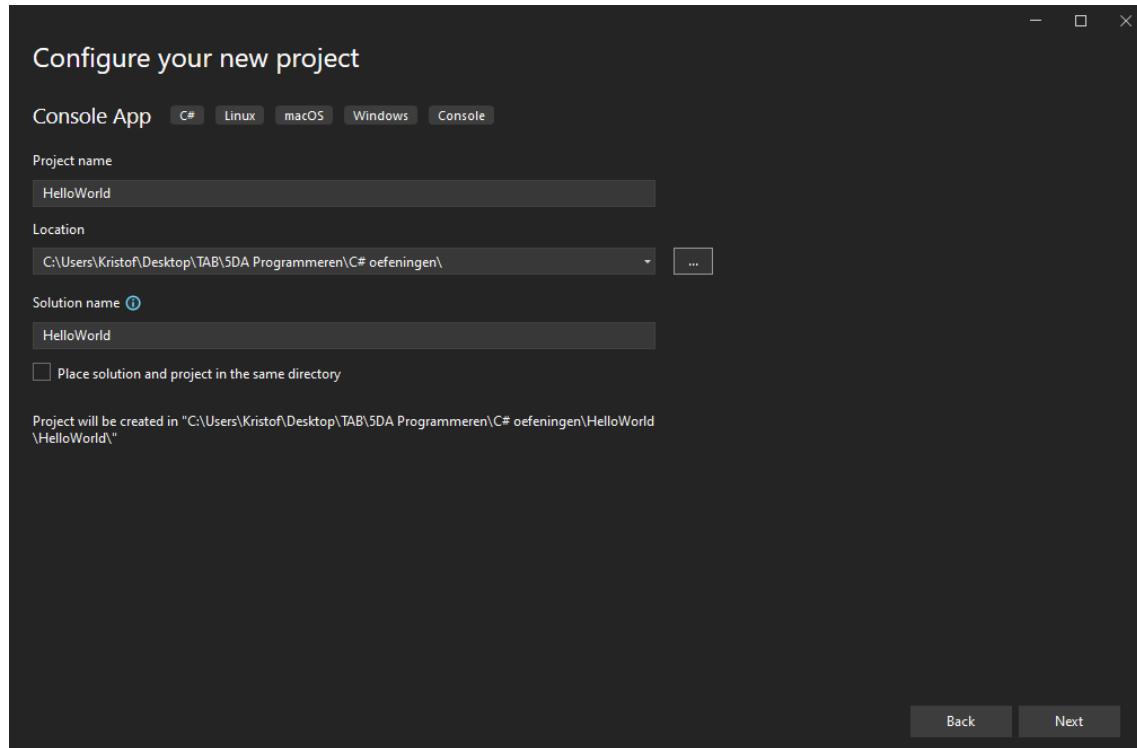
3. Selecteer de template ‘Console App’ voor C# en klik op ‘Next’.



4. Geef het project de naam “HelloWorld”.

Bij ‘Location’ geef je aan waar je het project wenst op te slaan door een bestandlocatie te selecteren. Solution name is de naam van de folder van de applicatie die je aanmaakt.

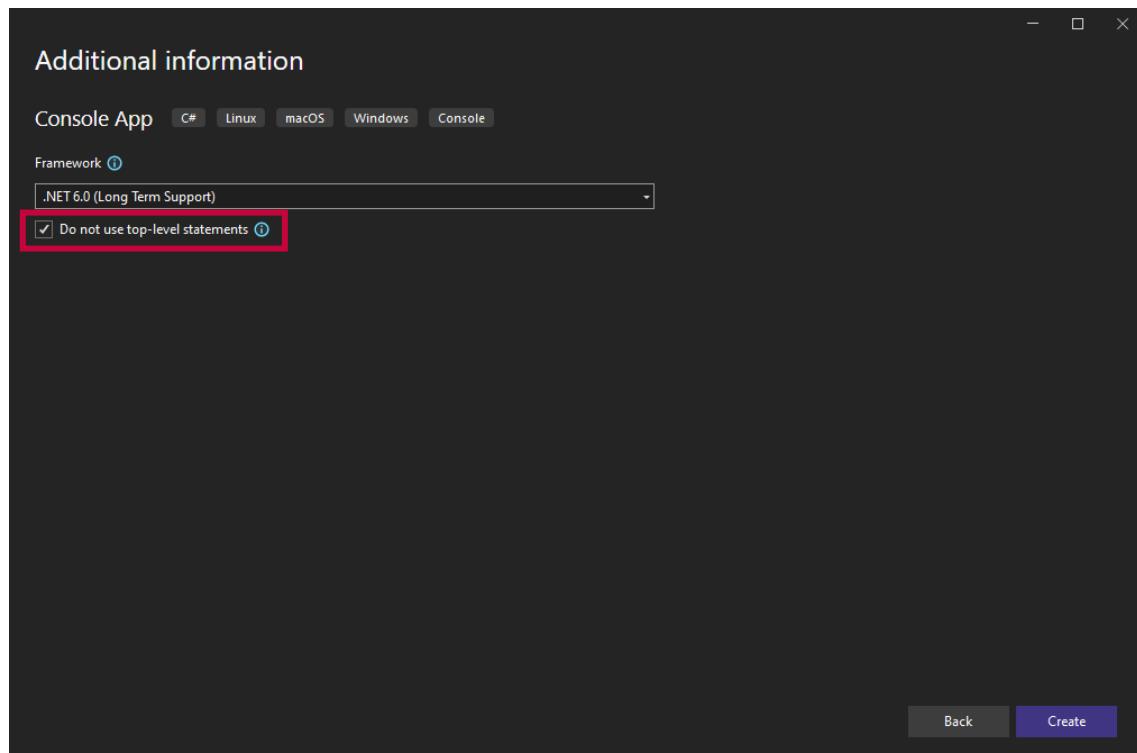
Klik vervolgens op ‘Next’.



5. Kies bij ‘Framework’ om gebruik te maken van “.NET 6.0” of hoger.

Vink ‘Do not use top-level statements’ **AAN** bij het maken van een project. Dit zorgt ervoor dat je de namespace, internal class en methode Main() te zien krijgt.

Klik vervolgens op ‘Create’.



6. De broncode van je programma kan nu geschreven worden. Neem de onderstaande code over (let op hoofdletters en kleine letters!):

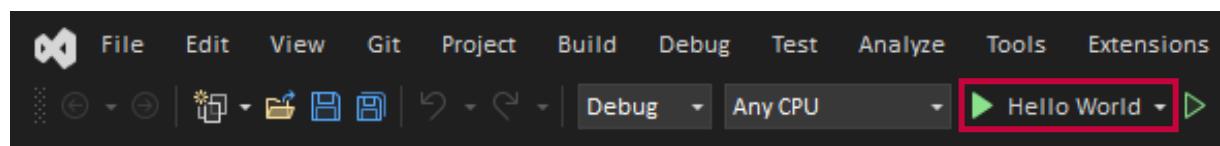
```
1  namespace HelloWorld
2  {
3      0 references
4      internal class Program
5      {
6          0 references
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello, World!");
10         }
11     }
```

De broncode wordt geschreven in een tekstbestand met de extensie “.cs” (bv. mijnprogramma.cs).

0.2.2 Het compileren van de broncode

In de tweede fase wordt de broncode rechtstreeks door de C#-compiler gecompileerd naar machine code, meestal de "csc.exe". De compiler genereert een uitvoerbaar bestand met de extensie ".exe" (bv. MijnProgramma.exe).

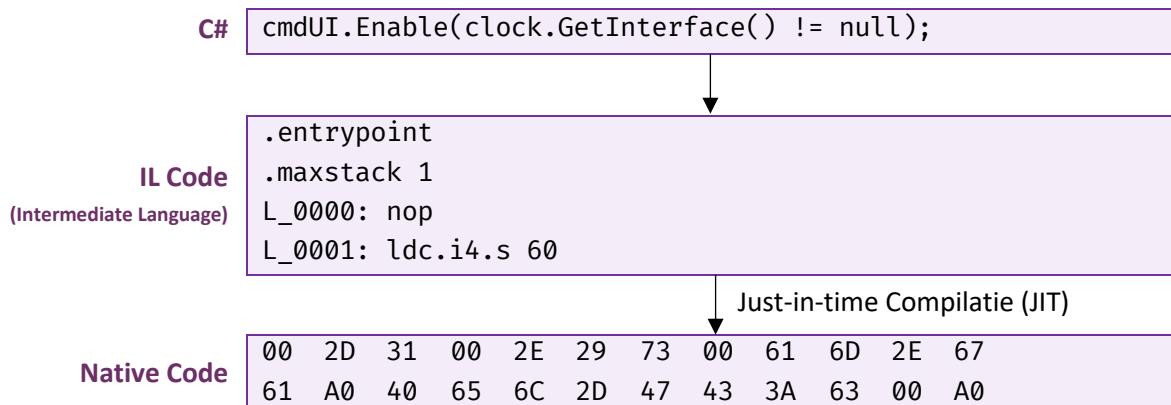
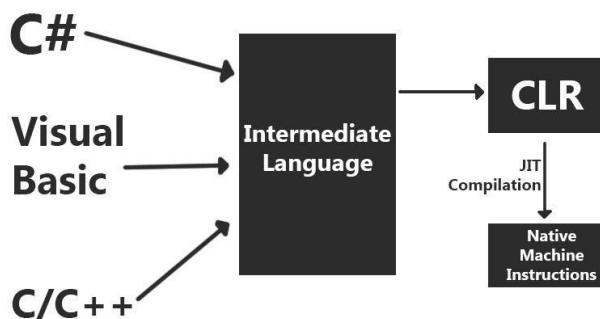
In Visual Studio wordt de broncode gecompileerd van zodra je de het programma laat draaien.



Het compileren van het programma kan je volgen in de module output > Build.

A screenshot of the Visual Studio Output window titled 'Output'. It shows the build log for the 'Hello World' project. The log starts with 'Build started...', followed by 'Build started: Project: Hello World, Configuration: Debug Any CPU -----'. It then lists 'Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.' and 'Hello World -> C:\Users\Kristof\Desktop\TAB\SDA Programmeren\C#\ oefeningen\Hello World\Hello World\bin\Debug\net6.0\Hello World.dll'. The log concludes with 'Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======' and 'Build started at 10:15 and took 01,223 seconds ======'.

In het .NET-framework wordt nog een **tussenliggende, platformonafhankelijke taal** gegenereerd wanneer een C#-programma wordt gecompileerd; de intermediate language (IL) of Common Intermediate Language (CIL). Het fungeert als een tussenstap waarbij de C#-broncode wordt omgezet naar een **assemblytaal**. Vervolgens wordt de IL pas vertaald naar de native code van de computer op het moment van uitvoering. De vertaling gebeurt dus “net op tijd” door de JIT-compiler (Just In Time) op de Common Language Runtime (CLR). Deze geoptimaliseerde machinecode wordt vervolgens uitgevoerd door de processor.



IL-code is niet direct gebonden aan een specifiek besturingssysteem of hardware. Dit is een belangrijk kenmerk van het .NET-platform en stelt code in staat om op **verschillende besturingssystemen** te draaien, zoals Windows, macOS en Linux.

Codeer hier

Draai het programma hier



Over het algemeen zorgt de combinatie van C# als programmeertaal en IL als tussenliggende taal voor krachtige en flexibele toepassingen die op verschillende platformen kunnen draaien.

0.2.3 Het uitvoeren van de code

Meteen nadat de broncode gecompileerd is, wordt het programma uitgevoerd. Het resultaat krijgen we te zien in de ‘Microsoft Visual Studio Debug Console’.



Dit lijkt op de opdrachtenprompt van Windows computers omdat we ervoor gekozen hebben om te werken met een Console applicatie. Dit staat ook bekend als een Command-Line application, wat wilt zeggen dat de gebruikersinterface alleen uit tekst bestaat.

0.3 De opbouw van een programma

Onze eerste toepassing ‘Hello World’ bevat slechts enkele regels broncode. We zullen deze regels eens onder de loep nemen en verklaren wat ze betekenen.

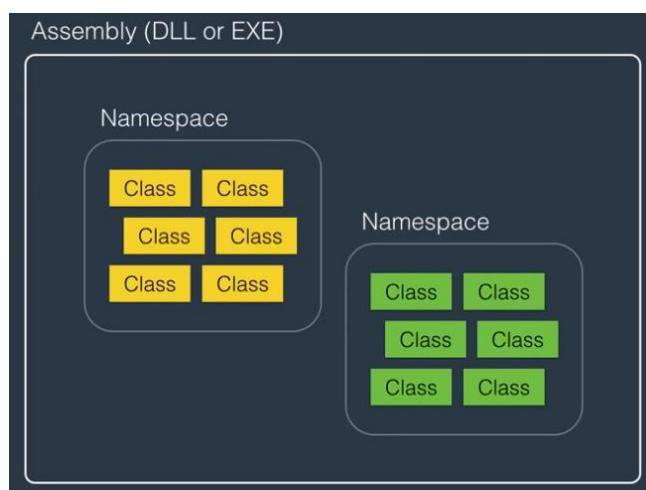
Begin niet te panikeren als je deze uitleg nu nog erg ingewikkeld vindt! Naarmate je meer vertrouwd wordt met programmeren, zal deze uitleg duidelijker worden.

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

0.3.1 Namespace

Als Visual Studio een nieuw project voor je genereert maakt het ook automatisch een standaard namespace aan waarin je eerste bestand wordt opgeslagen. In Java noemt men dit een pakket, wat het eenvoudiger maakt om te begrijpen. Code die bij elkaar hoort, wordt namelijk vaak in eenzelfde pakket, of beter gezegd namespace, gestopt.

Een namespace is dus eigenlijk een manier om namen van klassen en andere types te organiseren en te voorkomen dat er conflicten optreden tussen namen. Het helpt bij het structureren van de code en maakt het mogelijk om dezelfde namen te gebruiken voor verschillende doeleinden in verschillende delen van een programma.



Als je net begint te programmeren vraag je je misschien af waarom we namespaces nodig hebben. Waarom niet alle klassen in dezelfde namespace plaatsen zodat ze altijd allemaal toegankelijk zijn? Dat kan je doen, maar alleen zolang je aan kleine projecten werkt. Zodra je meer en meer klassen begint toe te voegen aan je programma, is het een goed idee om ze onder te verdelen in verschillende namespaces. Het maakt het makkelijker om je code terug te vinden, zeker als je je bestanden in corresponderende folders plaatst.

0.3.2 Class

C# is een objectgeoriënteerde programmeertaal. In plaats van te werken met procedures zoals C, werkt C# met objecten van een bepaalde klasse. Een object is eigenlijk een verzameling van gegevens en methoden (of functies) om met die gegevens om te gaan. Een klasse is een soort blauwdruk van een object.

Aangezien objecten en klassen nog uitvoerig aan bod zullen komen in deze cursus, gaan we er in dit hoofdstuk niet verder op in.

0.3.3 De methode Main()

De **uitvoering van het programma** start door de methode `Main()` van de applicatieklasse aan te roepen. Iedere applicatie moet dus een methode `Main()` hebben. Het is als het ware de toegangspoort (of entry point) van de toepassing. Zie het een beetje als de “magische toverstok” die ervoor zorgt dat je programma werkt. Dit klinkt misschien wat belachelijk, maar hopelijk snap je zo wel dat je programma niet kan werken zonder deze hele belangrijke methode.

De methode `Main()` is opgebouwd als volgt:

- **Static**: de methode is gemeenschappelijk voor alle objecten van deze klasse.
- **Void** (leeg): de functie geeft na de uitvoering geen waarde terug aan diegene die ze heeft opgeroepen.
- **String[] args**: de parameter die de functie al dan niet meekrijgt als hij wordt aangeroepen. Deze parameter is een *array (reeks)* van *strings (tekenreeksen)*. De inhoud van deze strings is gelijk aan de *command-line parameters* bij het opstarten van het programma.
- **{}**: de programmastappen die door de functie `Main()` worden aangeroepen, worden gedefinieerd tussen accolades.

0.3.4 Het eigenlijke werk

Vanaf hier wordt de programmacode geschreven.

In het voorbeeld van ‘Hello World’ gaat het slechts om één regel code, namelijk:

```
System.Console.WriteLine("Hello World");
```

In deze regel wordt de methode `WriteLine()` aangeroepen van het object `Console` dat hoort tot het object `System`. Objecten die horen tot andere objecten duidt men aan door de naam van het bezittende object te nemen gevuld door een punt en vervolgens de naam van het object.

Tegenwoordig hoeft je `System` niet meer te schrijven omdat op de achtergrond er al een referentie naar wordt gemaakt.

```
1 // <auto-generated>
2 global using global::System;
3 global using global::System.Collections.Generic;
4 global using global::System.IO;
5 global using global::System.Linq;
6 global using global::System.Net.Http;
7 global using global::System.Threading;
8 global using global::System.Threading.Tasks;
9
```

1.1 Programmeeralgoritmes

1.1.1 Wat is een algoritme?

In tegenstelling tot het programma ‘Hello World’, bestaan de meeste programma’s uit **meerdere instructies** die elkaar opvolgen. De stappen worden één voor één (sequentieel) uitgevoerd **in de volgorde** waarin ze zijn ingegeven. Zo’n stappenplan noemt men een **programmeeralgoritme**.

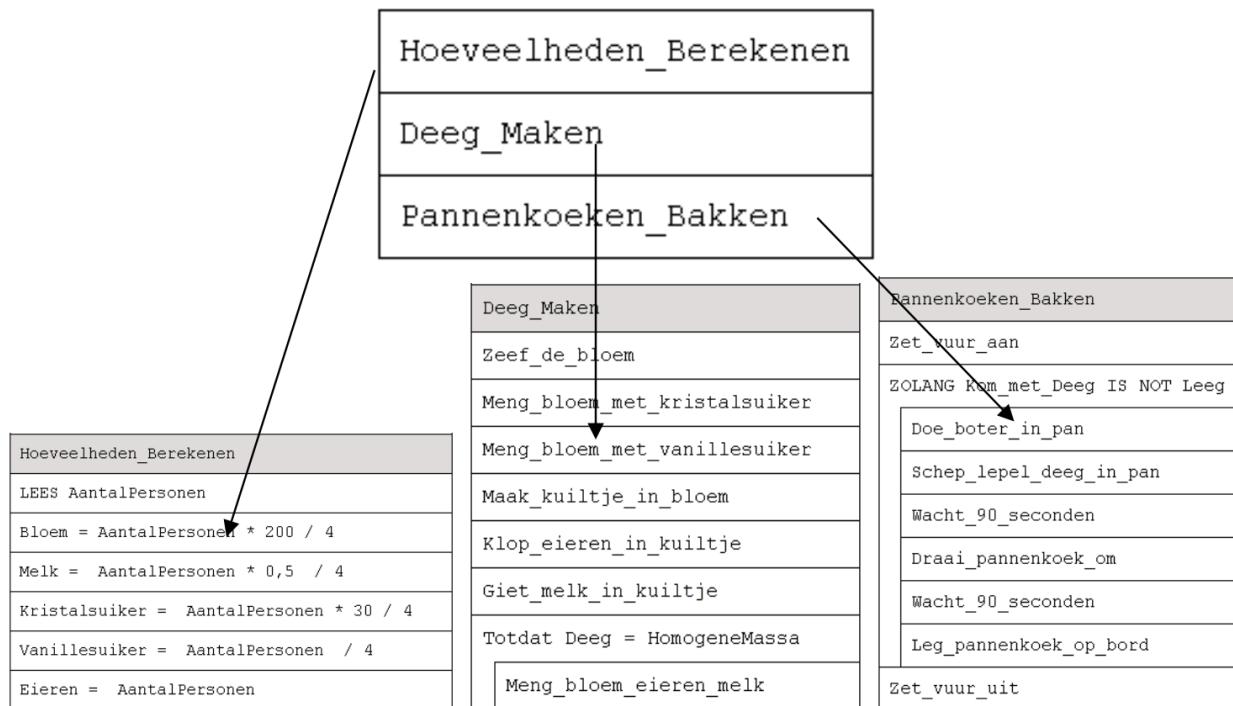
We illustreren een algoritme aan de hand van een robot die een pizza moet bestellen:

1. Neem menu;
2. Lees menu;
3. Bel Pizzaphone;
4. Geef keuze;
5. Geef adres;
6. Wacht tot bel gaat;
7. Open deur;
8. Geef geld;
9. Neem pizza aan.



Dit is natuurlijk een simpel voorbeeld met enkelvoudige, duidelijk geformuleerde stappen. Soms kunnen enkele stappen omgedraaid worden zonder dat dit invloed heeft op het eindresultaat, al is dit zeker niet altijd het geval!

Verder kan het ook nodig zijn om de stappen zodanig op te splitsen totdat de computer de basisinstructies begrijpt. Kijk maar eens naar het algoritme om een pannenkoekenrobot te programmeren (weergegeven in een NS-diagram):

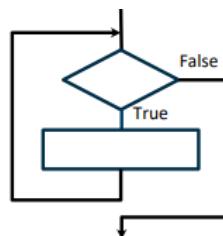
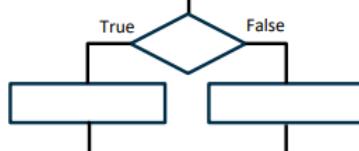
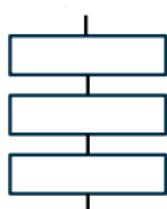


Zoals je ziet is hier hoofdalgoritme opgesplitst in deelalgoritmes die elk een deelprobleem voor hun rekening nemen.

1.1.2 Basisinstructies

Een algoritme kan bestaan uit één of meerdere soorten basisinstructies:

Sequentie	Selectie	Iteratie
een opeenvolging van verschillende instructies in de volgorde waarin het is neergeschreven	de volgende instructie is afhankelijk van een logische uitdrukking. Dit wilt zeggen dat het programma een voorwaarde stelt die meerdere uitkomstmogelijkheden heeft. Aan de hand van het resultaat van de voorwaarde zal het algoritme de daaropvolgende instructie kiezen.	een herhaling die afhankelijk is van een logische uitdrukking. Ook hier zal het programma een reeks instructies blijven herhalen aan de hand van een voorwaarde. De herhaling stopt pas wanneer de voorwaarde niet langer true (waar) is of wanneer het opgegeven aantal herhalingen is bereikt.



1.2 Probleemoplossen denken

1.2.1 Van probleem tot computerprogramma

Om een goed programma te ontwikkelen, volg je best de volgende stappen:

1. Probleemstelling	<ul style="list-style-type: none">Probeer het probleem zo goed mogelijk te begrijpen.Formuleer in je eigen woorden wat de gebruiker wil.
2. Probleemanalyse	<ul style="list-style-type: none">Deel het probleem op in kleine stukken. (TOP DOWN methode)Bepaal de invoer, de verwerking en de uitvoering.
3. Algoritme	<ul style="list-style-type: none">Ontwerp de verschillende stappen die leiden tot de oplossing.Wij gebruiken structogrammen om het algoritme voor te stellen.
4. Programma	<ul style="list-style-type: none">Vertaal het algoritme naar een computerprogramma.Ontwerp de gebruikersinterface en schrijf de programmacode.
5. Testen	<ul style="list-style-type: none">Test het programma verschillende keren uit. Gebruik steeds andere invoergegevens en kijk na of het programma blijft werken.
6. Documenteren	<ul style="list-style-type: none">Voorzie de programmacode van extra informatie.// voor 1 regel en /* */ voor meerdere regels commentaar.

Eigenlijk is er nog stap 7, namelijk het publiceren van het programma. Hiermee maak je een echt programma van de programmacode dat je kan gebruiken zonder gebruik te maken van een IDE. Deze stap zal aan bod komen wanneer je grafische applicaties leert maken.

1.2.2 Voorbeeld van probleemoplossend denken en programmeren



Opdracht:

Schrijf een programma waarbij we 2 getallen kunnen ingegeven. Het programma berekent de som en toont deze op het scherm.

1. Probleemstelling: is het probleem duidelijk?

We moeten een programma maken waarbij:

- twee getallen kunnen worden ingegeven;
- de som van de ingevoerde getallen wordt berekend;
- de som op het scherm wordt getoond.

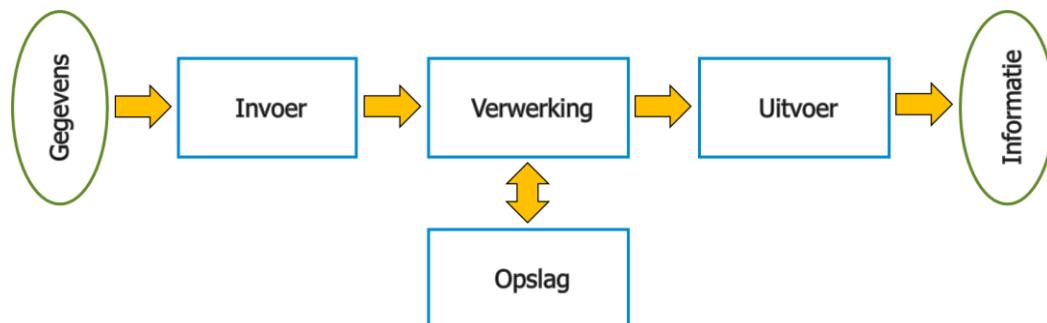
Visueel voorgesteld zal het programma er als volgt uitzien:

```
Geef het eerste getal in: 15  
Geef het tweede getal in: 35  
Dit is de som: 50
```

2. Probleemanalyse: wat is gegeven en wat is nog nodig?

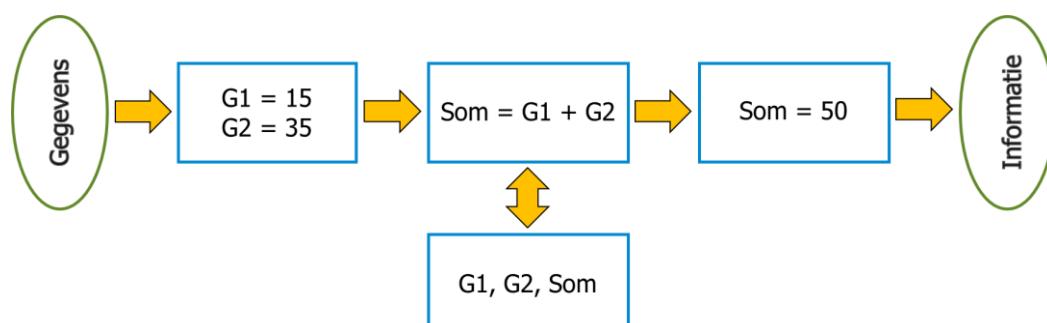
Het is een vrij eenvoudig probleem. Er is dus geen opsplitsing in deelproblemen nodig.

Voor beginners kan het helpen om kleine programmeeropdrachten te bekijken vanuit het **gegevensverwerkend proces**:



- **Invoer:** twee getallen lezen ($g1$ en $g2$)
- **Verwerking:** de som van twee getallen ($som = g1 + g2$)
- **Uitvoer:** som tonen ($som = ?$)
- **Opslag:** de ingegeven getallen en de uitkomst van de som ($g1, g2, som$)

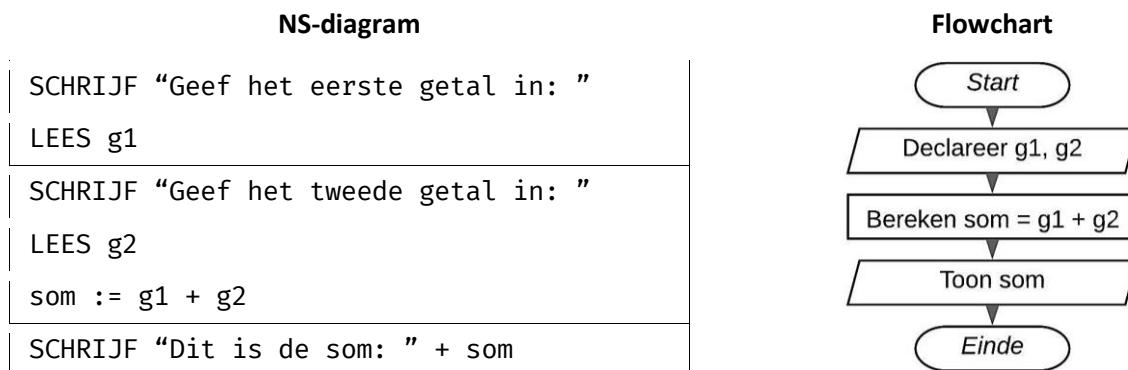
Een ingevuld schema ziet er dan uit als volgt:



3. Algoritme: welke oplossingsmethode hanteren?

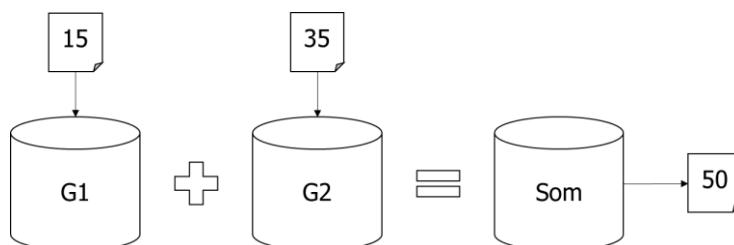
We kunnen het **algoritme schematisch weergeven** met behulp van een NS-diagram (Nassi-Schneidermann diagram) of een flowchart. In deze cursus zullen algoritme voornamelijk in een NS-diagram worden weergegeven. Hieronder vatten we kort samen hoe je het schema moet gebruiken en interpreteren:

- Elke rij bevat één instructie.
- Het woord **LEES** geeft aan welke gegevens (variabelen) moet worden ingelezen (= **input**).
- Het woord **SCHRIJF** geeft aan wat op het scherm moet worden getoond (= **output**).
- Voor de verwerking van een opdracht gebruik je `:=` als het om een toekenning gaat (bv. `som := getal1 + getal2`).



Visuele voorstelling:

- **Invoer:** We voeren de 2 getallen in.
- **Opslag:** de getallen worden opgeslagen in het geheugen (variabelen g1 en g2).
- **Verwerking:** de 2 getallen worden opgeteld en opgeslagen in de variabele som.
- **Uitvoer:** de inhoud van de som wordt getoond.



4. Programma: per gebeurtenis de noodzakelijke broncode programmeren

In de volgende stap wordt het **algoritme omgezet naar programmacode**. In dit geval schrijven we de broncode in de programmeertaal C#. De eigenlijke programma code ziet eruit als volgt:

```
Console.WriteLine("Geef het eerste getal in: ");
int g1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Geef het tweede getal in: ");
int g2 = Convert.ToInt32(Console.ReadLine());
int som = g1 + g2;
Console.WriteLine("Dit is de som: " + som);
```

Deze code werkt, maar is niet overzichtelijk. Laten we de code een beetje opkuisen en toelichten ...

Als je de code vergelijkt met het NS-diagram of de flowchart, zal je merken dat er weinig verschillen zijn. Maar hoe komen we er zelf op? Een **verklaring van de programmacode** zal veel duidelijk maken:

Algemene syntaxregels:

- een instructie afgesloten met een puntkomma ;
- Codeblokken worden afgebakend met accolades { }
- Teksten (= strings) worden afgebakend met dubbele aanhalingsstekens “ ”

Declaratie van variabelen:

Je reserveert geheugenplaatsen voor de invoer, berekeningen en uitvoer. In dit geval: g1, g2 en som. Je doet dit door eerst het **gegevenstype** te vermelden en vervolgens kies je een **gepaste naam** voor de variabele. In dit hoofdstuk beperken we ons tot gegevenstypes voor:

- gehele getallen (integers): `int`
- tekst: `string`

```
int g1;
int g2;
int som;
```

Invoer van gegevens:

De invoer gebeurt door gegevens te **lezen** in een variabele die overeenstemt met het gegevenstype. Je leest teksten dus in een string variabele en getallen in een integer variabele. Dit noemen we een **toekenning**.

Elke waarde die je inleest is in C# automatisch een tekenreeks of string. We zijn helaas dus genoodzaakt om aan het programma te laten weten dat we numerieke tekenreeksen willen converteren naar een numerieke waarde.

- Tekst: `Console.ReadLine();`
- Gehele getallen: `Convert.ToInt32(Console.ReadLine());`

```
Console.WriteLine("Geef het eerste getal in: ");
int g1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Geef het tweede getal in: ");
int g2 = Convert.ToInt32(Console.ReadLine());
```

Verwerking:

In dit voorbeeld kan je de eigenlijke berekening uitvoeren door de variabelen g1 en g2 met elkaar op te tellen en toe te kennen aan de variabele som.

In een toekenningsopdracht wordt de **waarde van het rechterlid toegekend aan de waarde van het linkerlid**. Een toekenning is dus niet hetzelfde als een vergelijking in wiskunde!

```
int som = g1 + g2;
```

Uitvoer van de gegevens:

Je kan tekst en variabelen op het scherm schrijven als volgt:

```
Console.WriteLine("tekst" + variabelenaam);
```

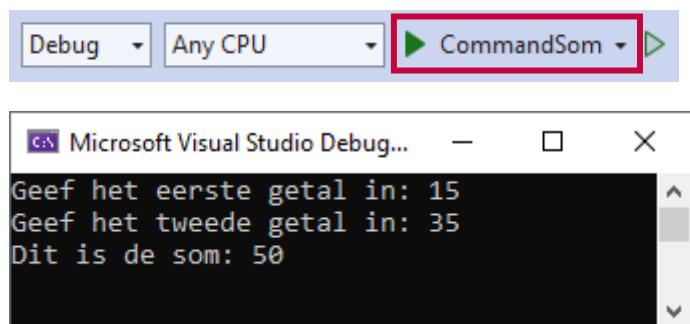
```
Console.WriteLine("Dit is de som: " + som);
```

Als we de volledige broncode schrijven in Visual Studio, zal het eruit zien als volgt:

```
1  namespace CommandSom
2  {
3      0 references
4      internal class Program
5      {
6          0 references
7          static void Main(string[] args)
8          {
9              int g1;
10             int g2;
11             int som;
12
13             Console.Write("Geef het eerste getal in: ");
14             g1 = Convert.ToInt32(Console.ReadLine());
15             Console.Write("Geef het tweede getal in: ");
16             g2 = Convert.ToInt32(Console.ReadLine());
17
18             som = g1 + g2;
19
20             Console.WriteLine("Dit is de som: " + som);
21         }
22     }
```

5. Testen: het programma testen en eventueel bijsturen.

Wanneer het programma af is, kunnen we het uitvoeren (= runnen) en testen.



Controle: werkt het programma wel in alle omstandigheden? Wat gebeurt er als we:

- positieve getallen invoeren?
- negatieve getallen invoeren?
- Niets invullen?
- Tekst invullen?
- ...

6. Documenteren: het programma voorzien van commentaar.

Tenslotte kunnen we ons programma **documenteren** door een groep code te voorzien van commentaarregels. **Commentaar** kan je naar hartenlust toevoegen aan je code omdat deze regels toch **genegeerd** worden wanneer het programma wordt uitgevoerd.

Commentaarregels kan je ingeven als volgt:

- Een enkele regel: //
- Meerdere regels: /* */

In ons programma zal dit eruit zien als volgt:

```
/* We maken een programma dat 2 getallen inleest
en daarna de som ervan berekent */

//Opslag: declaratie van de variabele gegevens
int g1;
int g2;
int som;

//Input: invoer van de getallen
Console.WriteLine("Geef het eerste getal in: ");
g1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Geef het tweede getal in: ");
g2 = Convert.ToInt32(Console.ReadLine());

//Verwerking: berekening van de som
som = g1 + g2;

//Output: toont het resultaat
Console.WriteLine("Dit is de som: " + som);
```

Voor kleine programma's zal dit niet nodig zijn. Als je echter grotere, complexe programma's maakt, zal het jezelf en andere programmeurs helpen om op een later moment wegwijs te geraken in jouw code.

1.3 Rekenen in C#

1.3.1 Elementaire berekeningen

De volgende operatoren ken je reeds vanuit de wiskunde en die kan je gebruiken om operanden met elkaar op te tellen:

Operator	Bewerking
+	Som
-	Verschil
*	Vermenigvuldiging
/	Deling

1.3.2 Complexere berekeningen met de wiskundige klasse `Math`

Voor complexere berekeningen gebruik je methoden van de klasse `Math`. Hieronder krijg je een overzicht van de meest gebruikte methoden:

Methode	Beschrijving
<code>Max(x,y)</code>	Hoogste waarde in een reeks
<code>Min(x,y)</code>	Laagste waarde in een reeks
<code>Sqrt(x)</code>	Macht berekenen
<code>Pow(x,y)</code>	Vierkantwortel berekenen
<code>Round(x)</code>	Wiskundig afronden
<code>Ceiling(x)</code>	Afronden naar boven
<code>Floor(x)</code>	Afronden naar beneden
<code>Abs(x)</code>	Waarde geven zonder toestandsteken

Dit is slechts een hulpmiddel om nu al complexere berekeningen te kunnen uitvoeren. Het gebruik van objecten en klassen zal pas later in deze cursus uitgebreid aan bod komen.

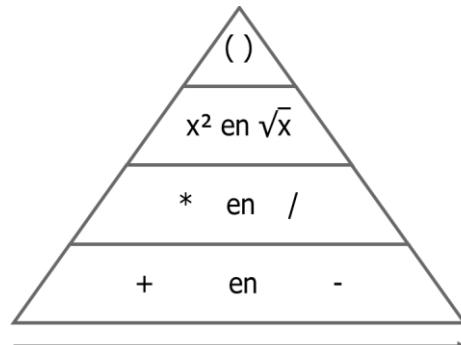
De parameters `x` en `y` stellen argumenten voor die je moet invoeren om het gewenste resultaat te bekomen.

Voorbeeld: `Math.Pow(2, 10) = 210 = 1024`

1.3.3 De volgorde van bewerkingen in C#

in C# gelden dezelfde voorrangsregels als in de wiskunde:

1. Haakjes
2. Machten en wortels
3. Vermenigvuldigen en delen
4. Optellen en aftrekken
5. Bewerkingen uitvoeren van links naar rechts



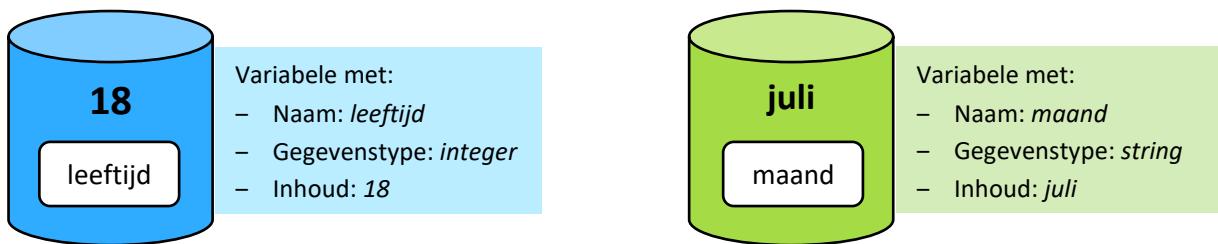
2.1 Inleiding

In het vorige hoofdstuk hebben we reeds kennis gemaakt met variabelen. We hebben toen gezien dat het vaak nodig is om gegevens te kunnen gebruiken én bij te houden in het geheugen van de computer, voor de uitvoering van een programma. Denk bijvoorbeeld maar aan de invoer van getallen of het resultaat van een berekening.

Deze gegevens kunnen veranderen tijdens de werking van het programma. Om deze **veranderlijke gegevens** tijdelijk op te slaan, gebruik je variabelen.

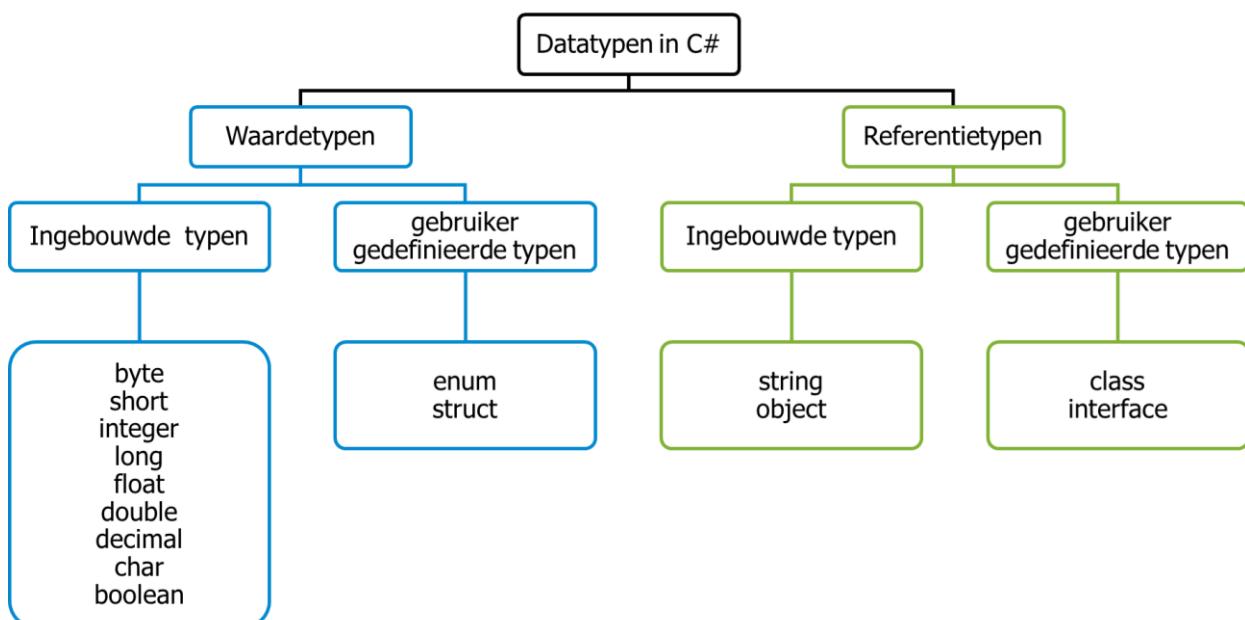
Een variabele is dus een stukje **geheugen** met een unieke naam waarin je gegevens kunt stoppen. Een variabele is opgebouwd met de volgende kenmerken:

- **Gegevenstype (Data type)**: dit bepaalt welk soort gegevens erin kunnen worden opgeslagen zoals gehele getallen, tekenreeksen, booleaanse waarden, ...
- **Naam (Identifier)**: iedere variabele heeft een unieke naam die bestaat uit letters en cijfers.
- **Bereik (Scope)**: het gebied binnen de programmacode waarin je de variabele kan gebruiken.
- **Inhoud**: de data (gegevens) die in een variabele worden opgeslagen.



2.2 Gegevenstypen

In C# kan men gegevenstypen onderverdeelen in twee categorieën: **waardetypen** en **referentietypen**.



2.2.1 Waardetypen

Waardetypen (ook bekend als primitieve typen) zijn de meest fundamentele gegevenstypen die in programmeertalen worden gebruikt. Ze vertegenwoordigen elks een **feitelijke, enkele waarde** en hebben geen onderliggende structuur.

In de onderstaande tabel staan de meest gebruikte gegevenstypen in C#. De werkelijke grootte van gegevenstypen kan variëren op verschillende platformen, maar deze waarden zijn representatief voor de meeste moderne systemen.

C# type	.NET type	Omschrijving	Formaat	Minimaal	Maximaal
<i>Gehele getallen</i>					
byte	Byte	Byte waarde (2^8 bits)	1 byte	0	255
short	Int16	Korte gehele getallen	2 bytes	-32.768	32.767
int	Int32	Integer	4 bytes	-2.147.483.648	2.147.483.647
long	Int64	Lange gehele getallen	8 bytes	$-9,2 \cdot 10^{17}$	$9,2 \cdot 10^{17}$
<i>Reële getallen</i>					
float	Single	Kommagetallen met enkele precisie	4 bytes	$-3,4 \cdot 10^{38}$	$3,4 \cdot 10^{38}$
double	Double	Kommagetallen met dubbele precisie	8 bytes	$-1,7 \cdot 10^{308}$	$1,7 \cdot 10^{308}$
decimal	Decimal	Getallen met hoge precisie	16 bytes	$-7,9 \cdot 10^{28}$	$7,9 \cdot 10^{28}$
<i>Andere types</i>					
bool	Boolean	Booleanse waarde	1 bit	False	True
char	Char	Een teken of karakter	16 bit Unicode	\u0000	\uFFFF

Er bestaan nog heel wat andere waardetypen zoals sbyte, ushort, uint en ulong, maar die zal je in de praktijk zelden of zelfs nooit nodig hebben om een programma te schrijven.

2.2.2 Referentietypen

Een referentietype is een **verwijzing** of referentie naar een **object** in plaats van de werkelijke gegevens zelf. De variabelen zijn m.a.w. verwijzingen naar geheugenlocaties waarin de waarden worden opgeslagen. Dit betekent dat meerdere variabelen naar hetzelfde object kunnen verwijzen.

Voorbeelden hiervan zijn strings, arrays, classes, interfaces, delegates, ...

In de volgende code wordt een nieuw object gecreëerd van de klasse Persoon. De variabele persoon1 is een verwijzing naar dit object. De inhoud van deze variabele is het adres van het object.

```
Persoon eenPersoon = new Persoon();
```

Een referentiemerkwaarde die naar geen enkel object verwijst, heeft de voorgedefinieerde waarde null.

i Referentietypen zullen verder in deze cursus uitvoerig besproken worden omdat ze de basis vormen voor object georiënteerd programmeren. Voorlopig werk je alleen met waardetypen en strings.

2.2.2.1 Het referentietype string

De klasse `String` is een speciale klasse omdat het eigenlijk een referentietype is, maar intern is het zodanig geoptimaliseerd dat het behandeld kan worden als een waardetype.

```
String tekst1 = new String("voorbeeld");    //niet nodig  
string tekst2 = "voorbeeld";                  //dit volstaat
```

Het is dus niet nodig om expliciet een instantie van de klasse `String` te maken zoals je zou doen met een aangepaste klasse. Je kunt simpelweg een string variabele declareren en initialiseren met een waarde.

2.3 De naamgeving

Iedere variabele heeft een naam (of identifier) die moet voldoen aan de volgende **voorwaarden**:

- De naam bestaat uit een reeks Unicodekarakters, beginnend met een letter, underscore (_) of een dollarteken (\$).
- De naam mag niet beginnen met een cijfer.
- De naam mag geen gereserveerd woord zijn.
- De naam moet uniek zijn binnen zijn bereik (scope). Men mag dus geen twee variabelen met dezelfde naam geven binnen hetzelfde bereik.

Tenslotte willen we de nadruk leggen op het kiezen van **duidelijke en zinvolle namen!** Gebruik geen afkortingen, maar schrijf liefst de namen voluit. Jij weet op het moment zelf wat de variabele betekent, maar voor anderen (en op een later moment ook jzelf) kan het onduidelijk zijn wat een variabele juist represeneert. Kies dus voor namen als `huidigeMeterstand` in plaats van `hms`.

2.3.1 Naming conventions

Naming conventions zijn **richtlijnen** die worden gevuld bij het geven van namen aan variabelen, functies, klassen en andere elementen in een programma. Daardoor schrijf je overzichtelijke en leesbare code. Zo kunnen ontwikkelaars elkaars code ook beter begrijpen en onderhouden.

In C# zal je voornamelijk gebruik maken de volgende notaties: **camelCase** en **PascalCase**

2.3.1.1 camelCase

De camelCase-notatie (of kameelnotatie) wordt gebruikt om **lokale variabelen** te benoemen.

De eerste letter van de naam wordt geschreven in kleine letters. Als de naam uit meer woorden bestaat, begint de eerste letter van elk nieuw woord met een hoofdletter. Er zijn geen spaties of leestekens tussen de woorden.

```
int grootsteGemeneDeler;  
double eenKleinGetal;
```

2.3.1.2 PascalCase

De PascalCase (of Pascalnotatie) wordt gebruikt om **constanten**, **functies** of **klassen** te benoemen.

Het lijkt op camelCase, maar hier beginnen alle woorden met een hoofdletter. Ook hier zijn er geen spaties of leestekens tussen de woorden.

```
const int KleinsteGemeenschappelijkVeelvoud;
public class OppervlakteBerekenen;
```

2.3.2 Gereserveerde namen

De woorden in de onderstaande tabel hebben reeds een speciale betekenis voor de compiler in C# en kunnen dus niet als naam voor een variabele gebruikt worden.

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

2.4 Het bereik

Het bereik (of scope) van de variabele is het gedeelte van de code waarbinnen die variabele geldig en toegankelijk is. Anders gezegd is dit het gebied van het programma waarbinnen de variabele kan worden gebruikt. Daarbuiten heeft de variabele geen invloed.

Voorbeeld van het bereik van een **lokale variabele**:

```
{
    byte a = 1;
    {
        byte b = 2;
        {
            byte c = 3;
        }
    }
}
```

In dit voorbeeld is de variabele b overal toegankelijk binnen het gearceerde codeblok en zijn subcodeblokken. Als je b buiten dit codeblok wilt gebruiken, krijg je echter een Compile-Time error. Je kan variabele b dus alleen gebruiken met de variabele a als je binnen dit bereik blijft.

Het is belangrijk om te begrijpen waar en wanneer variabelen geldig zijn om correcte en efficiënte code te schrijven!

 Er bestaan verschillende soorten variabelen met verschillende reikwijdtes, maar die leer je gaandeweg wel kennen. Voorlopig beperken we ons tot de lokale variabelen.

2.5 Het declareren en initialiseren van variabelen

2.5.1 Variabelen declareren

Voordat we een variabele kunnen gebruiken moet het eerst **gedeclareerd** worden. Dit doen we door eerst het gegevenstype te typen, gevolgd door de naam: <type> <naam>;

```
int leeftijd;
```

Meerdere variabelen van hetzelfde gegevenstype kunnen op één regel gedeclareerd worden door de namen te scheiden met een komma. Dit is echter af te raden!

```
int getal1, getal2, getal3;
```

 Declareer elke variabele altijd op een aparte regel.

2.5.2 Variabelen initialiseren

Initialiseren is het toekennen van een waarde aan een variabele. De letterlijke waarden die men toekent aan een variabele, noemt men **literals**. Een variabele kan op verschillende manieren geïnitialiseerd worden:

- Tijdens de declaratie een waarde toekennen aan een variabele.

```
int getal = 18;
```

- Na de declaratie een waarde toekennen aan een variabele.

```
int getal;  
getal = 18;
```

- De invoer van de gebruiker toekennen aan een variabele.

```
int getal = Convert.ToInt32(Console.ReadLine());
```

- Het resultaat van een bewerking toekennen aan een variabele.

```
int x = 7;  
int y = 10;  
int resultaat = x + y;
```

2.5.3 Literals voor waardetypen

Afhankelijk van het gebruikte gegevenstype zal je een **literal** volgens een **specifieke notatie** moeten neerschrijven.

Datatype	Notatie	Voorbeeld
byte short int	Typ gewoon het getal.	<code>int getal = 42;</code>
long	Plaats een l of L achter het getal.	<code>long getal = 1247418127475L;</code>
float	Plaats een f of F achter het getal.	<code>float afstand = 3.25F;</code>
double	Typ de komma met een punt of gebruik de wetenschappelijke notatie.	<code>double gemiddelde = 13.25;</code> <code>double fractie = 1.0E-8;</code>
decimal	Plaats een m achter het getal.	<code>decimal precisie = 0.51231m;</code>
char	Tussen enkele aanhalingstekens of de overeenstemmende code uit de ASCII-tabel .	<code>char letter = 'A';</code> <code>char letter = 65;</code>
bool	Typ true of false.	<code>bool controle = true;</code> <code>bool lichtAan = false;</code>
string	Tussen dubbele aanhalingstekens " ".	<code>string naam = "Joske";</code>

2.5.3.1 Karakter literals

C# werkt met Unicode (UTF-8) waarvan ASCII een subset is. Op deze manier kan je speciale tekens met het datatype char toevoegen aan strings met behulp van karaktercodes.

```
char letter = (char) 65; //Deze code komt overeen met het teken A
```

Indien het karakter niet beschikbaar is op het toetsenbord, kan je gebruik maken van de Unicode-notatie. Dit is een 16 bits-getal, geschreven als een hexadecimaal getal (16-tallig talstelsel: 0-9A-F).

```
char letter = '\u0041'; //Deze code komt overeen met het teken A
```

Indien je tekens uit de ASCII-2 uitbreidingsset wilt gebruiken zoals een €, %, ... kan dit een ?-teken geven als output in Visual Studio. Dit kan te maken hebben met de karaktercodering die wordt gebruikt in je console. De console moet dus correct geconfigureerd zijn om Unicode-karakters weer te geven.

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.UTF8;      //Uitbreiding tekenset
    char euroTeken = '€';
    Console.WriteLine(euroTeken);
}
```

Naast de unicode-notatie zijn er nog andere escape-codes voor speciale karakters. Deze neem je gewoonlijk op in een tekenreeks (**string**) met behulp van het **backslash escape character (\)**.

Escape code	Resultaat	Betekenis
\'	'	Enkelvoudig aanhalingsteken
\"	"	Dubbel aanhalingsteken
\\"	\	Backslash
\n	↵	Nieuwe regel
\t	→	Tabulator
\b	←	Backspace
\'	'	Enkelvoudig aanhalingsteken
\"	"	Dubbel aanhalingsteken

De tekens worden gebruikt bij het afdrukken van karakters op het scherm of op een printer.

```
string vb1 = "Wij zijn \"programmeurs\";           //Wij zijn "programmeurs".  
string vb2 = "\'s morgens";                         //'s morgens  
string vb3 = "Welcome to \nC#!";                   //Welcome to  
                                                C#!
```

2.5.4 Constanten

Constanten zijn variabelen waarvan men de inhoud niet meer kan wijzigen van zodra die is toegekend. Dit is vooral handig wanneer je een bepaalde waarde heel vaak moet gebruiken in eenzelfde programmacode.

Een constante wordt gedeclareerd door **const** voor het gegevenstype te schrijven.

```
const int MaxTemperatuur = 100;  
const double Pi = 3.141_592_653;
```

Net als bij veranderlijke variabelen kan de initialisatie van constanten tijdens de declaratie gebeuren of erna. Als je achteraf toch probeert om de waarde te veranderen, krijg je een foutmelding van de compiler.

2.5.5 Overloopfouten

Een overloopfout (of overflow error) treedt op wanneer een waarde die wordt opgeslagen in een gegevenstype, te groot is om door dat type te worden vertegenwoordigd.

```
byte nummer = 255;  
nummer = nummer + 1;    //Overflow error
```

In het voorbeeld wordt een overflow error veroorzaakt omdat de variabele nummer al het maximum van het gegevenstype byte heeft bereikt.

Om overloopfouten te **voorkomen**, kun je verschillende maatregelen nemen:

- Een groter gegevenstype gebruiken.
- De invoerwaarden controleren om ervoor te zorgen dat ze binnen het bereik vallen
- Het gebruik van typeconversie om met dergelijke situaties om te gaan.

2.5.6 Typeconversie

Gegevens van het ene gegevenstype kunnen worden omgezet naar een ander gegevenstype. Dit noemt men typeconversie (of type casting). Dit kan handig zijn wanneer je bijvoorbeeld gegevens van het ene type wilt gebruiken in een context waarin een ander type vereist is.

Er zijn twee soorten typeconversies in C#: **implicit casten** en **explicit casten**

2.5.6.1 Implicite typeconversie

Dit zijn **automatische conversies** die worden uitgevoerd door de compiler omdat er geen gegevens verloren kunnen gaan bij de conversie van een **kleiner type naar een groter type**.

Een voorbeeld hiervan is het toewijzen van een `int` aan een `double`. Omdat een `double` een breder bereik heeft dan een `int`, is dit een veilige conversie.

```
int aantal = 10;  
double bedrag = aantal;
```

2.5.6.2 Expliciete typeconversie

Dit zijn **handmatige conversies** waarbij de programmeur expliciet aangeeft dat een waarde van een **groter type** moet worden omgezet **naar een kleiner type**.

Een voorbeeld hiervan is het toewijzen van een `double` aan een `int`. Omdat een `double` een waarde bevat die niet in een `int` past, zal de conversie **niet automatisch** gebeuren. Er bestaat immers het **risico op gegevensverlies**.

```
double bedrag = 10.5;  
int aantal = bedrag; // De compiler zal een foutmelding geven
```

Om deze conversie wel te forceren, kan een expliciete typeconversie worden uitgevoerd door het gewenste type tussen haakjes voor de waarde te plaatsen.

```
double bedrag = 10.5;  
int aantal = (int) bedrag; //De waarde van aantal wordt 10
```



Opgelet, je mag geen conversie doen tussen een `bool` en een ander waardetype!

Operatoren zijn bewerkingstekens (symbolen) waarmee je bewerkingen kan uitvoeren op operanden (gegevens). Een operand kan een variabele, constante of object zijn. Operatoren zijn dus de bouwstenen waarmee je code wordt opgebouwd.

Operand1 **operator** operand2 \Rightarrow Bijvoorbeeld: $1 + 2$

We kunnen operatoren onderverdelen volgens het soort bewerking dat ze uitvoeren:

- Rekenkundige operatoren
- Incrementeer- en decrementeer operatoren
- Toekenningsoperatoren
- Vergelijkingsoperatoren
- Logische operatoren
- Overige operatoren

3.1 Rekenkundige operatoren

Rekenkundige operatoren (of Arithmetic Operators) dienen om wiskundige bewerkingen uit te voeren op numerieke waarden.

Operator	Bewerking
+	Som
-	Verschil
*	Vermenigvuldiging
/	Deling
%	Rest na deling

Voorbeelden:

```
int a = 10;
int b = 3;
int som = a + b;           // som = 13
int verschil = a - b;      // verschil = 7
int product = a * b;       // product = 30
int quotiënt = a / b;      // quotiënt = 3
int rest = a % b;          // rest = 1
```

3.2 Incrementeer- en decrementeeroperatoren

de increment (++) en decrement (--) operatoren worden gebruikt om de waarde van een variabele met één te verhogen (incrementeeren) of te verlagen (decrementeeren).

Operator	Bewerking	Hoe gebruiken?	Volledige schrijfwijze
++	verhoog met 1	i++;	i = i + 1;
--	verminder met 1	i--;	i = i - 1;

Voorbeelden:

```
int x = 10;
int y = 10;
x++;           // Equivalent aan: x = x + 1; (x = 11)
y--;           // Equivalent aan: y = y - 1; (y = 9)
```

3.3 Toekenningsoperatoren

Er is slechts één toekenningsoperator '=' die de waarde van de tweede operand toekent aan de eerste operand.

Operator	Hoe gebruiken?	Resultaat
=	x = y	x krijgt de waarde van y toegekend.
=	x = y + z	Het resultaat van de bewerking (y + z) is de nieuwe waarde van x.

Voorbeelden:

```
int x = 5;
int y = x * 10;    // y = 50
```

Deze toekenningsoperator wordt vaak gecombineerd met andere operatoren.

Operator	Hoe gebruiken?	Volledige schrijfwijze
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2

Voorbeelden:

```
int x = 10;
x += 5;      // Equivalent aan: x = x + 5; (x = 15)
x -= 3;      // Equivalent aan: x = x - 3; (x = 12)
x *= 2;      // Equivalent aan: x = x * 2; (x = 24)
x /= 4;      // Equivalent aan: x = x / 4; (x = 6)
x %= 2;      // Equivalent aan: x = x % 2; (x = 0)
```

3.4 Vergelijgingsoperatoren

Vergelijgingsoperatoren worden gebruikt om de relatie tussen twee waarden te beoordelen en een booleanse waarde (true of false) als resultaat te geven.

Ze worden regelmatig gebruikt om een voorwaarde te stellen voor een selectie of iteratie. In de volgende hoofdstukken zal je deze operatoren uitvoerig leren gebruiken.

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

Voorbeelden:

```
int x = 10;
int y = 5;
bool isGelijk = (x == y);           //false
bool isNietGelijk = (x != y);       //true
bool isGroterDan = (x > y);        //true
bool isKleinerDan = (x < y);        //false
bool isGroterOfGelijk = (x >= y);   //true
bool isKleinerOfGelijk = (x <= y);  //false
```

3.5 Logische operatoren

Logische operatoren worden gebruikt om logische bewerkingen uit te voeren op booleanse waarden. Het kan ook gebruikt worden om meerdere voorwaarden in selecties en iteraties samen te voegen om samen één logisch geheel te vormen.

Operator	Betekenis
&&	EN = de voorwaarde is true indien beide termen van de operator true zijn.
	OF = de voorwaarde is true indien minstens één van de termen van de operator true is.
!	NIET = keert de waarde van de booleaanse vergelijking om.

Voorbeelden:

```
int x = 5;
bool en = (x >= 1 && x <= 100)      //true, want x ligt tussen 1 en 100
bool of = (x < 6 || x > 10)          //true, want x is kleiner dan 6
bool niet = (!x == 5)                  //false, want x is wél gelijk aan 5
```

Mogelijke resultaten wanneer je een **EN-operator** uitvoert op twee booleanse waarden:

bool1	&&	bool2	bool1	&&	bool2
false	false	false	0	0	0
true	false	false	1	0	0
false	false	true	0	0	1
true	true	true	1	1	1

Mogelijke resultaten wanneer je een **OR-operator** uitvoert op twee booleanse waarden:

bool1		bool2	bool1		bool2
false	false	false	0	0	0
true	true	false	1	1	0
false	true	true	0	1	1
true	true	true	1	1	1

3.6 Conditionele operatoren

In het volgende hoofdstuk zal je onder andere leren werken met `if else` statements. Het is ook mogelijk om dit op één regel met een conditionele operator tussen 3 operanden te schrijven:

```
condition ? true_expression : false_expression
```

Het doet eigenlijk een beetje denken aan de opbouw van een ALS-functie in het programma Excel omdat het dezelfde structuur volgt:

```
=ALS(logische-test;waarde-als-waar;waarde-als-onwaar).
```

Operator	Hoe gebruiken?	Resultaat
? :	op1 ? op2 : op3	Als de booleanse waarde van op1 = true, dan wordt de waarde van op2 genomen, anders de waarde van op3.

Op1 moet dus steeds een voorwaarde zijn waarbij 2 of meer waarden met elkaar worden vergeleken. Bovendien moeten op2 en op3 van hetzelfde gegevenstype zijn.

Voorbeeld:

```
int score = 5;
string resultaat
resultaat = (score >= 5) ? "geslaagd" : "niet geslaagd"; //geslaagd
```

4.1 Bouwstenen van code

In de programmeertaal C# wordt onze code opgebouwd uit verschillende bouwstenen. Hierin maken we een onderscheid in uitdrukkingen, statements en codeblokken.

4.1.1 Uitdrukkingen

Een uitdrukking (of expression) is een combinatie van waarden, variabelen, operatoren en methodenoproepen die resulteren in een **enkelvoudige waarde**. Afhankelijk van de aard van de uitdrukking kan deze waarde een getal, een tekst, een booleaanse waarde, een object, ... zijn.

```
int getal = 1;
int resultaat = 5 + 3 * 2;
int getal = Math.Pow(2, 10);
string groet = "Hallo " + "wereld!";
bool isGelijk = (10 == 10);
```

4.1.2 Statements

Een statement is een enkele **instructie** die wordt uitgevoerd door het programma. Statements worden gebruikt om de ‘flow’ van een programma te sturen door taken uit te voeren zoals:

- Waarden toekenningen aan variabelen.

```
int leeftijd = 18;
```

- Berekeningen uitvoeren.

```
int x = 5;
int y = x * 10;
```

- Methoden aanroepen.

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

- Condities in selecties (keuzestructuren if, if else en switch).

```
if (score >= 5)
{
    Console.WriteLine("Geslaagd");
}
```

- Iteraties (herhalingen of lussen while, do while en for).

```
while (temperatuur < 0)
{
    Console.WriteLine("Het vriest!");
}
```

Een algoritme is dus eigenlijk niet meer dan een opeenvolging van statements.

4.1.3 Codeblokken

Een codeblok is een **groepering van een aantal statements** die samen worden uitgevoerd als één statement. De statements worden tussen accolades geplaatst. Het wordt o.a. gebruikt bij de definitie van methoden en bij programmaverloopstatements zoals selecties en iteraties.

In dit voorbeeld worden twee statements gegroepeerd in het codeblok van de methode Main().

```
static void Main(string[] args)
{
    string boodschap = "Hello world!";
    Console.WriteLine(boodschap);
}
```

Codeblokken helpen ook bij het **beheren van de scope van variabelen**, omdat variabelen die binnen een codeblok worden gedefinieerd, alleen zichtbaar en bruikbaar zijn binnen datzelfde blok en zijn geneste “sub blokken”. Kortom, codeblokken hebben invloed op het bereik (scope) van een variabele.

Tenslotte kan je de leesbaarheid van de programmacode verbeteren door de codeblokken te **identeren**. Dit wilt zeggen dat je de codeblokken laat inspringen met behulp van spaties of tabs om zo een goed overzicht te krijgen van alle codeblokken die in elkaar genest zijn.

4.2 Selectie statements

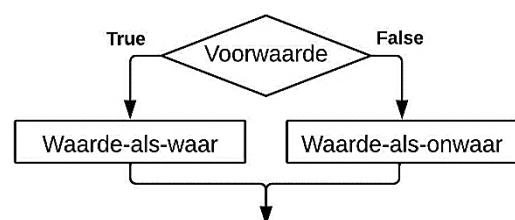
In bepaalde situaties zal een algoritme een selectie (of **keuze**) moeten maken. Het proces van het maken van beslissingen in je code gebeurt op basis van **voorwaarden**. Afhankelijk of een gestelde voorwaarde **waar** (true) of **onwaar** (false) is, zal de uitvoer van het programma verschillen.

Een voorwaarde wordt opgesteld met een uitdrukking door tenminste twee waarden met elkaar te vergelijken met behulp van een vergelijkingsoperator, of met een bool. Je kan ook meerdere voorwaarden aan elkaar koppelen met behulp van logische operatoren, met een bool als resultaat.

NS-diagram

Voorwaarde	
Waar	Onwaar
Waarde-als-waar	Waarde-als-onwaar

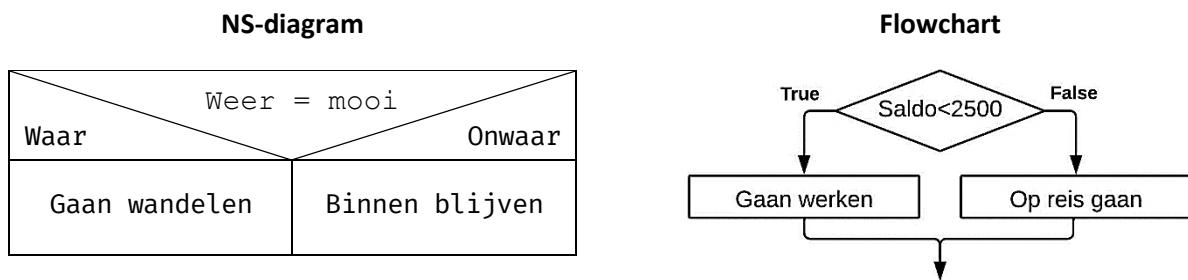
Flowchart



Het principe van selecties gebruik je regelmatig in het dagelijkse leven in de vorm van de zinsconstructie **“Als ... dan ... anders ...”**.

We illustreren dit aan de hand van enkele voorbeelden:

- “Als het mooi weer is, ga ik wandelen, anders blijf ik binnen.”
- “Als ik minder dan € 2 500,00 op mijn bankrekening heb, ga ik werken, anders ga ik op reis”.
- “Als ik tenminste 50% behaal voor dit vak, ben ik geslaagd, anders ben ik gebuisd.”



4.2.1 if statement

Het `if` statement voert alleen een statement of codeblok uit als het resultaat van de voorwaarde **waar** (true) is. Dit is een **enkelvoudige keuze**.

Syntax:

```
if (voorwaarde) {
    //statements die worden uitgevoerd als de waarde true is
}
```

Voorbeeld van een toets die op 10 punten staat:

```
if (score >= 5) {
    Console.WriteLine("Geslaagd");
}
```

Als de voorwaarde onwaar (false) is, wordt er geen actie ondernomen en gaat het programma verder naar de volgende coderegel.

4.2.2 if else statement

Met het `if else` statement kan je een verschillend statement of codeblok uitvoeren naargelang het resultaat van de voorwaarde **waar** (true) of **onwaar** (false) is. Dit is een **tweevoudige keuze**.

Syntax:

```
if (voorwaarde) {
    //statements die worden uitgevoerd als de waarde true is
}
else {
    // statements die worden uitgevoerd als de waarde false is
}
```

Voorbeeld van een toets die op 10 punten staat:

```
if (score >= 5) {
    Console.WriteLine("geslaagd");
}
else {
    Console.WriteLine("niet geslaagd");
}
```

4.2.3 `else if` statement

Als je meer dan twee keuzes wilt, kan je `if else` statements in elkaar verwerken (nesten) om voor elke voorwaarde de juiste statements uit te voeren. Dit is een geneste- of **meervoudige keuze**.

De uitvoer van een tweede `if` statement is dan afhankelijk van de evaluatie van het eerste `if` statement.

Syntax:

```
if (voorwaarde1) {  
    //statements  
} else {  
    if (voorwaarde2) {  
        //statements  
    } else {  
        //statements  
    }  
}
```

Voorbeeld van een beoordeling gebaseerd op de behaalde score:

```
if (score >= 90) {  
    Console.WriteLine("Grote onderscheiding");  
} else {  
    if (score >= 70) {  
        Console.WriteLine("Onderscheiding");  
    } else {  
        if (score >= 50) {  
            Console.WriteLine("Geslaagd op voldoende wijze");  
        } else {  
            Console.WriteLine("Niet geslaagd");  
        }  
    }  
}
```

Dit is echter minder overzichtelijk en omslachtig omdat je zo telkens diepere insprongen moet maken. Gelukkig kunnen we meervoudige selecties ook **verkort weergeven** dankzij het `else if` statement.

Syntax:

```
if (voorwaarde1) {  
    //statements  
} else if (voorwaarde2) {  
    //statements  
} else {  
    //statements  
}
```

Beter voorbeeld van een beoordeling gebaseerd op de behaalde score:

```
if (graad >= 90) {
    Console.WriteLine("Grote onderscheiding");
} else if (graad >= 70) {
    Console.WriteLine("Onderscheiding");
} else if (graad >= 50) {
    Console.WriteLine("Geslaagd op voldoende wijze");
} else {
    Console.WriteLine("Niet geslaagd");
}
```

4.2.4 switch statement

Je kan ook gebruik maken van het **switch** statement indien **meervoudige keuzes** moeten worden gemaakt op basis van een integerwaarde (char, byte, short, int) of tekenreeks (string). Dit is een handige manier om verschillende keuzes te behandelen zonder dat je opeenvolgende **if** **else** statements hoeft te schrijven. Bovendien is de code hiermee veel overzichtelijker.

Voor elke mogelijke waarde wordt er een **case** gedefinieerd met daarin een literal of constante, die van hetzelfde datatype moeten zijn als de waarde van de **switch**. Bovendien zijn duplicaten van **case**-waarden niet toegestaan. Elke **case**-waarde moet dus uniek zijn. Daarnaast moet deze waarde tijdens de compilatie van het programma gekend zijn.

Vervolgens worden per **case** de statements gedefinieerd die uitgevoerd moeten worden. Het is best dat je elke statement afsluit met een **break** statement. Het **break** statement wordt namelijk gebruikt om de **switch** te verlaten nadat een **case** is uitgevoerd. Doe je dit niet, dan zullen alle statements van de daaropvolgende **cases** worden uitgevoerd tot de eerstvolgende **break**.

Indien gewenst kan je statements achter **default** zetten voor alle waarden zonder een specifieke **case**. Deze optie is optioneel. Meerdere **default** statements zijn trouwens niet toegestaan.

Syntax:

```
switch (waarde) {
    case literal1: //statements; break;
    case literal2: //statements; break;
    ...
    default: //statements; break;
}
```

Voorbeeld van een beoordeling gebaseerd op de behaalde score:

```
switch (score) {
    case 90: Console.WriteLine("Grote onderscheiding"); break;
    case 70: Console.WriteLine("Onderscheiding"); break;
    case 50: Console.WriteLine("Geslaagd op voldoende wijze"); break;
    default: Console.WriteLine("Niet geslaagd"); break;
}
```

4.2.4.1 Fall through

De volgorde van de cases hoeven niet in oplopende volgorde te zijn, tenzij je gebruik wilt maken van ‘**fall through**’. Dit houdt in dat je doorheen de cases gaat tot aan het eerstvolgende **break** statement.

Voorbeeld van de indeling van het jaar in kwartalen:

```
int maand = 12;
switch(maand) {
    case 1:
    case 2:
    case 3: Console.WriteLine("Eerste kwartaal"); break;
    case 4:
    case 5:
    case 6: Console.WriteLine("Tweede kwartaal"); break;
    case 7:
    case 8:
    case 9: Console.WriteLine("Derde kwartaal"); break;
    case 10:
    case 11:
    case 12: Console.WriteLine("Vierde kwartaal"); break;
    default: Console.WriteLine("Ongeldige maand");
}
```

4.2.4.2 Gebruik nooit het goto statement!



Het is **bad practice** om gebruik te maken van goto omdat het kan leiden tot slecht leesbare, ongestructureerde programmaflow, ook wel gekend als “spaghetticode”. Maak er dus nooit gebruik van!

Het goto statement dient om de uitvoering van een programma over te dragen naar een ander gedeelte van de code. In een switch statement zou je zo de statements van verschillende cases kunnen afgaan.

Voorbeeld:

```
int begroeting = 2;
switch (begroeting) {
    case 1: Console.WriteLine("Hello"); goto default;
    case 2: Console.WriteLine("Bonjour"); goto case 3;
    case 3: Console.WriteLine("Hola"); goto default;
    default:
        Console.WriteLine("De ingevoerde waarde is: " + begroeting);
        break;
}
```

In dit voorbeeld zal eerst de afdruk van case 2: “Bonjour” worden getoond. Vervolgens springt de code naar case 3 en wordt “Hola” op de volgende regel afgedrukt. Tenslotte springt de code naar de default statement en wordt “De ingevoerde waarde is 2” op de derde regel afgedrukt.

5.1 Soorten iteraties

Een iteratie (of **herhaling**) is het herhaaldelijk blijven uitvoeren van een reeks instructies.

Als het aantal herhalingen afhankelijk is van een bepaalde voorwaarde, spreekt men van een **voorwaardelijke herhaling**. Hiervoor gebruik je een `while` of `do while` statement.

Voorbeeld: "Blijf aan de gebruiker vragen om een getal in te geven tot het juiste getal is geraden."

Weet je echter op voorhand precies hoeveel keer je een reeks instructies wenst te herhalen, dan spreken we van een **begrensde herhalingen**. Hiervoor gebruik je een `for` statement

Voorbeeld: "Werp twee dobbelstenen 1001 keer en toon hoeveel keer elke waarde is gegooid."

5.2 Voorwaardelijke herhaling

5.2.1 while statement

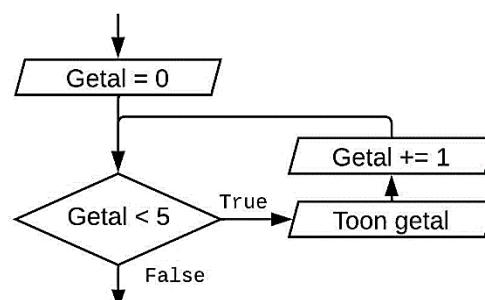
De `while` lus (of loop) herhaalt een reeks instructies zolang een opgegeven voorwaarde **waar** (true) is.

Voorbeeld: "Schrijf de waarde van een getal, beginnend bij 0, en verhoog de waarde met 1 zolang de waarde kleiner is dan 5."

NS-diagram

```
Getal := 0
ZOLANG getal < 5
    SCHRIJF getal
    getal = getal + 1
```

Flowchart



Syntax:

```
while (voorwaarde){
    //statements
}
```

Voorbeeld:

```
int getal = 0;
while (getal < 5){
    Console.WriteLine(getal);
    getal++;
}
```

5.2.2 do while statement

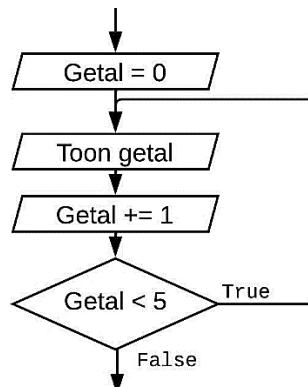
De **do while** lus is vergelijkbaar met de **while** lus, maar het garandeert dat de instructies **minstens één keer** worden **uitgevoerd**, ongeacht het resultaat van de voorwaarde. Dit komt omdat de voorwaarde pas aan het einde van de lus wordt geëvalueerd. Vervolgens blijft het de instructies herhalen zolang de opgegeven waarde **waar** (true) is.

Voorbeeld: "Schrijf de waarde van een getal, beginnend bij 0, en verhoog de waarde met 1 zolang de waarde kleiner is dan 5. Toon op z'n minst de eerste beginwaarde."

NS-diagram

Getal := 0
SCHRIJF getal
getal = getal + 1
ZOLANG getal < 5

Flowchart



Syntax:

```
do {
    //statements
} while (voorwaarde);
```

Voorbeeld:

```
int getal = 0;
do {
    Console.WriteLine(getal);
    getal++;
} while (getal < 5);
```

5.2.3 Oneindige lussen onderbreken

Bij een **voorwaardelijke herhaling** moet je oppassen dat je niet in een **oneindige lus** terechtkomt. Dit kan gebeuren als je een voorwaarde opgeeft die **nooit onwaar** (false) wordt. Dit zorgt ervoor dat het programma zonder einde nieuwe iteraties blijft uitvoeren en bijgevolg loopt het programma dus vast.

Je kan een oneindige lus onderbreken met behulp van het **break** statement.

```
while (voorwaarde1){
    //statements
    if (voorwaarde2){
        break;      // Hiermee verlaat je de lus
    }
}
```

Het programma gaat vervolgens verder na het codeblok van de iteratie.

⚠ Het gebruik van break wordt vaak gezien als een **bad practice** omdat de ervaring leert dat het kan leiden tot slecht leesbare, ongestructureerde programmaflow, ook wel gekend als "spaghetticode (hetzelfde probleem als bij goto dus).

Er zijn betere manieren om de lus te verlaten, zoals met behulp van een **voorwaarde** die op een gegeven moment waar (true) wordt. Tracht dus altijd op deze manier een lus te onderbreken.

```
while (voorwaarde1){  
    ...      //statements  
    if (voorwaarde2){  
        voorwaarde1 = false; // Zorgt ervoor dat de lus stopt  
    }  
}
```

5.2.4 continue statement

Om de huidige iteratie over te slaan, kan je gebruik maken van het **continue** statement. Het programma gaat dan verder met de controle van de voorwaarde en zal eventueel verder gaan met een volgende iteratie.

Voorbeeld:

```
for (int i = 0; i < 5; i++){  
    if (i == 2){  
        continue;  
    }  
    Console.WriteLine(i + " "); //uitvoer: 1 3 4 5  
}
```

5.3 Begrensde herhaling

5.3.1 for statement

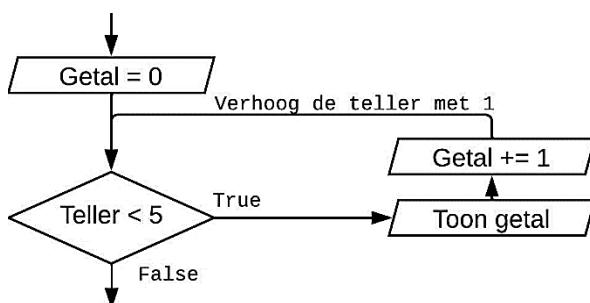
De **for** lus herhaalt de instructies het **aantal** opgegeven herhalingen.

Voorbeeld: "Schrijf 5 keer de waarde van een getal, beginnend bij 0, en verhoog de waarde bij elke herhaling met 1."

NS-diagram

Getal := 0
VOOR teller := 1 TM 5
SCHRIJF getal
getal = getal + 1

Flowchart



Syntax:

```
for (initialisatie; voorwaarde; increment) {  
    //statements  
}
```

Voorbeeld:

```
int getal = 0;  
for (int i = 0; i < 5; i++){  
    Console.WriteLine(getal);  
    getal++;  
}
```

In het begin wordt een variabele gedeclareerd en geïnitialiseerd die dienst doet als teller. Samen met de voorwaarde wordt zo het aantal iteraties van de lus bepaald. In de voorwaarde geef je namelijk de grens aan tot waar de waarde van de teller mag stijgen of dalen.

Na elke iteratie wordt de waarde van teller gewoonlijk verhoogd (increment) of verlaagd (decrement) met 1. Dit is echter niet verplicht, want je kan de teller verhogen of verlagen met eender welke waarde die je zelf wilt. Omwille van praktische redenen is dit echter af te raden.

Na elke iteratie wordt de voorwaarde geëvalueerd. De lus wordt uitgevoerd zolang de voorwaarde waar (true) is. Van zodra de conditie false is, verlaten we de lus.

Het verloop ziet eruit als volgt:



```
for (initialisatie; voorwaarde; increment) {  
    //statements  
}
```

5.3.2 foreach statement

Er bestaat nog een uitbreiding op de for-lus, namelijk de **foreach**-lus. Het foreach statement wordt gebruikt om een verzameling van elementen te itereren, zoals een array of een lijst. Het is een handige manier om elk element van de verzameling te doorlopen zonder de exacte lengte van de verzameling te kennen.

Syntax:

```
foreach (var element in verzameling){  
    //statements die iets doen met elk 'element' in de verzameling  
}
```



Het foreach statement zal je pas leren gebruiken wanneer je leert werken met arrays en is dus geen leerstof van dit hoofdstuk. Zie het als een korte kennismaking.

1.1 Soorten fouten

Naarmate programma's groter en uitgebreider worden, wordt ook de kans op fouten groter. Fouten maken is immers menselijk en het overkomt zelfs de beste programmeurs.

Fouten in een computerprogramma worden ook wel **bugs** genoemd.

Het opsporen van deze fouten en ze oplossen noemt men **debugging**.

Tijdens het programmeren of het uitvoeren van een programma kunnen verschillende soorten fouten optreden. We maken hierbij een onderscheid tussen:



- Syntax errors
- Runtime errors
- Logical errors

1.1.1 Syntaxfouten

Syntaxfouten (of compilefouten) zijn fouten die optreden **wanneer de code niet voldoet aan de grammaticale regels van de programmeertaal**. Deze fouten worden ontdekt tijdens het compileren van de code en voorkomen dat het programma wordt uitgevoerd.

Voorbeelden van syntaxfouten:

- Typ- en spelfouten
- Ontbrekende puntkomma, haakjes, accolades of aanhalingstekens
- Gereserveerde woorden gebruiken zonder correcte context
- Ongeldige toewijzingen van operatoren
- Ontbrekende komma's in functie-argumenten
- Onjuist gebruik van commentaar
- ...

Het is belangrijk om syntaxfouten te identificeren en te corrigeren voordat je probeert de code uit te voeren.

Visual Studio merkt deze fouten gewoonlijk zelf op en onderstreep ze met een rode, golvende lijn. Bovendien krijg je meer informatie over de gemaakte fout als je de cursor op de foute aanduiding plaatst. Hierdoor zijn syntaxfouten relatief eenvoudig oplossen.

```

5     static void Main(string[] args)
6     {
7         string boodschap = "Hello World"
8         int getal = tien
9
10        console.WriteLine(boodschap);
11    }
12

```

class System.String
Represents text as a sequence of UTF-16 code units.
CS1002: ; expected
Show potential fixes (Alt+Enter or Ctrl+.)

1.1.2 Runtimefouten

Een runtimefout, ook wel bekend als een uitvoeringsfout of een **uitzondering (exception)**, treedt op wanneer het programma loopt (runtime). Deze fouten worden niet gedetecteerd tijdens de compilatie, maar **treden op wanneer het programma daadwerkelijk wordt uitgevoerd**.

Voorbeelden van runtimefouten:

- Delen door nul
- Tekenreeks ingeven als een getal
- De vierkantswortel nemen van een negatief getal
- Foutieve typeconversie
- Methoden aanroepen van objecten met waarde null
- Toegang tot een niet-bestand element in een array of list
- ...

Runtimefouten kunnen variëren van relatief eenvoudig te herstellen fouten tot meer complexe problemen.

```
5 static void Main(string[] args)
6 {
7     int g1 = 5;
8     int g2 = 0;
9     int quotient = g1 / g2; ✘
10    Console.WriteLine(quotient);
11 }
12 }
13 }
```

1.1.3 Logische fouten

Logische fouten (of semantische fouten) zijn fouten in de logica van een programma die **leiden tot ongewenst of incorrect gedrag van het programma**. Zo bekomt men niet het gevraagde resultaat of volgt een programma niet de geplande weg.

Voorbeelden van logische fouten:

- Rekenfouten
- Verkeerde logische voorwaarden
- Oneindige lussen
- Verkeerd gebruik van een methode
- Fout in het algoritme
- ...

Logische fouten zijn vaak lastiger op te sporen en te corrigeren omdat ze niet leiden tot foutmeldingen in de broncode. In plaats daarvan blijft het programma gewoon draaien, maar met onverwachte resultaten.

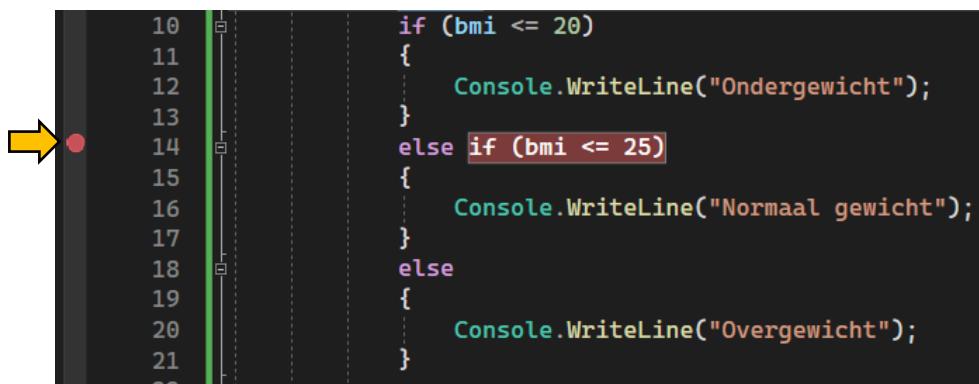
```
9      double bmi = (lengte* lengte) / gewicht;
10     if (bmi <= 20)
11     {
12         Console.WriteLine("Ondergewicht");
13     }
14     else if (bmi <= 25)
15     {
16         Console.WriteLine("Normaal gewicht");
17     }
18     else
19     {
20         Console.WriteLine("Overgewicht");
21     }
```

In dit voorbeeld zal het resultaat altijd “Ondergewicht” zijn.

1.2 Debuggen: breakpoints gebruiken

Een breakpoint is een markering dat je in je broncode plaatst om de uitvoering van je **programma tijdelijk te onderbreken** op een specifieke regel. Zo kun je variabelen controleren, de uitvoeringsvolgorde volgen en fouten opsporen tijdens het debuggen. Je plaats breakpoints dus op plaatsen **waar je problemen verwacht**. Bovendien kan je zo nagaan of het programma correct wordt uitgevoerd tot aan het breakpoint.

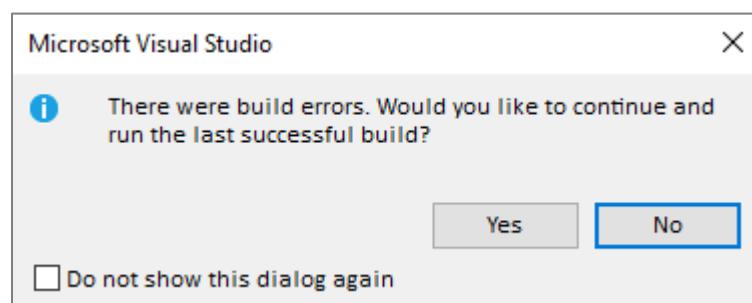
Om een breakpoint te plaatsen of terug te verwijderen, klik je in de linkse balk naast de coderegel waar je het programma wenst te onderbreken. Indien nodig kan je meerdere breakpoints plaatsen.



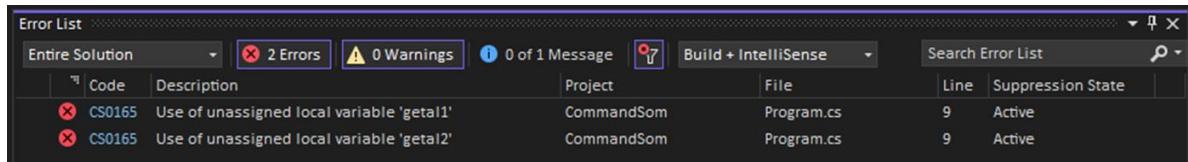
```
10     if (bmi <= 20)
11     {
12         Console.WriteLine("Ondergewicht");
13     }
14     else if (bmi <= 25)
15     {
16         Console.WriteLine("Normaal gewicht");
17     }
18     else
19     {
20         Console.WriteLine("Overgewicht");
21     }
```

1.3 Debuggen van syntaxfouten

Een programma met syntaxfouten toont een **foutmelding** wanneer je probeert het uit te voeren. Je krijgt dan de boodschap dat de toepassing niet kan worden gecompileerd omwille van ‘**build errors**’. Je kan er eventueel wel voor kiezen om het programma uit te voeren vanaf je laatste succesvolle build door op ‘Yes’ te klikken. Meestal kies je echter voor ‘No’ om het debuggen te onderbreken.



Onderaan in de **error list** staan de fouten beschreven met vermelding van de regel (line) waar de fout staat.



Code		Description	Project	File	Line	Suppression State
CS0165		Use of unassigned local variable 'getal1'	CommandSom	Program.cs	9	Active
CS0165		Use of unassigned local variable 'getal2'	CommandSom	Program.cs	9	Active

1.3.1 Stappenplan

Door de volgende stappen te volgen en geduldig door je code te gaan, kan je de meeste syntaxfouten identificeren en corrigeren:

1. Lees de foutmeldingen

Wanneer je probeert te compileren, zal Visual Studio specifieke foutmeldingen weergeven die je vertellen waar de syntaxfouten zich bevinden. Lees deze meldingen zorgvuldig om de locatie en het type van de fout te begrijpen.

2. Analyseer de gemelde regels met errors

Ga naar de regels in je code die in de foutmeldingen vermeld staan. Kijk zorgvuldig naar de syntax op die locaties.

3. Controleer puntenkomma's en haakjes

Ga na of elke regel eindigt met een puntkomma (;), en controleer of haakjes en accolades correct worden geopend en gesloten.

4. Controleer op typ- en spelfouten

Let op onjuist gespelde woorden of verkeerd gebruikte variabelennamen.

5. Controleer de schrijfwijzen van controlestructuren

Zorg ervoor dat voorwaarden, lussen en schakelaars correct zijn geschreven.

6. Gebruik de IDE-hulpmiddelen

Visual Studio beschikt over een automatische kleurcodering voor verschillende soorten code die kan helpen bij het vinden van syntaxfouten.

7. Test in kleine stappen

Als je grote stukken code hebt geschreven, probeer dan eerst een klein deel ervan te compileren. Als dat lukt, voeg dan geleidelijk meer code toe om de exacte locatie van de fout te isoleren. Dit kan je doen met behulp van breakpoints.

8. Gebruik commentaar om tijdelijk code uit te schakelen

Als je vermoedt dat een specifieke regel code een syntaxfout veroorzaakt, kun je deze tijdelijk uitschakelen door er commentaartekens (//) voor te plaatsen.

9. Raadpleeg documentatie, bronnen of (online) hulp

Als je niet zeker weet hoe je een specifiek probleem moet oplossen, zoek dan (online) naar documentatie, forums of tutorials die je kunnen helpen.

Bijvoorbeeld: Youtube, Stackoverflow, Github, ChatGPT, ...

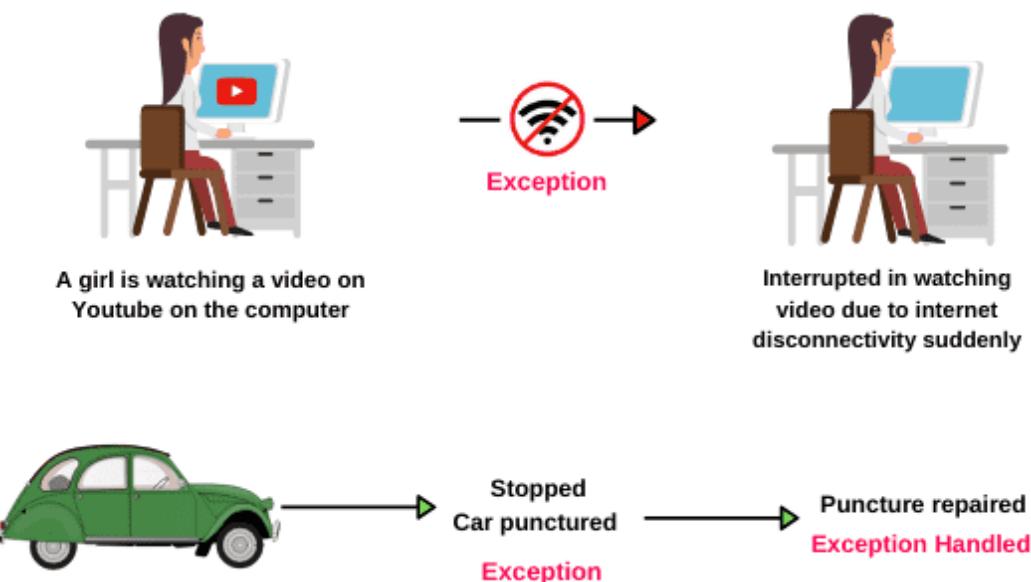
1.4 Debuggen van Runtimefouten

Indien je voor runtimefouten geen voorzieningen treft, wordt het programma afgebroken. De programmeur kan echter ervoor zorgen dat fouten die optreden onderschept en afgehandeld worden met behulp van **Exception handling**.

1.4.1 Exception handling

Exception handling (of uitzonderingsafhandeling) is een programmeerconcept dat wordt gebruikt om runtimefouten in een programma te behandelen. We noemen deze fouten uitzonderingen (of exceptions) omdat onvoorziene omstandigheden zich kunnen voordoen tijdens de uitvoering van het programma die de normale werking kunnen onderbreken.

Met exception handling kun je deze uitzonderlijke situaties opvangen en er op een gecontroleerde manier mee omgaan. Zo voorkom je dat het programma onverwacht crasht. In plaats daarvan kun je een passende reactie (of foutmelding) tonen aan de gebruiker of een alternatieve uitvoer genereren. Dit doen we met behulp van **try-catch** blokken.



1.4.2 try-catch blokken

Het **try-catch** blok bestaat uit:

- **try:** In dit codeblok plaats je de statements die eventueel een uitzondering kunnen genereren. De code wordt dus normaal uitgevoerd in het try-blok tot een runtimefout optreedt.
- **catch:** Als er een uitzondering optreedt, wordt de uitvoering van het try-blok gestopt en wordt de uitzondering doorgegeven aan het bijbehorende catch-blok. Hierin wordt de uitzondering verder afgehandeld.
- Door middel van de parameter wordt aangegeven wat voor soort uitzondering moet worden opgevangen in een specifiek catch-blok.

Aangezien er verschillende soorten fouten bestaan die opgevangen moeten kunnen worden, is het mogelijk meer catch-blokken na elkaar toe te voegen aan een try-catch blok.

Syntax:

```
try
{
    //statements die een fout kunnen opleveren
}
catch (TypeFout naamFout)
{
    //statements die uitgevoerd worden wanneer een fout optreedt
}
```

Voorbeeld:

```
try
{
    Console.WriteLine("Voer een getal in: ");
    int getal = Convert.ToInt32(Console.ReadLine());
    int resultaat = 10 / getal;
    Console.WriteLine("Het resultaat is: " + resultaat);
}
catch (FormatException)
{
    Console.WriteLine("Ongeldige invoer: voer een numerieke waarde in.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Delen door nul is onmogelijk.");
}
catch (Exception e)
{
    Console.WriteLine("Er is een fout opgetreden: " + e.Message);
}
```

In het bovenstaande voorbeeld wordt in het `try`-blok gevraagd om een getal als noemer van een deling in te geven. De volgende uitzonderingen worden opgevangen door de `catch`-blokken:

- **catch (FormatException)**: Als de gebruiker geen geldig getal ingeeft, verschijnt de foutmelding dat de invoer een getal moet zijn.
- **catch (DivideByZeroException)**: Als de gebruiker het getal nul ingeeft, verschijnt de foutmelding dat delen door nul niet mogelijk is.
- **catch (Exception e)**: Alle overige uitzonderingen worden hierin opgevangen. De gebruiker krijgt een foutmelding dat er een fout is opgetreden. Aangezien dit alle overige fouten ontvangt, moet dit blok helemaal onderaan komen.

Als geen enkele fout optreedt, wordt in het `try`-blok het resultaat van de deling getoond.

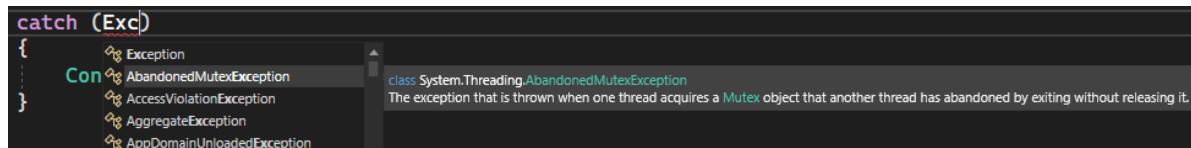
Met behulp van `naamFout.Message` kan je C# zelf een fouttype laten herkennen en een gepaste foutmelding geven. Al zijn deze foutmeldingen soms moeilijk te begrijpen voor beginnende programmeurs.

1.4.2.1 Meest voorkomende uitzonderingen

Er bestaan veel soorten uitzonderingen die je kan opnemen in catch-blokken. We sommen hier de uitzonderingen op die je het meeste zult tegenkomen in deze cursus in alfabetische volgorde:

Uitzonderingen	Beschrijving
ArgumentException	Als een ongeldig argument wordt doorgegeven aan een methode.
DividedByZeroException	Als er een poging wordt gedaan om door nul te delen.
Exception	Fouten die niet specifiek zijn afgehandeld door andere catch-blokken.
FormatException	Als er een ongeldige conversie plaatsvindt . (bv. tekst opnemen naar een numerieke variabele)
IndexOutOfRangeException	Als je toegang wilt tot een element buiten het bereik van een array of list.
InvalidOperationException	Als je een ongeldige conversie (cast) uitvoert tussen twee datatypes.
NullReferenceException	treedt op wanneer je probeert toegang te krijgen tot een object dat niet geïnitialiseerd is en bijgevolg de waarde null heeft.

Een volledige lijst van mogelijke uitzonderingen krijg je in Visual Studio te zien als je het woord 'Exception' in de parameter van een catch-blok schrijft. Als je de cursor op een exception plaatst, krijg je een beschrijving te zien zodat je weet waarvoor het dient.



1.4.3 finally blok

Een `finally`-blok kan worden toegevoegd na een `try-catch` blok. De code in dit codeblok wordt altijd uitgevoerd, ongeacht of er een fout optreedt of niet.

Syntax:

```
try
{
    //statements die een fout kunnen opleveren
}
catch (TypeFout naamFout)
{
    //statements die uitgevoerd worden wanneer een fout optreedt
}
finally
{
    //statements die uitgevoerd worden, ongeacht of er fouten optreden
}
```

Voorbeeld:

```
int teller = 10;
int noemer = 0;
int resultaat;
try
{
    resultaat = teller / noemer;
    Console.WriteLine("resultaat: " + resultaat);
}
catch (DivideByZeroException e)
{
    Console.WriteLine("Delen door nul is niet toegestaan: " + e.Message);
}
finally
{
    Console.Write("Druk op enter om het programma te beëindigen");
    Console.ReadLine();
    Environment.Exit(0); //Beëindigt het programma
}
```

-  In de praktijk zul je zelden een finally-blok tegenkomen. Het is voornamelijk toegevoegd aan de cursus voor de volledigheid.

1.4.4 Exceptions genereren met het throw statement

Het `throw` statement dient om een nieuwe, specifieke uitzondering opzettelijk te creëren en af te laten handelen door een `catch`-blok.

Syntax:

```
throw new Exception("zelfgeschreven foutmelding");
```

Voorbeeld:

```
try
{
    int leeftijd = -5;
    if (leeftijd < 0)
    {
        throw new ArgumentException("Leeftijd mag niet negatief zijn.");
    }
    Console.WriteLine("Leeftijd: " + leeftijd);
}
catch (ArgumentException e)
{
    Console.WriteLine("Fout opgetreden: " + e.Message);
}
```

In dit voorbeeld wordt een foutmelding aangemaakt voor leeftijden die worden ingegeven als een negatief getal. Deze uitzondering wordt vervolgens doorgegeven aan het catch-blok met de uitzondering ArgumentException, die de fout verder afhandelt.

1.4.5 Stappenplan

Door de volgende stappen te volgen en geduldig door je code te gaan, kan je de meeste runtimefouten identificeren en corrigeren:

1. Lees de foutmeldingen

Wanneer een runtimefout optreedt, zal Visual Studio specifieke foutmeldingen weergeven die je vertellen waar de fout zich bevindt en wat er mogelijk misgaat. Lees deze meldingen zorgvuldig om de aard van de fout te begrijpen.

2. Gebruik exception handling

Je kan try-catch-blokken gebruiken om de code op te vangen waarvan je denkt dat deze een fout kan veroorzaken. Hierdoor kan je reageren op de fout zonder dat het programma crasht.

3. Print variabelenwaarden

Voeg tijdelijke Console.WriteLine()-opdrachten toe om de waarden van variabelen te controleren op het moment dat de fout optreedt. Dit kan helpen om te begrijpen waarom de fout plaatsvindt.

4. Test in kleine stappen

Plaats breakpoints op strategische punten in je code waar je denkt dat de fout kan optreden. Hierdoor kun je het programma stap voor stap doorlopen.

5. Controleer de stappen van de uitvoering

Volg de stappen die je programma doorloopt en controleer of deze overeenkomen met wat je hebt bedoeld. Soms kan een verkeerde uitvoeringsvolgorde leiden tot fouten.

6. Test met verschillende invoerwaarden

Soms kunnen bepaalde waarden leiden tot runtimefouten. Test je programma met verschillende invoerwaarden om te zien of de fout dan nog optreedt.

7. Raadpleeg documentatie, bronnen of online hulp

Als je niet zeker weet hoe je een specifiek probleem moet oplossen, zoek dan online naar documentatie, forums of tutorials die je kunnen helpen.

Bijvoorbeeld: Youtube, Stackoverflow, Github, ChatGPT, ...

1.5 Debuggen van Logische fouten

Het debuggen van logische fouten in een programma kan uitdagender zijn dan het oplossen van syntax- en runtimefouten. Logische fouten doen zich namelijk voor wanneer de syntax van je code correct is, maar het programma niet doet wat je in gedachten hebt.

Onthoud dat het oplossen van logische fouten een vaardigheid is die verbeterd moet worden met ervaring.

1.5.1 Stappenplan

Je hebt minder hulpmiddelen om logische fouten op te lossen, maar er zijn wel stappen die je kan doorlopen om het probleem efficiënt aan te pakken:

1. Zoek het probleemgebied

Probeer het probleemgebied te bepalen waar de logische fout waarschijnlijk optreedt. Ga door je code en probeer te begrijpen wat het beoogde gedrag is.

2. Print variabelenwaarden om de inhoud te controleren

Voeg tijdelijke uitvoer toe aan je code om variabelen en tussenresultaten af te drukken. Hierdoor kun je zien wat er gebeurt tijdens de uitvoering en waar eventuele afwijkingen plaatsvinden.

3. Analyseer de logica

Analyseer de logica van je code om te controleren of het algoritme correct is. Ga de stappen door die je programma moet uitvoeren en zorg ervoor dat ze de gewenste resultaten opleveren.

4. Test in kleine stappen en isoleer het probleem

Plaats breakpoints op strategische punten in je code waar je denkt dat de fout kan optreden. Probeer het probleem te isoleren door verschillende delen van de code uit te schakelen of te testen om te zien waar het probleem zich bevindt.

5. Maak gebruik van code reviews of peer programming

Een andere ontwikkelaar kan nieuwe inzichten bieden en mogelijke logische fouten opsporen die je zelf misschien over het hoofd hebt gezien.

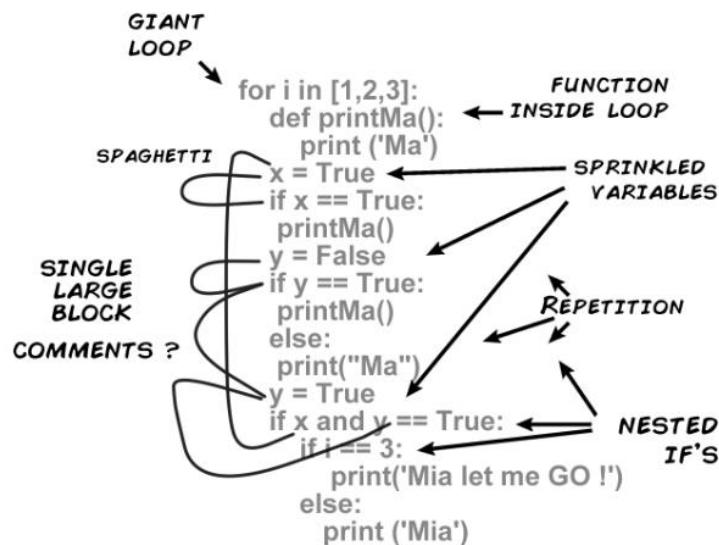
6. Documenteer wat (niet) werkt

Houd notities bij van wat je hebt geprobeerd en wat wel of niet heeft gewerkt. Dit kan nuttig zijn als je later op het probleem terugkomt.



7.1 Het belang van gestructureerde programmacode

Tot nu toe heb je leren werken met de fundamentele structuren die de basis vormen van programmacode; sequenties, selecties en iteraties. Hoewel je met deze kennis al eenvoudige programma's kan schrijven, is deze manier van werken verre van efficiënt. Naarmate je projecten groter worden zullen er namelijk al snel honderden regels code onder elkaar verschijnen. Dit leidt echter gemakkelijk tot spaghetti-code waardoor de code ongestructureerd en moeilijk onderhoudbaar wordt. Het doel is eigenlijk om het aantal regels code in de methode Main() zoveel mogelijk te minimaliseren en code hergebruik te optimaliseren.



7.2 Wat zijn methoden?

Een methode is een codeblok dat een specifieke taak uitvoert.

Het codeblok van een methode wordt gedeclareerd buiten de methode Main(). Dit kan zowel erboven als eronder, want de volgorde speelt geen rol. Dat komt omdat het programma altijd start vanuit de methode Main(). De instructies van een methode worden pas uitgevoerd als methode wordt aangeroepen vanuit de methode Main(). Met andere woorden, de methode wordt pas opgezocht wanneer het programma de instructies in dit codeblok moet uitvoeren.

Syntax:

```

//Opbouw van een methode
static returntype Methodenaam(parameters)
{
    //statements
    return waarde; //niet nodig bij returntype void
}

static void Main(string[] args)
{
    Methodenaam(parameters); //De methode wordt aanroepen
}

```

Voorbeeld 1: een methode zonder returntype (`void`) en zonder parameter

```
static void Main(string[] args)
{
    Console.WriteLine("Ongeval op de Ring van Antwerpen!");
    Console.Write("Druk op enter om verder te gaan");
    Console.ReadLine();
    Console.Clear();      //Maakt het scherm van de console leeg
    Console.WriteLine("Vermijd de R1.");
    Console.Write("Druk op enter om verder te gaan");
    Console.ReadLine();
    Console.Clear();
    Console.WriteLine("Omleiding via de E19 richting Brussel.");
    Console.Write("Druk op enter om verder te gaan");
    Console.ReadLine();
    Console.Clear();
}
```

In het bovenstaande voorbeeld moet de gebruiker telkens op enter duwen om het scherm eerst leeg te maken zodat de volgende boodschap kan worden getoond. Zoals je ziet is deze **code volledig identiek**. Dit kan dus overzichtelijker worden geschreven met behulp van een methode.

```
static void VolgendeBoodschap()
{
    Console.Write("Druk op enter om verder te gaan");
    Console.ReadLine();
    Console.Clear();
}

static void Main(string[] args)
{
    Console.WriteLine("Ongeval op de Ring van Antwerpen!");
    VolgendeBoodschap();      //aanroep van de code in de methode
    Console.WriteLine("Vermijd de R1.");
    VolgendeBoodschap();
    Console.WriteLine("Omleiding via de E19 richting Brussel.");
    VolgendeBoodschap();
}
```

Voorbeeld 2: een methode met returntype en parameter:

```
Console.Write("Geef het eerste getal in: ");
int g1 = Convert.ToInt32(Console.ReadLine());
Console.Write("Geef het tweede getal in: ");
int g2 = Convert.ToInt32(Console.ReadLine());
int som = g1 + g2;
Console.WriteLine("Dit is de som: " + som);
```

Zoals je ziet is de **code** om een getal in te lezen **quasi identiek**. Alleen de vraagstelling wijkt een beetje af, maar dat kunnen we oplossen door deze op te laten nemen als parameter van de methode.

```
static int GetalInvoeren(string vraag)
{
    Console.WriteLine(vraag);
    return Convert.ToInt32(Console.ReadLine());
}

static void Main(string[] args)
{
    int g1 = GetalInvoeren("Geef het eerste getal in: ");
    int g2 = GetalInvoeren("Geef het tweede getal in: ");
    int som = g1 + g2;
    Console.WriteLine("Dit is de som: " + som);
}
```

-  Voorlopig mag je onthouden dat een methode start met de modifier **static**. Er bestaan echter nog meer modifiers zoals **abstract** en **virtual**. De volledige werking van deze modifiers zal je gaandeweg in de cursus leren kennen.

7.2.1 Voordelen

Doordat een methode een codeblok op zichzelf is, met daarin instructies, kan je dezelfde set instructies **hergebruiken** in verschillende delen van je programma. Een bijkomend voordeel is dat de programmacode zo **overzichtelijker** en **beter onderhoudbaar** wordt.

7.2.2 Naamgeving

Net als variabelen moet een methode gedeclareerd worden met een naam. Het is juist aan de hand van deze naam dat de methode kan worden aangeroepen. In tegenstelling tot variabelen, worden de namen van methoden geschreven met de **Pascal Case** notatie. Alle woorden in de methodenaam beginnen dus met een hoofdletter en worden aan elkaar geschreven.

Vervolgens moeten achter de methodenaam altijd gesloten haakjes () staan. Binnen deze haakjes kan je parameters (of argumenten) toevoegen aan de methode. Het gebruik van parameters is echter niet verplicht.

Voorbeelden:

```
static void MijnMethode1() { }
Static int MijnMethode2(int getal1, int getal2) { }
```

7.2.3 Methoden aanroepen

Een methode kan je aanroepen in andere methoden zoals **Main()**, maar ook in andere codeblokken zoals klassen, structs, Dit doe je door de methodenaam te noteren, gevolgd met gesloten haakjes (). Die haakjes zijn erg belangrijk omdat je daarmee duidelijk aangeeft dat het om

een methode gaat. Zie het een beetje als een telefoon waarmee je communiceert met het codeblok van de methode dat wenst aan te roepen. Tussen deze haakjes kunnen parameters worden geplaatst, maar dit is niet altijd nodig. Tot slot sluit je het statement af met een puntkomma.

Voorbeeld:

```
static void Main(string[] args)
{
    HelloWorld();
}

static void HelloWorld()
{
    Console.WriteLine("Hello world!");
}
```

7.3 Procedures en functies

In sommige programmeertalen zoals VB.NET wordt gewerkt met procedures en functies. Deze zijn erg vergelijkbaar met methoden omdat beide **stukjes programmacode** zijn die je kan aanroepen vanuit het hoofdprogramma. In C# verwijzen beide termen eigenlijk naar hetzelfde, namelijk methoden.

In dit hoofdstuk bekijken we methodes vanuit het perspectief van procedures en functies. Dit maakt het onderscheid tussen het gebruik van een returntype of void veel duidelijker en zal het makkelijker maken om andere programmeertalen te leren die hierin wel een onderscheid maken.

7.3.1 Procedure

Een procedure is een codeblok die een taak uitvoert **zonder een waarde terug te geven**. Het is mogelijk om parameters op te nemen in de methode om de taak uit te voeren, maar dit wordt minder gedaan dan bij functies. Een methode met de eigenschappen van een procedure wordt altijd gedefinieerd met het returntype **void**.

Voorbeeld van een methode dat je begroet met de naam die je als parameter ingeefst:

```
static void Begroet(string naam)
{
    Console.WriteLine("Hallo, " + naam + "!");
}
```

7.3.2 Functie

Een functie is een codeblok die een taak uitvoert en **een waarde teruggeeft**. Deze methoden maken gebruik van parameters om berekeningen te maken. Een methode met de eigenschappen van een functie wordt altijd gedefinieerd met een datatype als returntype zoals **int**, **double**, **bool**, **string**, De datatypes van de parameters hoeven hierbij niet hetzelfde te zijn als die van het returntype. Tenslotte moet er altijd een **return statement** aanwezig zijn in de methode, want anders kan er geen waarde van het resultaat worden teruggegeven.

Voorbeeld van een methode dat het product van de ingegeven parameters retourneert:

```
static int Vermenigvuldig(int getal1, int getal2)
{
    int resultaat = getal1 * getal2;
    return resultaat;
}
```

7.4 Parameters

Parameters zijn **variabelen in een methode** die gebruikt worden om gegevens van buitenaf te ontvangen, zodat de methode met verschillende waarden kan werken. Dit heeft als voordeel dat de code niet telkens opnieuw geschreven hoeft te worden. Parameters worden gedefinieerd tussen de haakjes van een methode. Ze bestaan uit een **datatype** gevolgd door een **naam**.

Voorbeeld van een methode met één parameter:

```
static void Begroet(string naam)
{
    Console.WriteLine("Hallo " + naam + "!");
}

static void Main(string[] args)
{
    Begroet("Liam");      //Hallo Liam!
    Begroet("Jenny");    //Hallo Jenny!
}
```

7.4.1 Meerdere parameters in een methode

Je kan **meerdere parameters** toevoegen aan een methode door ze te **scheiden met een komma**. Bij het aanroepen van een methode met meerdere parameters is het belangrijk om de **waarden in dezelfde volgorde** door te geven als de volgorde waarin de parameters gedefinieerd zijn.

Voorbeeld van een methode met twee parameters:

```
static void Leeftijd(string naam, int leeftijd)
{
    Console.WriteLine(naam + " is " + leeftijd + " jaar oud.");
}

static void Main(string[] args)
{
    Leeftijd("Liam", 5);      //Liam is 5 jaar oud.
    Leeftijd("Jenny", 8);    //Jenny is 8 jaar oud.
}
```

7.4.2 Standaard parameterwaarde

Je kan een **standaardwaarde** (of default parameter value) toewijzen aan de parameter in de methode met het gelijkteken `=`. Als gebruikers ervoor zouden kiezen om geen parameter in te geven, zal de methode deze standaardwaarde gebruiken.

Voorbeeld:

```
static void Country(string land = "België")
{
    Console.WriteLine(land);
}

static void Main(string[] args)
{
    Country("Frankrijk");           //Frankrijk
    Country("Nederland");          //Nederland
    Country();                      //België
}
```

7.5 Method overloading

Method (name) overloading (of methodnaam overbelasting) maakt het mogelijk om meerdere versies van een methode te definiëren met **dezelfde naam**, maar met **verschillende parameterlijsten**. Hierdoor kunnen dezelfde acties worden uitgevoerd met verschillende datatypen en/of een verschillend aantal parameters.

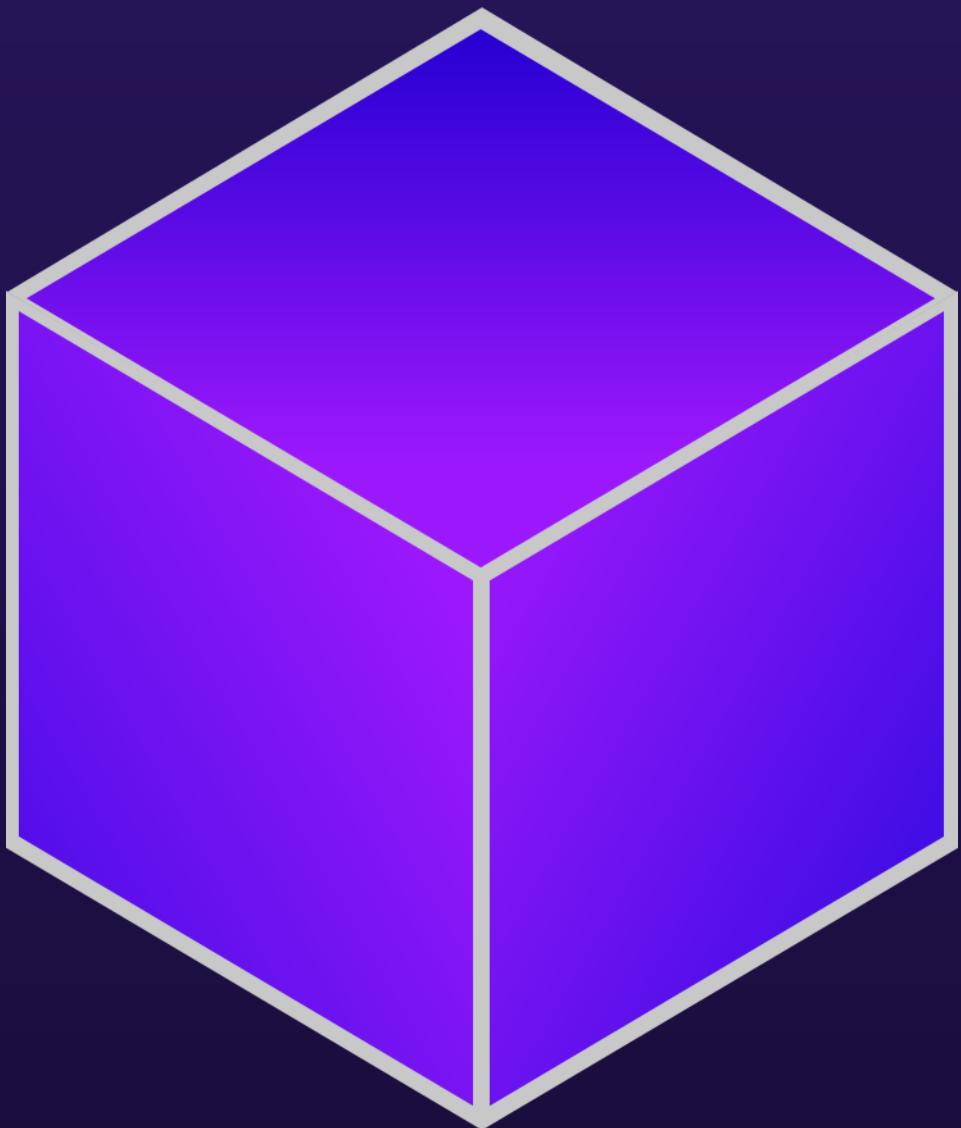
Voorbeelden van dezelfde methodnaam voor het optellen van getallen:

```
//Optellen met 2 integers
static int Optellen(int a, int b)
{
    return a + b;
}

//Optellen met 3 integers
static int Optellen(int a, int b, int c)
{
    return a + b + c;
}

//Optellen met 2 doubles
static double Optellen(double a, double b)
{
    return a + b;
}
```

Deel 2: Object georiënteerd programmeren



8.1 Object georiënteerd programmeren

Dit hoofdstuk introduceert het concept van object georiënteerd programmeren (OOP) en maakt je vertrouwd met de belangrijkste basisprincipes ervan.

OOP is een veelgebruikte manier van programmeren die erop gericht is om een project zo **structureel** mogelijk op te bouwen met **objecten**. Dit **paradigma** wordt tegenwoordig gehanteerd in elke moderne programmeertaal, zoals Java, C++, Python, ..., en is dus zeker niet alleen gebonden aan C#. De populariteit van deze programmeerstijl komt door de vele voordelen die het biedt in vergelijking met andere programmeerparadigma's, zoals gestructureerd (of procedureel) programmeren, waarbij wordt gewerkt met codeblokken, procedures en functies.

We sommen enkele voordelen van OOP op:

- Sneller programmeren.
- Code beter organiseren en structureren.
- Verbeterde analyse en begrip van de situatie.
- Kleine stukjes code worden veel hergebruikt (modulair).
- Eenvoudiger te onderhouden.
- Eenvoudiger te debuggen.
- ...



Klinkt de omschrijving van gestructureerd programmeren je bekend in de oren? Dat komt omdat je tot nu toe voornamelijk op deze manier hebt leren programmeren. Maar wees gerust, we hebben je geen overbodige kennis en vaardigheden aangeleerd. Integendeel, alles wat je tot hiertoe geleerd hebt, zal je nodig hebben om vlot te leren programmeren volgens de principes van OOP.

Bij OOP draait namelijk alles om klassen en objecten die intern nog steeds dezelfde gestructureerde code bevatten zoals selecties, iteraties, methoden, Beschouw het een beetje als een (erg) krachtige upgrade van de programmeerstijl die je tot nu toe hebt gehanteerd.

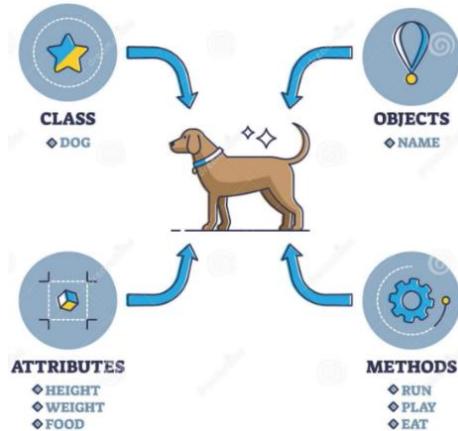
Gestructureerd programmeren kan je trouwens nog altijd prima gebruiken voor kleine oefeningen en applicaties. Voor grote projecten zal je echter zoveel mogelijk gebruik willen maken van OOP.

Naast objecten en klassen, kent OOP vier pijlers die het zo krachtig maken, namelijk:

- Inkapseling.
- Overerving.
- Abstractie.
- Polymorfisme.

8.1.1 Objecten en klassen

Een **klasse** is een blauwdruk of **sjabloon** is waarmee we nieuwe objecten kunnen creëren. Een individueel **object** is op zijn beurt een **instantie** van een klasse met zowel zijn eigen unieke eigenschappen, alsook gemeenschappelijke **eigenschappen** en **methoden** die zijn gedefinieerd in de klasse. Zo kunnen er duidelijk (uiterlijke) verschillen zijn tussen objecten zoals dit bij een chihuahua en een labrador het geval is, maar toch kunnen beide ‘objecten’ deel uitmaken van dezelfde klasse Hond omwille van hun overeenkomstige kenmerken en gedragingen.



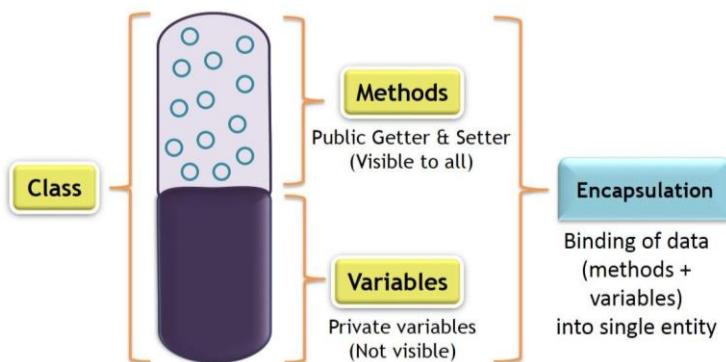
Een object is dus een **verzameling van data** (attributes or fields) **en methoden** (methods) die conceptueel bij elkaar horen. Met behulp van deze objecten proberen we **de echte wereld te modeleren in code**. Zo wordt het mogelijk om code te **structureren** zoals we dat ook in het echte leven doen. Alles om ons heen behoort immers tot een bepaalde klasse die alle objecten van dat **type** beschrijven. Ja dat lees je goed, **klassen** zijn eigenlijk gewoon **een nieuwe vorm van complexere datatypes** waarmee je variabelen een stuk krachtiger worden.

8.1.2 Inkapseling

Inkapseling (of Encapsulation) heeft als doel de **interne details** van een object **af te schermen** van de buitenwereld en alleen datgene bloot te stellen wat nodig is voor de buitenwereld om ermee te communiceren.

Access modifiers spelen hierbij een grote rol door te bepalen welke delen van een klasse **zichtbaar** en **toegankelijk** zijn vanuit andere delen van het programma. Een klasse kan namelijk zowel publieke als private eigenschappen en methoden hebben.

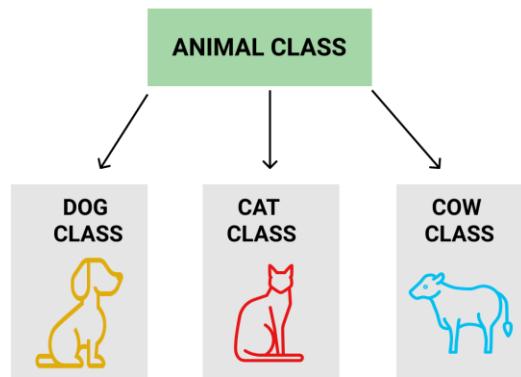
Daarnaast bestaan er speciale methoden genaamd **properties** om de interne staat van objecten in en uit te lezen. Hierover leer je meer in het hoofdstuk over klassen.



8.1.3 Overerving

Overerving (of Inheritance) biedt de mogelijkheid om een nieuwe klasse te creëren door **alle eigenschappen en methoden van een bestaande klasse over te nemen**.

Voorbeeld: van elke diersoort zoals hond, kat, koe ... kan je een aparte klasse maken die afgeleid is van de klasse Dier. Zo is de klasse Hond een **subklasse** van de **superklasse** Dier, maar je kan ze nu wel onderscheiden van de klasse Kat.



Een ander voordeel van subklassen in overerving is dat je er zelf nieuwe eigenschappen en methoden aan kan **toevoegen**, en de implementatie van bepaalde methoden **vervangen** (override). Kortom, met overerving hergebruiken we code en breiden we deze eventueel uit met nieuwe mogelijkheden.

8.1.4 Abstractie

Abstractie (of Abstraction) is het **vereenvoudigen** van complexe systemen door zich te focussen op de essentiële informatie, alsook door onnodige details te negeren. Hierdoor moet het voor gebruikers gemakkelijker worden om het programma te begrijpen en te gebruiken.

Neem bijvoorbeeld een knop op een afstandsbediening. Deze knop verbergt de complexiteit van het interne circuit en de werking ervan. Het volstaat namelijk voor de gebruiker om op de knop te duwen om het gewenste resultaat te bekomen. Dit maakt een afstandsbediening gemakkelijk begrijpbaar en bruikbaar voor de gebruiker ervan.

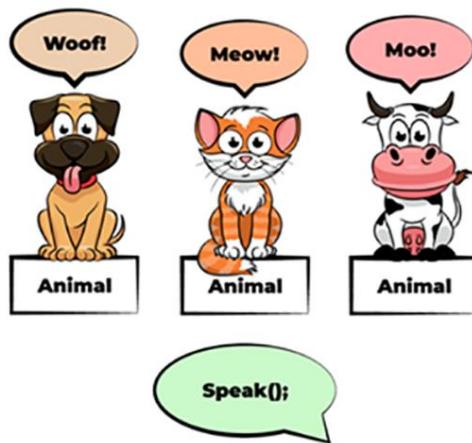


Dit doet misschien denken aan inkapseling, maar er zijn toch duidelijke verschillen. Abstractie gaat namelijk over het begrijpen van wat een object doet, terwijl inkapseling gaat over het beheren van wat een object doet en het beschermen van de interne details.

8.1.5 Polymorfisme

De term polymorfisme (of Polymorphism) betekent letterlijk **veelvormigheid of meerdere vormen**. Dit houdt in dat objecten van een subklasse behandeld kunnen worden als objecten van de superklasse waar ze van overerven. Concreet houdt dit in dat objecten **op verschillende manieren kunnen reageren op eenzelfde methode-aanroep**.

Neem als voorbeeld de methode Speak() die door verschillende dieren kan worden aangeroepen. In plaats van één standaard dierengeluid, kan elk dier de methode op zijn eigen manier implementeren. Daardoor zal een hond-object antwoorden met "Woof!" en een kat-object met "Meow!", en dit terwijl ze gebruik maken van exact dezelfde methode.



9.1 Wat zijn objecten?

C# is een object-georiënteerde programmeertaal dat objecten als basis gebruikt om code te organiseren en te hergebruiken. Maar wat is een object nu precies?

Tot nu toe hebben we primitieve datatypen en strings kunnen gebruiken om waarden te koppelen aan variabelen. Denk bijvoorbeeld maar aan een `int` voor leeftijden, `double` voor percentages of `strings` voor namen.

Maar wat als we een object zoals een boek, auto of zelfs een persoon willen declareren in onze programmacode? Deze objecten zijn immers veel te complex om eenvoudigweg beschreven te worden als een primitief datatype. Een boek heeft bijvoorbeeld niet alleen een titel (`string`), maar ook een aantal pagina's (`int`).

Men kan software-objecten dus vergelijken met objecten uit het dagelijkse leven. Een object heeft trouwens niet alleen **eigenschappen** (of properties), maar ook **gedragingen in de vorm van methoden** (of methods).

Neem nu een auto als voorbeeld:

- Eigenschappen: kleur, afmetingen, bouwjaar, cilinderinhoud, maximumsnelheid, ...
- Gedragingen: rijden, remmen, draaien, ...

De eigenschappen kunnen we gelukkig wel onderbrengen in variabelen die verbonden zijn aan het object. Voor de gedragingen van het object kunnen we dan weer gebruik maken van methoden die verbonden zijn aan het object.

9.2 Klassen en objecten

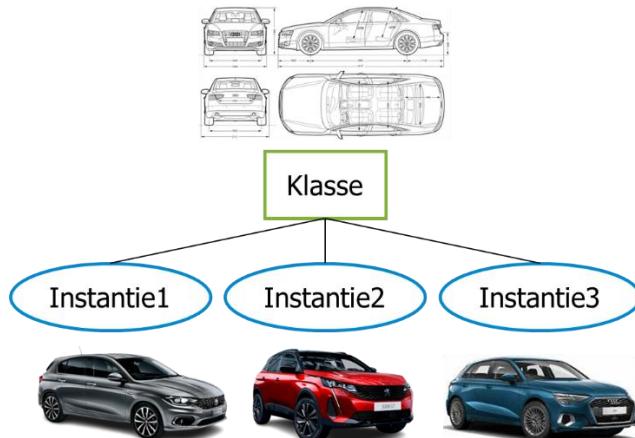
Als we het voorbeeld van de auto nemen, kan je stellen dat veel mensen met een auto rijden. Deze auto's zijn allemaal uniek, want er zijn altijd wel kenmerken of gedragingen waarin ze verschillen van elkaar. Toch kan iedereen deze unieke objecten herkennen en categoriseren als een auto. Dat komt omdat iedere auto behoort tot eenzelfde soort ontwerp of blauwdruk.

Voorbeeld van een Auto-object:

- Klasse: auto
- Object: MCI2107
- Eigenschappen: merk BMW, model M3, kleur grijs, gewicht 1.700 kg, brandstof benzine, ...
- Gedragingen: starten, rijden, versnellen stoppen, draaien, claxonneren, tanken, ...



Een **klasse** is een blauwdruk of **sjabloon** voor het maken van objecten. Een klasse bepaalt namelijk de eigenschappen en methoden van objecten van dezelfde soort. Elk **object** is dan ook een unieke **instantie** van een klasse.



Wanneer afzonderlijke objecten worden gemaakt, erven ze alle variabelen en methoden uit de overeenkomstige klasse.

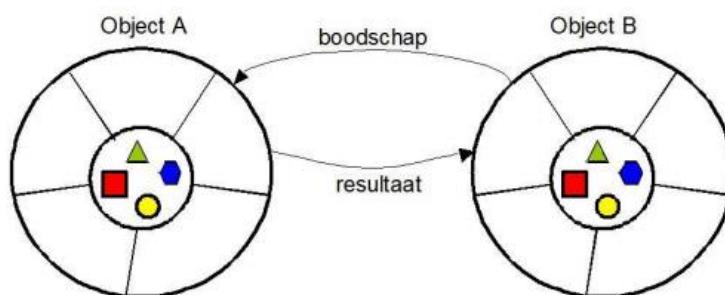
Alle eigenschappen die gemeenschappelijk zijn voor objecten van een klasse noemt men een **class variabele** (of klasse-variabele). Alle auto's hebben bijvoorbeeld vier wielen. Er zijn echter ook eigenschappen die verschillend zijn voor iedere instantie, zoals de kleur van de auto. Zo'n eigenschap noemt men een **instance variabele**.

Hetzelfde geldt voor methoden: gemeenschappelijke methoden noemt men **class methods** en methoden die gekoppeld zijn aan een concreet object, of instantie, noemt men **instance methods**.

9.3 Boodschappen

Een programma bestaat meestal uit verschillende objecten die met elkaar samenwerken en communiceren via boodschappen. Het ene object stuurt een boodschap naar het andere object om het iets te laten doen. Vervolgens stuurt het ontvangende object het resultaat van de actie terug.

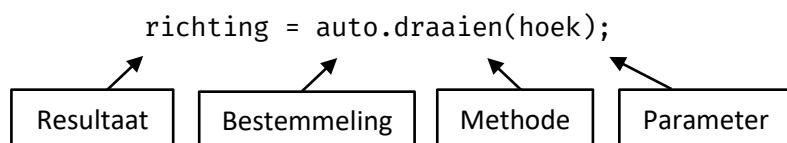
Het zenden van boodschappen verloopt via het aanroepen van methoden van een object.



Boodschappen hebben 4 kenmerken:

- **Bestemming**: het object waaraan de boodschap gericht is.
- **Methode**: handeling van het object dat wordt aangeroepen.
- **Parameters**: bijkomende informatie over de boodschap (optioneel).
- **Resultaat**: uitkomst van de boodschap (optioneel).

Neem terug de auto als voorbeeld. Als de bestuurder aan het stuur draait, verandert de auto van richting. De bestuurder stuurt dus een boodschap naar de auto om te draaien.



Samengevat:

- De **bestemming** van de boodschap is de auto.
 - De aangeroepen **methode** is draaien.
 - De **parameter** is de hoek waarover de bestuurder het stuur draait.
 - Het **resultaat** is dat de auto én de bestuurder van richting veranderen.

9.4 Objecten aanmaken

Een object wordt gemaakt op basis van een blauwdruk of sjabloon; namelijk de **klasse** waartoe het object behoort. Er moet dus eerst een klasse worden geschreven alvorens je objecten van die klasse kan creëren.

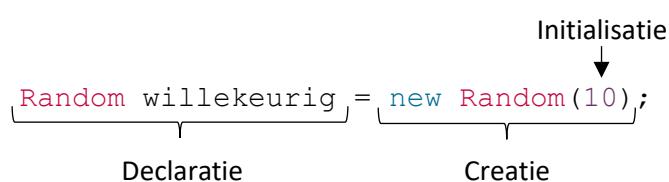
 Het definiëren van een eigen klasse behandelen we pas in H11 Klassen.

Gelukkig bestaan er ook klassen die reeds gedefinieerd zijn in het programma. Elke klasse behoort tot een **namespace** die eerst geïmporteerd moet worden alvorens deze kan worden gebruikt om objecten aan te maken.

Hieronder heb je voorbeelden van interessante klassen met hun bijhorende namespaces:

- namespace System:
 - Object
 - String
 - DateTime
 - Math
 - Console
 - Convert
 - Random
 - ...
 - namespace System.Text:
 - Encoding
 - StringBuilder
 - ...

In het volgende voorbeeld maken we een random number generator aan van de klasse Random.



De opbouw van een nieuw object bestaat uit 3 onderdelen:

1. De declaratie van het object
2. De creatie van het object
3. De initialisatie van het object

9.4.1 Declaratie van het object

Je hebt reeds geleerd dat je primitieve variabelen (waardetypen) declareert als volgt:

```
datatype naam;
```

Een referentievariabele wordt op dezelfde manier gedeclareerd. Het type is in dit geval geen waardetype, maar de **naam van de klasse**. Vervolgens kies je op dezelfde manier een **naam voor het object**.

Syntax:

```
Klasse objectnaam;
```

Voorbeeld van de klasse Random:

```
Random willekeurig;
```

9.4.2 Creatie van het object

De creatie (of instantiatie) van een object gebeurt met de operator `new` gevolgd door de **naam van de klasse** met daarachter gesloten haken. Zo bekomen we een nieuwe instantie van een klasse.

Syntax:

```
new Klasse();
```

Voorbeeld van de klasse Random:

```
new Random();
```

9.4.3 Initialisatie van het object

Tussen de ronde haken kan eventueel een aantal parameters worden ingegeven. Met deze parameters wordt een object geïnitialiseerd.

Voorbeeld van de klasse Random:

```
new Random(10);
```

Bij het aanmaken van een object wordt een speciale methode aangeroepen; namelijk de **constructor**. De constructor wordt enkel gebruikt om een object tijdens de creatie te **initialiseren**. Iedere klasse heeft één of meerdere constructors die van elkaar verschillen in type en aantal parameters. Daardoor kan een object op verschillende manier geïnitialiseerd worden.



Constructors kan je dus net als methoden **overloaden**.

9.4.4 Het resultaat van de declaratie en initialisatie van de referentievariabele

Tijdens de creatie van een object, wordt er ruimte in het geheugen gereserveerd voor dat object. Het resultaat van de creatie is dus niet het object zelf, maar wel een **verwijzing** (of referentie) naar de **plaats in het geheugen** waar het object is opgeslagen.

Een referentievariabele die naar geen enkel object verwijst heeft standaard de waarde `null`.

9.4.5 Namespaces importeren

In ons voorbeeld behoort de klasse Random tot de namespace System. De volledige naam van de klasse is dus eigenlijk System.Random. Het is echter omslachtig om dit telkens te moeten typen. Daarom is het interessanter om de namespace eerst te importeren met behulp van het `using` statement. Dit gebeurt helemaal bovenaan in de broncode.

Syntax:

```
using Namespace;
```

Voorbeeld:

```
using System.Text;
```

In het voorbeeld wordt de gehele namespace System.Text geïmporteerd om toegang te krijgen tot alle klassen en methoden binnen deze namespace.

Tegenwoordig zijn er al meerdere namespaces standaard geïmporteerd in Visual Studio wanneer je een nieuw project aanmaakt. Bovendien wordt een namespace (gewoonlijk) automatisch geïmporteerd als je een bepaalde klasse van die namespace wilt gebruiken. Moest dit toch niet automatisch gebeuren, moet je de namespace alsnog manueel importeren met het `using` statement.

Voorbeeld: Wanneer je een `StringBuilder`-object aanmaakt, zal de namespace System.Text automatisch worden geïmporteerd.

9.5 Objecten gebruiken

De creatie en het gebruik van een object wordt hieronder in stappen toegelicht aan de hand van een voorbeeld:

- 1) Definieer een nieuwe **klasse** (`Persoon`), of gebruik een bestaande klasse.

```
Public class Persoon
{
    //Eigenschappen
    public string Naam { get; set; }
    public int Leeftijd { get; set; }

    //Methode
    public void ZegHallo()
```

```

    {
        Console.WriteLine("Hallo, mijn naam is " + Naam + " en ik ben" +
        Leeftijd + " jaar oud.");
    }
}

```

- 2) Maak een **nieuwe instantie** aan van de klasse.

```
Persoon simon = new Persoon();
```

- 3) Geef het object **eigenschappen** om mee te werken.

```
simon.Naam = "Simon Van Zandvliet";
simon.Leeftijd = 18;
```

- 4) Roep **methoden** van het object aan.

```
simon.ZegHallo();
```

Volledig overzicht van het voorbeeld:

```

Public class Persoon
{
    //Eigenschappen
    public string Naam { get; set; }
    public int Leeftijd { get; set; }

    //Methode
    public void ZegHallo()
    {
        Console.WriteLine("Hallo, mijn naam is " + Naam + " en ik ben" +
        Leeftijd + " jaar oud.");
    }
}

class Program
{
    static void Main()
    {
        Persoon simon = new Persoon();
        simon.Name = "Simon Van Zandvliet";
        simon.Age = 18;
        simon.ZegHallo();
    }
}

```

In dit voorbeeld hebben we een object genaamd Simon aangemaakt. Vervolgens hebben we aan de eigenschap Naam de volledige naam "Simon Van Zandvliet" gegeven en aan de eigenschap Leeftijd de leeftijd van 18 (jaar oud). Tot slot roepen we de methode zegHallo() aan die de volgende uitvoer geeft:

```
Hallo, mijn naam is Simon Van Zandvliet en ik ben 18 jaar oud.
```

9.6 Objecten opruimen

Objecten nemen geheugen in beslag van zodra ze worden aangemaakt. Om te voorkomen dat een programma onnodig veel geheugen gebruikt, moeten objecten die niet meer nodig zijn, worden opgeruimd. Hierdoor komt het gereserveerde geheugen weer vrij.

In talen zoals C en C++ moet de programmeur nauwkeurig bijhouden welke objecten niet meer worden gebruikt om het gereserveerde geheugen vrij te geven. Gelukkig hebben moderne programmeertalen zoals Python en C# een automatisch geheugensysteem dat verantwoordelijk is voor het volgen en opruimen van ongebruikte objecten in het geheugen: de **garbage collector**.

Een object wordt gemarkerd als ‘niet meer in gebruik’ als iedere referentie naar het object verdwenen is. Vervolgens wordt deze opgeruimd tijdens de **garbage collection**. Dit gebeurt in een afzonderlijke thread met een lage prioriteit en heeft geen invloed op de prestaties van het programma.

- (i) Multithreading is een techniek waarbij een programma meerdere threads uitvoert binnen een enkel proces. Elke thread kan onafhankelijk werken, waardoor taken gelijktijdig kunnen worden uitgevoerd en de efficiëntie van het programma wordt verhoogd. Dit is vooral nuttig voor taken die kunnen worden opgesplitst en parallel verwerkt, zoals het afhandelen van meerdere gebruikersverzoeken of het uitvoeren van complexe berekeningen.

Het leren werken met multithreading valt helaas buiten de scope van dit handboek.

Zoals je merkt biedt de garbage collector heel wat voordelen voor de programmeur, want die hoeft zich geen zorgen meer te maken over geheugenlekken of andere geheugenproblemen. Bovendien kan de focus zo meer gelegd worden op het schrijven van code.

9.7 De willekeurige klasse Random

de klasse `Random` is een ingebouwde klasse die wordt gebruikt voor het **genereren van (pseudo)willekeurige getallen**.

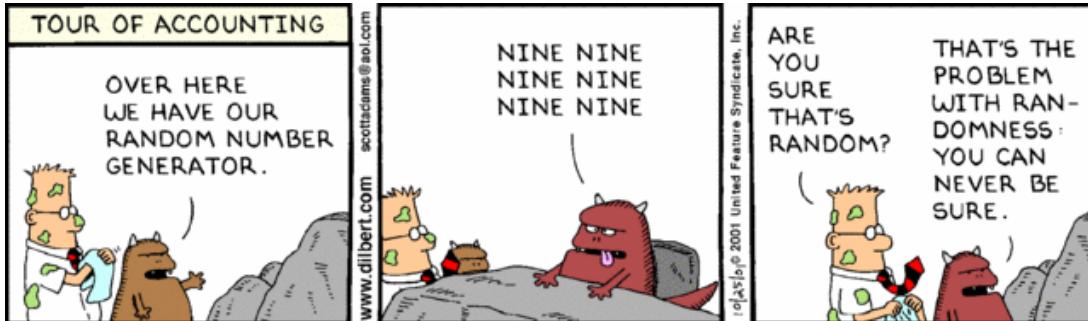
Een pseudowillekeurig getal is een reeks getallen die, hoewel ze op het eerste gezicht willekeurig lijken, het resultaat zijn van een wiskundig algoritme. Dit algoritme neemt een beginwaarde (of seed) en gebruikt dat als startpunt om een reeks getallen te genereren die op het eerste gezicht willekeurig lijken.

9.7.1 Constructors van de klasse Random

Constructor	Beschrijving
<code>Random()</code>	Creëert een nieuwe random-generator met een willekeurige seed (gebaseerd op de systeemklok).
<code>Random(bovengrens + 1)</code>	Creëert een nieuwe random-generator met een opgegeven seed.
<code>Random(ondergrens, bovengrens + 1)</code>	

Pseudowillekeurige getallen zijn deterministisch. Dat betekent dat als je dezelfde beginwaarde (seed) gebruikt, het algoritme steeds dezelfde reeks getallen zal genereren. Dit kan nuttig zijn bij het testen van software of in situaties waarin herhaalbaarheid belangrijk is.

Indien we een constructor nemen zonder argumenten, zal intern een willekeurige seed gebruikt worden. Zo zullen twee Random-objecten die gecreëerd zijn zonder seed hoogstwaarschijnlijk niet dezelfde reeks willekeurige getallen opleveren.



9.7.2 Methoden van de klasse Random

In deze tabel worden de **meest gebruikte** methoden van de klasse Random weergegeven:

Returntype	Methode	Beschrijving
int	<code>Next()</code>	Geeft de volgende willekeurige int.
long	<code>NextInt64()</code>	Geeft de volgende willekeurige long.
float	<code>NextSingle()</code>	Geeft de volgende willekeurige float.
double	<code>NextDouble()</code>	Geeft de volgende willekeurige double.

Hierbij wordt telkens aangegeven wat het datatype van de parameters is en wat het datatype van de teruggegeven waarde (returntype) is. Indien er geen waarde wordt teruggegeven is het datatype void.

Declaratie en creatie van een Random-object:

```
Random rng = new Random(); //rng = random number generator
```

Voorbeelden van methode aanroepen van een Random-object:

```
//Een zuivere dobbelsteen met 6 vlakken werpen:  
int dobbelsteen1 = rng.Next(6) + 1;  
int dobbelsteen2 = rng.Next(1, 7);  
  
//Een muntstuk werpen (kop = 0 en munt = 1):  
int muntstuk = rng.Next(2);  
  
//Een willekeurig decimal getal tussen 0.0 en 1.0:  
double kommagetal = rng.NextDouble();  
  
//Een geheel getal genereren van exact 3 cijfers:  
double getal = rng.Next(900) + 100;
```

De bovengrens moet altijd één waarde hoger liggen dan de hoogste waarde in het opgegeven bereik. Als we het getal 6 dus als hoogste waarde in de reeks willen opgeven, moeten we 7 (= 6 + 1) ingeven.

De ondergrens heeft standaard de waarde nul (0). Je hoeft dus geen ondergrens op te geven als de laagste waarde in een reeks het getal nul moet zijn.

9.8 De tekenreeksklasse `String`

Zoals je weet is `string` een tekenreeks, oftewel een opeenvolging van karakters. In C# zijn er meerdere klassen die tekenreeksen kunnen bevatten. In deze cursus beperken we ons tot de klassen `String` en `StringBuilder`.

Een object van de klasse `String` is een **onveranderlijke tekenreeks**. De toegekende waarde van een string kan achteraf dus niet meer veranderd worden. Bijgevolg betekent dit dat elke verandering in een string eigenlijk een nieuw `String`-object creëert.

9.8.1 Constructors van de klasse `String`

Constructor	Beschrijving
<code>String()</code>	Maakt een lege string.
<code>String(string literal)</code>	Maakt een string met dezelfde inhoud als de meegegeven string.

Voorbeeld:

```
String tekst = new String("Hello world!");
```

Via de constructor wordt de tekenreeks "Hello world!" als een **string literal** ingegeven.

In tegenstelling tot andere objecten, hoef je strings niet aan te maken met behulp van een `new` operator. Dat komt omdat strings in C# anders worden behandeld vanwege hun onveranderlijkheid.

`String`-objecten kan je ook declareren en initialiseren op dezelfde manier als een waardetype:

```
string tekst = "Hello world!";
```

Het is dus niet nodig om expliciet een instantie van de klasse `String` te maken. Je kunt simpelweg een `string` variabele declareren en initialiseren met een string literal tussen dubbele aanhalingstekens ("").

De methode `Console.WriteLine()` aanvaardt ook een string literal als parameter:

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

Op de achtergrond wordt dan eigenlijk een nieuwe `String`-object gecreëerd en geïnitialiseerd met als "waarde" de tekenreeks "Hello world!".

9.8.2 Eigenschappen van de klasse String

Returntype	Eigenschap	Beschrijving
int	Length	Geeft het aantal karakters in de string.
char	Stringnaam[index]	Geeft het teken op een gegeven positie (index) in de string.

Met de index van een string zal het String-object intern kijken welk karakter op deze positie staat. Vervolgens wordt dit karakter teruggegeven. De positie van karakters begint altijd bij nul, wat betekent dat het eerste karakter in de string zich op index 0 bevindt, het tweede karakter op index 1,

Voorbeelden:

```
string tekst = "Hello world!";
Console.WriteLine(tekst.Length);           //12
Console.WriteLine(tekst[0]);                //H
Console.WriteLine(tekst[tekst.Length - 1]); //!
```

In het voorbeeld willen we de eerste en de laatste letter van de string laten afdrukken. Aangezien de indexering begint bij 0, kunnen we niet de eigenschap `tekst.Length` als index ingeven omdat de laatste index van de string eindigt bij index 11, en niet bij 12.

Als je toch een hogere index dan 11 ingeefdt, levert dit een `IndexOutOfRangeException` op als foutmelding.



Merk op dat alle eigenschappen zonder haakjes worden geschreven.

De enige uitzondering hierop zijn de eigenschappen met een verwijzing naar een specifieke index. Dit zal allemaal uitgelegd worden in het volgende hoofdstuk, maar hopelijk maken de gegeven voorbeelden al veel duidelijker.

9.8.3 Methoden van de klasse String

In deze tabel worden de **meest gebruikte** methoden van de klasse String weergegeven:

Returntype	Methode	Beschrijving
int	CompareTo(str)	Vergelijkt deze string alfabetisch met een andere string en geeft de volgende waarde terug: <ul style="list-style-type: none">-1: indien deze string kleiner is.0: indien de strings gelijk zijn.1: indien deze string groter is.
bool	Contains(str)	Controleert of een substring aanwezig is in een string en geeft true terug als de substring wordt gevonden.
bool	Equals(str)	Vergelijkt 2 strings en geeft true terug als ze hetzelfde zijn.

int	IndexOf(char/str)	Geeft de eerste positie van een karakter of substring in een string.
string	Replace(str1, str2)	Geeft een nieuwe string terug waarbij overal de karakters van str1 zijn vervangen door str2.
string	ToLower()	Zet de tekst in een string om in kleine letters.
string	ToUpper()	Zet de tekst in een string om in hoofdletters.
string	Trim()	Geeft een nieuwe string terug waarbij de lege ruimte (tabs en spaties) ervoor en erna zijn verwijderd.

Voorbeelden:

```
string man = "Jan";
string vrouw = "Marie";

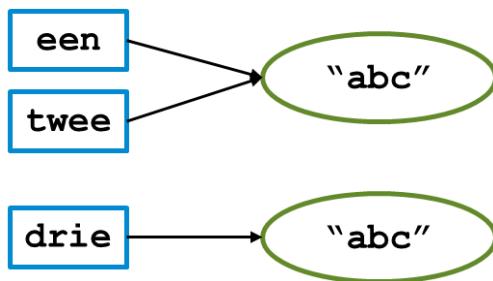
int negatief = man.CompareTo(vrouw);           // -1
int nul = vrouw.CompareTo("Marie");           // 0
int positief = vrouw.CompareTo(man);          // 1
bool bevatMar = vrouw.Contains("Mar");         // True
bool gelijk = man.Equals(vrouw);               // False
int positie = vrouw.IndexOf("rie");            // 2
string vervang = vrouw.Replace("e", "anne");    // Marianne
string kleineLetters = man.ToLower();           // jan
string hoofdletters = vrouw.ToUpper();          // MARIE
```

9.8.4 Geheugengebruik bij tekenreeksen

Doordat strings onveranderbaar zijn, zouden strings met dezelfde inhoud (duplicaten) meer geheugen innemen. Om dit te voorkomen wordt een overzicht van String-objecten bijgehouden; de **string constant pool**. Dit is een mechanisme waarmee string literals slechts éénmaal in het geheugen worden geplaatst.

Er is echter een uitzondering, want strings die worden gecreëerd met de new operator, worden niet in de string constant pool geplaatst.

```
string een = "abc";
string twee = "abc";
String drie = new String("abc");
```



9.9 De tekenreeksmanipulatieklasse `StringBuilder`

De klasse `StringBuilder` biedt een efficiënte manier om **strings te manipuleren** zoals het toevoegen, invoegen en verwijderen van tekens. Dat komt omdat, in tegenstelling tot strings, de **tekenreeksen** van deze klasse wél **veranderlijk** zijn. Je kan de inhoud van een `StringBuilder`-object immers wijzigen zonder steeds nieuwe objecten te creëren.

9.9.1 Constructors van de klasse `StringBuilder`

Constructor	Beschrijving
<code>StringBuilder()</code>	Maakt een lege <code>StringBuilder</code> .
<code>StringBuilder(string literal)</code>	Maakt een <code>StringBuilder</code> met dezelfde inhoud als de meegegeven string.

Voorbeeld:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("stringliteral");
```

Merk op dat een `StringBuilder`-object alleen met een `new` operator kan worden aangemaakt. De parameter kan een stringvariable of een string literal zijn.

9.9.2 Eigenschappen van de klasse `StringBuilder`

Returntype	Eigenschap	Beschrijving
<code>int</code>	<code>Length</code>	Geeft het aantal karakters in de <code>StringBuilder</code> .
<code>char</code>	<code>sbnaam[index]</code>	Geeft het teken op een gegeven positie (index) in de <code>StringBuilder</code> .

9.9.3 Methoden van de klasse `StringBuilder`

In deze tabel worden de **meest gebruikte** methoden van de klasse `StringBuilder` weergegeven:

Returntype	Methode	Beschrijving
<code>StringBuilder</code>	<code>Append(str)</code>	Voegt de inhoud van variabelen en literals toe aan het einde van de <code>StringBuilder</code> .
<code>void</code>	<code>Clear()</code>	Verwijdt alle tekens uit de <code>StringBuilder</code> .
<code>Bool</code>	<code>Equals(sb)</code>	Vergelijkt twee <code>StringBuilder</code> s inhoudelijk en geeft true terug indien ze hetzelfde zijn.
<code>StringBuilder</code>	<code>Insert(index, str)</code>	Voegt een nieuwe string of een karakter in op een opgegeven positie in de <code>StringBuilder</code> .
<code>StringBuilder</code>	<code>Remove(index, length)</code>	Verwijdt een opgegeven aantal tekens, beginnend bij een bepaalde positie.
<code>StringBuilder</code>	<code>Replace(str1, str2)</code>	Vervangt alle exemplaren van een bepaalde substring door een andere substring.

string	ToString()	Converteert de StringBuilder naar een string.
--------	------------	---

Voorbeelden:

```

string gek = " crazy";
StringBuilder sb1 = new StringBuilder("Hello");
StringBuilder sb2 = new StringBuilder("Hello beautiful world");

sb1.Length;                                //5
sb1.Append(gek);                            //Hello crazy
sb1.Append(" world!");                      //Hello crazy world!
sb1.Remove(6,6);                            //Hello world!
sb1.Insert(6, "beautiful ");                //Hello beautiful world!
sb1.Equals(sb2);                           //False
sb1.Replace(" beautiful", " , you are my"); //Hello, you are my world!
sb1.ToString();                            //Hello, you are my world!
sb1.Clear();                                //

```

9.10 String Interpolatie

String interpolatie is een handige manier om het samenstellen van strings en variabelen te vereenvoudigen. Het biedt namelijk een duidelijker en meer leesbare manier om stringcomposities uit te voeren in vergelijking met andere manieren zoals de plusoperator (+), of de methode `String.Format`.

i De methode `String.Format` is een oude manier om stringcomposities uit te voeren. Het wordt trouwens nog steeds gebruikt in talen zoals C en C++, maar aangezien we met string interpolatie in C# op een veel eenvoudigere manier hetzelfde resultaat kunnen bekomen, wordt deze methode niet behandeld in deze cursus.

String interpolatie wordt bereikt door een dollarteken (\$) voor een string literal te plaatsen en vervolgens de variabelen tussen accolades { } te plaatsen.

```

string naam = "Laura";
intleeftijd = 21;

//Stringcomposities met de plusoperator:
string boodschap = "Hallo, ik ben " + naam + " en ik ben " + leeftijd + " jaar oud.';

//Dezelfde stringcompositie, maar met string interpolatie:
string boodschap = $"Hallo, ik ben {naam} en ik ben {leeftijd} jaar oud.'";

```

9.10.1 Format specifiers

Wat string interpolatie nog interessanter maakt, is dat je de notatie van getallen, datums en tijd kan aanpassen door gebruik te maken van **format specifiers** binnen de accolades. Hiermee bepaal

je dus de weergave van numerieke waarden zoals het aantal decimalen, de notatie van valuta, scheidingstekens tussen duizendtallen,

Je kan een format specifier toekennen aan je variabele door achter de naam van de variabele een dubbel punt (:) te plaatsen gevolgd door de gewenste symbolen.

Voorbeeld:

```
double pi = 3.14159265359;  
Console.WriteLine($"Pi, afgerond op 2 decimalen, is {pi:0.00}");
```

In de afdruk van het voorbeeld zal de waarde `pi` afgerond worden weergegeven op twee cijfers na de komma:

```
Pi, afgerond op 2 decimalen is 3,14.
```

9.10.1.1 Overzicht van format specifiers

In deze tabel worden de **meest gebruikte** format specifiers weergegeven die je kan toevoegen aan een variabele in een string interpolatie:

Symbol	Beschrijving	Notatie	Voorbeeld
<i>Algemene (aangepaste) notaties</i>			
0	Geeft op die plaats een cijfer of een 0 weer.	{:0.00}	1234,57
#	Geeft op die plaats een cijfer of niets weer.	{:(#).##}	(1234),57
.	Scheidingsteken kommagetallen (decimalen).	{:0.000}	1234,568
,	Scheidingsteken duizendtallen.	{:0,0}	1.235
%	Geeft het getal weer als percentage.	{:0.0%}	123456,8%
<i>Getalnotaties</i>			
c	Geeft het getal weer als valuta.	{:c}	\$ 1.234,57
e	Geeft het getal weer in de wetenschappelijke notatie.	{:e}	1,234568e+003
n	Geeft het getal weer met scheidingstekens voor duizendtallen.	{:n}	1.234,57
p	Geeft het getal weer als percentage.	{:p}	123.456,78%
<i>datumnotaties</i>			
d	Geeft de korte datumnotatie weer.	{:d}	29/11/2023
D	Geeft de lange datumnotatie weer.	{:D}	woensdag 29 november 2023
M	Geeft de maand en dag weer.	{:M}	29 november
Y	Geeft de maand en jaar weer.	{:Y}	november 2023

tijdnotaties			
t	Geeft de korte tijdnotatie weer.	{:t}	21:25
T	Geeft de lange tijdnotatie weer.	{:T}	21:25:08
ss	Geeft de seconden weer.	{:ss}	08
mm	Geeft de minuten weer.	{:mm}	25
hh	Geeft het uur weer (12u klok).	{:hh}	09
HH	Geeft het uur weer (24u klok).	{:HH}	21

- i** Merk op dat er nog meer format specifiers bestaan. Deze kan je gemakkelijk vinden op het internet. Visual Studio helpt je ook verder wanneer je ze wilt gebruiken in een string interpolatie.

10.1 Wat zijn arrays?

In veel programma's moeten grote hoeveelheden gegevens (getallen, tekst, datums, tijd en objecten) worden opgeslagen in het geheugen. Het is echter niet praktisch of efficiënt om voor elk gegeven een aparte variabele te declareren en initialiseren. Er zijn immers heel wat gegevens die je zou kunnen groeperen onder één categorie.

Voorbeelden:

- Namens van leerlingen in een klas.
- Winnende lottocijfers.
- Deelnemers van een wedstrijd.

Om dit probleem op te lossen kan je dankbaar gebruik maken van arrays. Een **array** is namelijk een verzameling van **gelijkaardige variabelen** (of elementen) van **hetzelfde datatype** onder **één naam**. Elk element is hierbij voorzien van een **rangnummer** (of index) dat de plaats van dat element in de array aanduidt.

Voorbeelden:

- Namens van 9 leerlingen in een klas, opgeslagen als een verzameling van strings.

klas	Hayley	Luigi	Sim	Kai	Joren	Ismael	Axelle	Liam	Quin
Index	0	1	2	3	4	5	6	7	8

- De 7 winnende lottocijfers, opgeslagen als een verzameling van int.

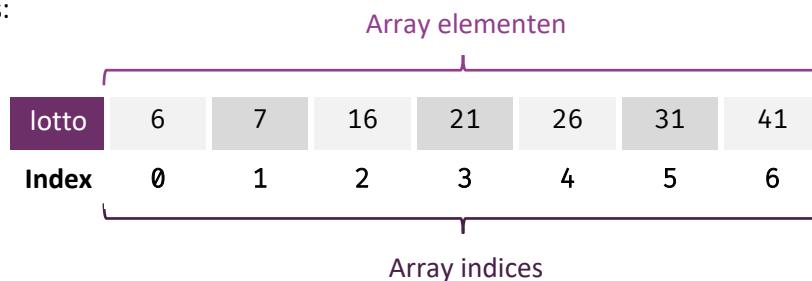
lotto	6	7	16	21	26	31	41
Index	0	1	2	3	4	5	6

Een array is dus een rij of verzameling van elementen. Deze elementen kunnen zowel waarde- als referentietypen zijn.

10.1.1 Elementen en indexwaarden

Een array heeft een **vaste lengte**, uitgedrukt in het **aantal elementen**. Dit zorgt soms voor verwarring aangezien het **eerste element** in een array altijd start met **indexnummer 0**. Bijgevolg heeft het **laatste element** altijd **indexwaarde lengte - 1** in plaats van de lengte van de array zelf.

Dit zien we duidelijk in het voorbeeld van de 7 winnende lottocijfers waarbij 6 (7 - 1) de laatste indexwaarde is:



De arraynaam en de index tussen vierkante haakjes bepalen het element: `lotto[1] = 7`

10.2 Arrays creëren

Net als bij variabelen moet een array voorzien worden van een datatype met daarachter een zelfgekozen naam. Je kan ervoor kiezen om eerst een array te declareren en achteraf elementen eraan toe te voegen, of je kan onmiddellijk tijdens de creatie elementen toevoegen aan een array.

10.2.1 Met een opgegeven lengte

In C# is de array op zich een **object**. Bijgevolg worden arrays net als objecten gecreëerd met de `new` operator. Tussen de vierkante haakjes wordt vervolgens de lengte (het aantal elementen) in de array vastgelegd. Deze lengte mag ook een afzonderlijke variabele zijn.

Syntax:

```
datatype[] naam = new datatype[lengte];
```

Voorbeeld van een literal als lengte:

```
int[] lottoGetallen = new int[7];
```

Voorbeeld van een variabele als lengte:

```
int aantalLeerlingen = 9;
string[] leerlingen5AD = new string[aantalLeerlingen];
```

Na de creatie van een lege array worden de waarden van de elementen automatisch geïnitialiseerd met de respectievelijke standaardwaarden `0`, `null` of `false`.

10.2.2 Met een opsomming van elementen

Meteen na de declaratie kan je een array aanvullen met waarden. Je hoeft hierbij geen bovengrens (of grootte) te vermelden. Deze wordt immers automatisch bepaald door het aantal opgegeven elementen tussen de accolades `{ }` .

Syntax:

```
datatype[] naam = { //opsomming van elementen };
```

Voorbeelden:

```
string[] autos = {"Volvo", "BMW", "Ford", "Mazda"};
int[] lottonummers = {6, 7, 16, 21, 26, 31, 41};
```

10.3 Arrays gebruiken

10.3.1 Verwijzen naar een element in een array

We kunnen de elementen van een array gebruiken door de arraynaam te nemen, gevolgd door vierkante haken met daartussen de index van het element dat je wenst te gebruiken.

Syntax:

```
naam[index]
```

Voorbeeld:

```
int[] lottonummers = {6, 7, 16, 21, 26, 31, 41};  
lottonummers[0] //6  
lottonummers[6] //41
```

10.3.2 Waarden in een array zetten

Je kan rechtstreeks waarden toekennen aan elementen van een array door er een **literal** aan toe te kennen in de programmacode.

```
lottonummers[0] = 6;
```

Daarnaast kan je de gebruiker een waarde laten invullen door deze **in te lezen**.

```
lottonummers[1] = Convert.ToInt32(Console.ReadLine());
```

Je kan in één keer efficiënt alle waarden laten invullen door gebruik te maken van een begrensde herhaling (for-lus met de eigenschap Length).

```
for (int i = 0; i < lottonummers.Length; i++)  
{  
    Console.Write($"Geef lottonummer {i}: ");  
    lottonummers[i] = Convert.ToInt32(Console.ReadLine());  
}
```

Nu wordt meteen de kracht van arrays duidelijk. Met enkele coderegels van een begrensde herhaling zou je immers duizenden waarden kunnen opslaan.

10.3.3 Waarden in een array aanpassen

Je kan de waarden van elementen in een array gebruiken zoals een gewone variabele. Dit wilt dus ook zeggen dat de huidige waarden van een element kunnen worden aangepast.

Voorbeelden:

```
int[] lottonummers = {6, 7, 16, 21, 26, 31, 41};  
lottonummers[0] += lottonummers[1]; //6 + 7 = 13  
lottonummers[2] *= 2; //16 * 2 = 32  
lottonummers[3]--; //21 - 1 = 20
```

10.4 Waarden van arrays tonen

10.4.1 Een bepaalde waarde tonen

Je kan een bepaalde waarde tonen door de positie (index) in de array tussen vierkante haakjes op te geven.

```
Console.WriteLine(lottogetallen[0]);
```

10.4.2 Alle waarden van een array tonen

Het kan echter ook nodig zijn om alle elementen van een array af te lopen en tonen op het scherm.

Hiervoor gebruik je best een begrensde herhaling. Uiteraard denk je er dan aan om gebruik te maken van een **for-lus**.

```
for (int i = 0; i < lottonummers.Length; i++)
{
    Console.WriteLine(lotto.nummers[i]);
}
```

Nochtans is dit niet de enigste manier, want de for-lus kent een uitbreiding die het gemakkelijker maakt om te kunnen itereren over arrays en verzamelingen (of collections); namelijk de **foreach-lus**.

Syntax:

```
foreach (datatype element in arraynaam)
{
    //statements
}
```

Voorbeeld:

```
foreach (int getal in lottonummers)
{
    Console.WriteLine(getal);
}
```

De variabele `element` neemt tijdens de iteratie één voor één de waarde aan van de elementen uit de array.

Je kan de foreach-lus niet gebruiken als je de index nodig hebt tijdens de iteratie of een andere stapgrootte dan +1 in de lus wilt gebruiken. Bovendien mag je tijdens het doorlopen van de array de inhoud niet wijzigen. In alle andere gevallen is het interessanter om gebruik te maken van de klassieke for-lus.

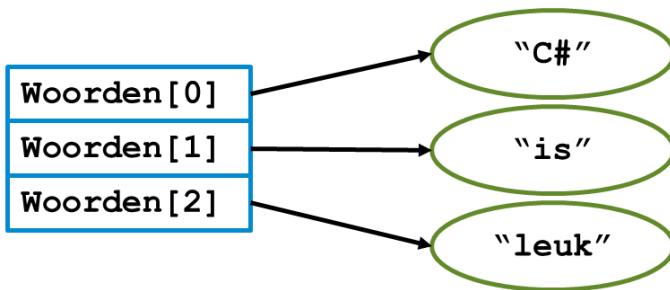
10.5 Arrays van objecten

Een array kan een aantal verwijzingen bevatten naar objecten van hetzelfde type.

Voorbeeld 1: een array van String-objecten:

```
String[] woorden = new String[3];
woorden[0] = "C#";
woorden[1] = "is";
woorden[2] = "leuk";
```

In dit voorbeeld worden er geen drie String-objecten gecreëerd, maar wel een array van drie **referenties naar een String-object**. Daarom moeten we de String-objecten afzonderlijk creëren en de referenties toekennen aan de elementen van de array.



Een array van objecten kan ook meteen geïnitialiseerd worden tijdens de declaratie.

```
String[] woorden = {"C#", "is", "leuk"}
```

Voorbeeld 2: een array van een zelf gedefinieerde klasse Persoon:

```
class Persoon
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }
}

class Program
{
    static void Main()
    {
        Persoon[] personen = new Persoon[2];

        personen[0] = new Persoon { Naam = "John", Leeftijd = 25 };
        personen[1] = new Persoon { Naam = "Jane", Leeftijd = 30 };

        foreach(Persoon persoon in personen)
        {
            Console.WriteLine($"Naam: {persoon.Naam}, Leeftijd: {persoon.Leeftijd}");
        }
    }
}
```

De foreach-lus in dit het voorbeeld geeft de volgende afdruk:

```
Naam: John, Leeftijd: 25
Naam: Jane, Leeftijd: 30
```

10.6 Arrays van arrays

Omdat een array ook een object is, kun je een array van arrays maken. **Een element van een array kan dus zelf een array zijn.** Dit wordt vaak een "multidimensionale array" of een "2D array" genoemd, omdat je een matrix van waarden creëert met meerdere rijen en kolommen. In dit geval hebben alle arrays binnen de array dezelfde lengte.

Wanneer de lengtes van de arrays binnen een array niet gelijk zijn, wordt dit een gekartelde array (of jagged array) genoemd.

Syntax van een 2D-array:

```
datatype[,] naam = new datatype[aantalRijen,aantalKolommen];
```

Voorbeeld van een 2D-array:

```
int[,] matrix = new int[2,3];
```

We gebruiken bij de creatie van 2D-arrays twee indices gescheiden met een komma. De **eerste index** geeft altijd het rangnummer van de **rij** en de **tweede index** geeft de rangnummer van de **kolom**.

	Kolom 1	Kolom 2	Kolom 3
Rij 1	matrix[0,0]	matrix[0,1]	matrix[0,2]
Rij 2	matrix[1,0]	matrix[1,1]	matrix[1,2]

10.6.1 Elementen toevoegen aan een arrays van arrays

10.6.1.1 Elementen toevoegen aan multidimensionale arrays

We kunnen een multidimensionale array afzonderlijk declareren met [,] en vervolgens initialiseren. Het aantal dimensies wordt aangegeven door het aantal komma's tussen []. Een 3D-array zou je bijvoorbeeld moeten declareren met [, ,].

Voorbeeld van en 2D-array:

```
int[,] matrix = new int[2,3];
matrix[0,0] = 1; // eerste rij
matrix[0,1] = 2;
matrix[0,2] = 3;
matrix[1,0] = 4; // tweede rij
matrix[1,1] = 5;
matrix[1,2] = 6;
```

Dit geeft visueel het volgende resultaat:

1	2	3
4	5	6

Multidimensionale arrays kunnen ook meteen geïnitialiseerd worden tijdens de declaratie. Dit doe je door elke array van elementen nog eens apart tussen accolades { } te plaatsen.

```
int[,] matrix =
{
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};
```

10.6.1.2 Elementen toevoegen aan gekartelde arrays

We kunnen een gekartelde array declareren met een [] voor elke dimensie en vervolgens initialiseren door telkens per element een nieuwe array te creëren. Een gekartelde array met drie dimensies zou je bijvoorbeeld moeten declareren met [][][].

Voorbeeld van een gekartelde array met twee dimensies:

```
int[][] jaggedArray = new int[][]  
{  
    new int[3],  
    new int[2],  
    new int[1]  
};
```

Ook gekartelde arrays kunnen meteen geïnitialiseerd worden tijdens de declaratie door elke array van elementen nog eens apart tussen accolades { } te plaatsen. In dat geval ben je niet verplicht een lengte per array in te geven.

```
int[][] jaggedArray = new int[][]  
{  
    new int[] { 1, 2, 3 },  
    new int[] { 5 },  
    new int[] { 6 }  
};
```

Dit geeft visueel het volgende resultaat:

1	2	3
4	5	
6		

10.6.2 Waarden in een array van arrays tonen

10.6.2.1 Waarden in multidimensionale arrays tonen

Je kan een bepaalde waarde in een multidimensionale array tonen door de indexwaarden van elke dimensie tussen vierkante haakjes, gescheiden met een komma, op te geven.

Voorbeeld van en 2D-array:

```
Console.WriteLine(matrix[0,0]);
```

Voor het afdrukken van alle elementen in een multidimensionale array kan je gebruik maken van een **dubbele for-lus** met de methode `GetLength(index)`. De index van de methode `GetLength` verwijst in dit geval niet naar een element, maar naar een dimensie in de array:

- 0 = x-as (rijen).
- 1 = y-as (kolommen).
- 2 = z-as (dieptes).
- ...

```

for(int i = 0; i < matrix.GetLength(0); i++)      //x-as (rijen)
{
    for(int j = 0; j < matrix.GetLength(1); j++) //y-as (kolommen)
    {
        Console.WriteLine($"{matrix[i, j]} ");
    }
    Console.WriteLine(); //nieuwe regel na elke rij
}

```

10.6.2.2 Waarden in gekartelde arrays tonen

Je kan een bepaalde waarde in een gekartelde array tonen door de indexwaarden elke dimensie afzonderlijk tussen vierkante haakjes op te geven.

Voorbeeld van een gekartelde array met twee dimensies:

```
Console.WriteLine(jaggedArray[0][1]);
```

Voor het afdrukken van alle elementen in een gekartelde array kan je gebruik maken van een **dubbele for-lus**.

```

for (int i = 0; i < jaggedArray.Length; i++)      //x-as (rijen)
{
    for (int j = 0; j < jaggedArray[i].Length; j++) //y-as (kolommen)
    {
        Console.WriteLine($"{matrix[i, j]} ");
    }
    Console.WriteLine(); //nieuwe regel na elke rij
}

```

10.7 Eigenschappen van arrays

In deze tabel wordt de **meest gebruikte** eigenschap van arrays weergegeven:

Returntype	Eigenschap	Beschrijving
int	Length	Geeft het aantal elementen in de array terug.

Voorbeeld:

```
int[] lottonummers = {6, 7, 16, 21, 26, 31, 41};
Console.WriteLine(lottonummers.Length);
```

In dit geval is de lengte gelijk aan 7 omdat er 7 elementen in de array voorkomen.

10.8 Methoden van de klasse Array

Je kan methoden van de klasse `System.Array` gebruiken om manipulaties uit te voeren op een array door de array als parameter in te geven.

- i** De meeste van deze methoden zijn overloaded. Dit wilt zeggen dat er meerdere invoermogelijkheden voor parameters zijn die niet in de cursus gedemonstreerd worden. Het loont zeker de moeite om zelf eens alle mogelijkheden te overlopen.

10.8.1 Sort: arrays sorteren

Je kan arrays eenvoudig oplopend sorteren met de methode `Sort(arraynaam)`.

```
string[] kleuren = { "rood", "groen", "blauw", "geel" };
Array.Sort(kleuren);
```

In het voorbeeld worden de kleuren in de stringarray in alfabetische volgorde geplaatst:

```
kleuren = { "blauw", "geel", "groen", "rood" };
```

Dit werkt ook voor arrays van andere datatypes. Bijvoorbeeld, numerieke waarden worden gesorteerd van klein naar groot.

10.8.2 Reverse: arrays omkeren

Je kan de huidige volgorde van een array omkeren met de methode `Reverse(arraynaam)`.

```
int[] scoresOpTien = { 8, 6, 5, 7, 9, 2, 10 };
Array.Reverse(scoresOpTien);
```

In het voorbeeld wordt de volgorde van de karakters in de array omgedraaid:

```
scoresOpTien = { 10, 2, 9, 7, 5, 6, 8 };
```

10.8.3 IndexOf: zoeken naar de eerste index van een element

Je kunt de index opzoeken van het eerste overeenkomende element dat wordt gevonden in een array met de methode `IndexOf(arraynaam, object)`. Als de gezochte waarde niet in de array wordt gevonden, retourneert de methode de waarde -1.

```
char[] tekens = { 'a', 'b', 'c', 'a', 'b', 'c' };
int index = Array.IndexOf(tekens, 'c'); //komt voor in de array: 2
int index = Array.IndexOf(tekens, 'd'); //komt niet voor in de array: -1
```

In dit voorbeeld wordt het karakter 'c' voor het eerst gevonden op index 2, dus deze waarde wordt geretourneerd door de methode. Het karakter 'd' komt echter nergens voor in de array, dus hiervoor wordt de waarde -1 geretourneerd.

10.8.4 LastIndexOf: zoeken naar de laatste index van een element

Je kunt de methode `LastIndexOf(arraynaam, object)` gebruiken om de index van het laatste overeenkomende element in een array te vinden. Ook hier retourneert de methode -1 als het element niet wordt gevonden.

```
char[] tekens = { 'a', 'b', 'c', 'a', 'b', 'c' };
int index = Array.LastIndexOf(tekens, 'c'); //komt voor in de array: 5
int index = Array.LastIndexOf(tekens, 'd'); //komt niet voor in de array:
-1
```

In dit voorbeeld wordt het karakter 'c' voor het eerst gevonden op index 5, dus deze waarde wordt gereturneerd door de methode. Net als bij het vorige voorbeeld levert het karakter 'd' de waarde -1 op omdat deze niet voorkomt in de array.

10.8.5 Clear: Arrays leegmaken

De methode `Clear(arraynaam, index, lengte)` wordt gebruikt om een array volledig of gedeeltelijk te wissen. De gewiste elementen krijgen dan de standaardwaarden `0`, `null` of `false`.

```
//Een array volledig leegmaken
string[] kleuren = { "rood", "groen", "blauw", "geel" };
Array.Clear(kleuren);

//Een array gedeeltelijk leegmaken
int[] scoresOpTien = { 8, 6, 5, 7, 9, 2, 10 };
Array.Clear(scoresOpTien, 2, 3);
```

In het voorbeeld wordt de stringarray kleuren volledig leeggemaakt:

```
kleuren = { null, null, null, null };
```

In de array scoresOpTien worden de eerste 3 elementen vanaf index 2 in de array leeggemaakt:

```
scoresOpTien = { 8, 6, 0, 0, 0, 2, 10 };
```

10.8.6 Copy : een array by value kopiëren

De methode `Copy(bronarray, bestemmingsarray, lengte)` wordt gebruikt om elementen van één array naar een andere array te kopiëren.

```
string[] kleuren = { "rood", "groen", "blauw", "geel" };
string[] rgb = new string[3];
Array.Copy(kleuren, rgb, 3);
```

In het voorbeeld kopieert de nieuwe array `rgb` de eerste 3 elementen van de array `kleuren`:

```
rgb = { "rood", "groen", "blauw" };
```



Je denkt momenteel misschien: "Wat als ik gewoon `rgb = kleuren` doe? Dan kopieer ik toch ook de waarden van de ene array naar de andere?".

Dit is echter geen goed idee. Door de ene array toe te kennen aan een andere array, zullen beide arrays namelijk naar dezelfde plaats in het geheugen verwijzen. Een bewerking zoals de methode `Clear()` op een van de arrays zal daardoor beide arrays volledig leeggemaakt. Gebruik daarom altijd de methode `Copy()` als je de waarde van een array wilt overnemen in een andere array.

10.9 Arrays, tekens en tekenreeksen

Het type `string` is in feite een array van karakters: `char[]`. Daarom is het interessant om dit datatype apart te bespreken en enkele nuttige methoden te laten zien om strings te manipuleren.

Hou er rekening mee dat de volgende methoden die we hier behandelen behoren tot de klasse `String`, en niet tot een array of de klasse `System.Array`.

10.9.1 Split: een string opsplitsen in een array van deelstrings

De klasse String bevat de methode `Split(String regex)` om een string op te splitsen in een array van substrings. Dit gebeurt op basis van de string regex, wat eigenlijk gewoon een karakter is dat dienst doet als scheidingsteken.

```
string zin = "Dit is een voorbeeldzin";
string[] woorden = zin.Split(' ');
```

In dit voorbeeld wordt de string "Dit is een voorbeeldzin" opgesplitst in woorden, waarbij de spatie als scheidingsteken fungert. Het resultaat is een array van substrings:

```
woorden = { "Dit", "is", "een", "voorbeeldzin" };
```

Je kunt trouwens ook meerdere scheidingstekens opgeven. Als je geen scheidingskarakter opgeeft, wordt de spatie als de standaard gehanteerd.

10.9.2 ToCharArray: een string opsplitsen in een array van karakters

De klasse String bevat de methode `ToCharArray()` om een string op te splitsen in een array van karakters.

Voorbeeld:

```
string zin = "C# is leuk";
char[] karakters = zin.ToCharArray();
```

In het voorbeeld wordt elk karakter in de string "C# is leuk" opgesplitst. Het resultaat is een array van karakters:

```
karakters = { 'C', '#', ' ', 'i', 's', ' ', 'l', 'e', 'u', 'k' };
```

Merk op dat de array van karakters een vaste grootte heeft en niet kan worden gewijzigd, omdat strings in C# onveranderlijk zijn.

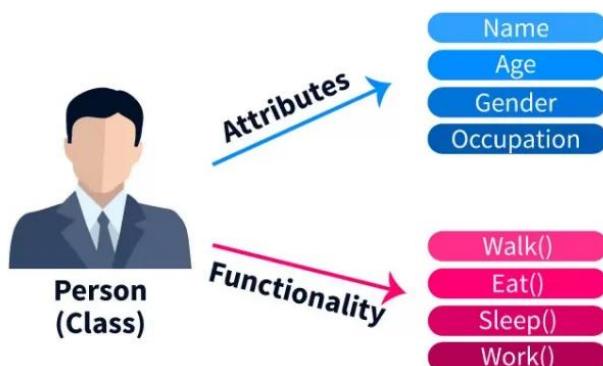


Als je een dynamische collectie van karakters wilt, zou je bijvoorbeeld een `List<char>` kunnen gebruiken. Collections zoals Lists zullen echter in een later hoofdstuk behandelen.

11.1 Wat is een klasse?

Even herhalen ...

Klassen doen dienst als sjablonen of blauwdrukken voor het creëren van objecten. Een klasse definieert de eigenschappen en gedragingen, in de vorm van methoden, die de objecten van dat type zullen hebben. Door gebruik te maken van klassen, kunnen we gestructureerd en herbruikbaar code schrijven.



11.2 Declaratie van de klasse

Om een klasse te maken heb je minimaal het sleutelwoord `class` en een **naam** nodig. De naam van de klasse wordt gewoonlijk geschreven met de **PascalCase notatie**.

Binnen het codeblok van de klasse kan je **gerelateerde methoden en variabelen groeperen**.

Syntax:

```
class KlasseNaam
{
    //Klasse-omschrijving (class body)
}
```

Er kunnen nog meer componenten toegevoegd worden aan de declaratie van de klasse. De volledige syntax ziet eruit als volgt:

```
public [abstract | static] class KlasseNaam : superKlasse, InterfaceNaam
{
    //Klasse-omschrijving (class body)
}
```

Deze componenten zullen pas in de volgende hoofdstukken uitgebreid worden toegelicht.

Voorbeeld van een klasse Auto:

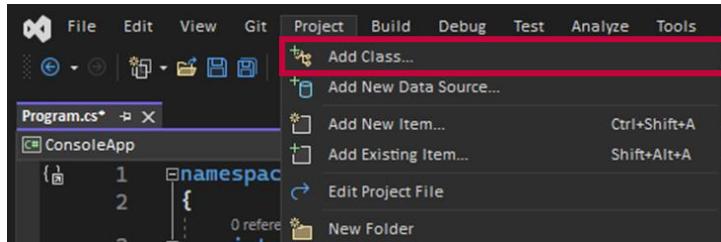
```
class Auto
{
    //inhoud van de klasse
}
```

11.2.1 Klassen declareren in een apart document

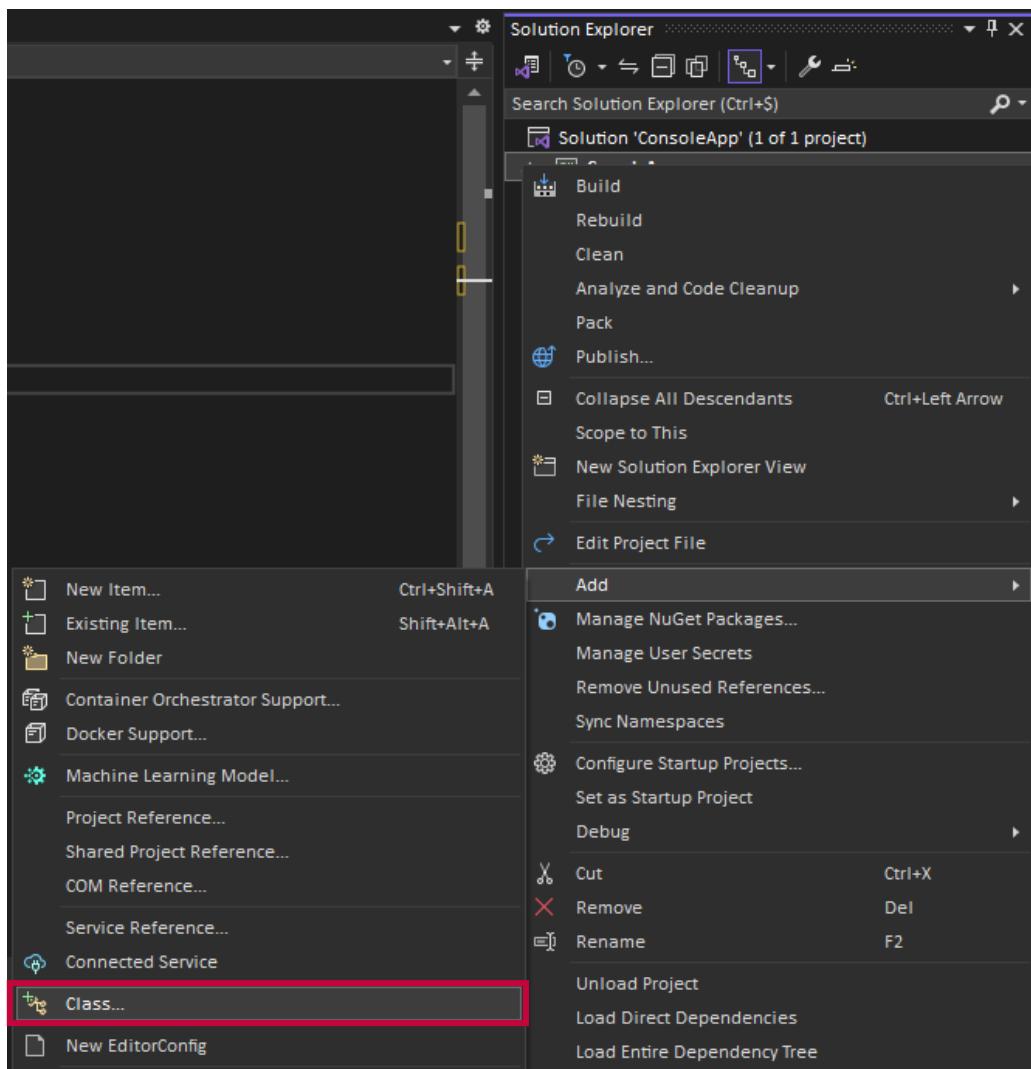
Net als bij methoden, kan je een klasse boven of onder de methode Main() aanmaken. Dit is echter een slechte gewoonte omdat het programma zo minder overzichtelijk wordt. Daarom zal je bij voorkeur altijd een **klasse definiëren in een apart document** in Visual Studio.

Dit kan op de volgende manieren:

- Project > Add Class... > Add New Item > Class > Geef de klasse een naam > Add;



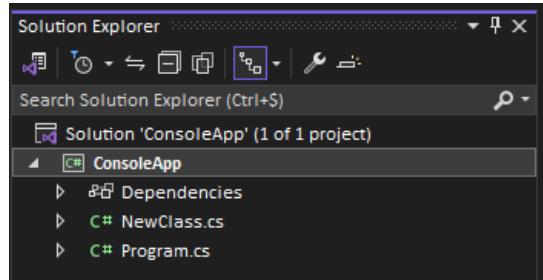
- Solution Explorer > rechterklik op je project > Add > Class... > Add New Item > Class > Geef de klasse een naam > Add.



Vervolgens kom je in de nieuwe klasse terecht. In Visual Studio 2022 krijgt de klasse standaard de access modifier **internal**. Aangezien we in deze cursus altijd in dezelfde assembly zullen werken, mag je deze access modifier negeren.

```
NewClass.cs  Program.cs*
ConsoleApp
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ConsoleApp
8  {
9      internal class NewClass
10  {
11  }
12 }
```

Je kan een overzicht van de klassen in je project vinden in de Solution Explorer.



11.3 De klasse-omschrijving (body)

Tussen de accolades van de klasse bevindt zich de omschrijving (of class body) van de klasse. De klasse-omschrijving kan **velden**, **eigenschappen**, **methoden** en **constructors** bevatten.

Voorbeeld van een klasse Rechthoek:

```
public class Rechthoek //class header
{
    //Velden (attributen / instantievariabelen)
    private int hoogte;
    private int breedte;

    //Eigenschappen (properties)
    public int Hoogte { get; set; }
    public int Breedte { get; set; }

    //Constructors
    public Rechthoek(int breedte, int hoogte)
    {
        this.breedte = breedte;
        this.hoogte = hoogte;
    }

    //Methoden
    public int Oppervlakte()
    {
        return Hoogte * Breedte;
    }
}
```

11.4 Access modifiers

De access modifier geeft de **zichtbaarheid** (of het toegangsniveau) van de methoden en eigenschappen van een klasse aan. Het bepaalt dus eigenlijk in welke mate een methode of eigenschap door de buitenwereld kan worden gebruikt.

Access modifiers	Beschrijving
private	De code is alleen zichtbaar en toegankelijk binnen dezelfde klasse.
protected	De code is alleen zichtbaar en toegankelijk binnen de klasse zelf en alle zijn subklassen.
public	De code is zichtbaar en toegankelijk van buiten de klasse.
internal	De code is toegankelijk binnen dezelfde assembly (een verzameling van gerelateerde codebestanden en bronnen) waarin het is gedeclareerd.

Eigenschappen en methoden zonder een access modifier worden standaard als private beschouwd. Toch wordt het ten strengste aangeraden om **ALTIJD een access modifier voor een methode of eigenschap te plaatsen!**

-  Er bestaan nog andere access modifiers zoals protected, internal en private protected. In deze cursus zullen we ons echter beperken tot public, protected en private.

11.4.1 Waarom private?

Waarom zou je bepaalde zaken **private** maken?

Allereerst verwijzen we hiervoor even terug naar het basisconcept **Inkapseling** (of Encapsulation). Het gebruik van private zorgt er namelijk voor dat methoden en gegevens alleen toegankelijk zijn binnen de klasse zelf. **De interne werking van een klasse wordt zo afgeschermd van externe code.** Bijgevolg kan alleen via publieke methoden en eigenschappen gecommuniceerd worden met de afgeschermd gegevens. Hierdoor krijgt de klasse **de volledige controle** over hoe de gegevens worden gebruikt. Kortom, de access modifier private voorkomt onbedoelde of ongeoorloofde wijzigingen van buitenaf en bevordert zo de **veiligheid** van jouw code.

Het volgende voorbeeld toont hoe je in een klasse kan communiceren met afgeschermd onderdelen:

```
class Mens
{
    public void Praat()
    {
        Console.WriteLine("Ik ben een mens!");
        VertelGeheim();
    }

    private void VertelGeheim()
    {
```

```
        Console.WriteLine("Ik ben verliefd op een klasgenoot.");
    }
}
```

In het bovenstaande voorbeeld is de methode `VertelGeheim()` een private methode die je niet rechtstreeks kan aanroepen. Toch is het mogelijk om via de methode `Praat()` de methode `VertelGeheim()` aan te roepen. Dat komt omdat de methode `VertelGeheim()` alleen private is voor de buitenwereld, maar niet binnin de klasse. Een andere methode van deze klasse kan deze dus wel aanroepen.

Nu creëren we een instantie van de klasse `Mens` in de methode `Main()` en we laten hem praten:

```
Mens vincent = new Mens();
vincent.Praat();
```

Dit geeft de volgende uitvoer:

```
Ik ben een mens!
Ik ben verliefd op een klasgenoot.
```

11.5 Velden

Een veld (of attribuut) is net als een **variabele**, maar dan gedefinieerd **op klasse niveau**. Deze variabelen kunnen **waarden** bevatten die **voor elk object (of instantie) verschillend** zijn. Omwille van die reden worden ze ook wel **instantievariabelen** genoemd.

Voorbeeld van een veld dat de kleur van een Auto-object beschrijft:

```
class Auto
{
    private string kleur = "rood";    //Veld (of attribuut)

    public void BeschrijfKleur()
    {
        Console.WriteLine($"De kleur van de auto is {kleur}.");
    }
}
```

Velden worden gewoonlijk bovenin de klasse gedeclareerd met de **access modifier private**. Het is weliswaar mogelijk om ze protected of public te maken, maar dit mag je nooit doen! Je wilt namelijk voorkomen dat de buitenwereld waarden geeft aan velden waardoor de werking van de klasse stuk zou gaan. Als je toch de interne staat van een veld wilt aanpassen, doe je dit via eigenschappen en methoden.



Velden worden geschreven met de **camelCase** notatie.

11.6 Eigenschappen

Een eigenschap (of property) is een speciaal lid van een klasse waarmee we **op een gecontroleerde manier de instantievariabelen van objecten in- en uitlezen**. Je maakt dus nooit rechtstreeks

aanpassingen in een veld, maar je maakt hiervoor gebruik van eigenschappen. Bekijk het een beetje als een toegangspoort met een bewaker die controleert of alles wat binnen en buiten gaat veilig en correct verloopt.

Aangezien eigenschappen ons de toegang verlenen tot instantievariabelen, worden ze altijd gedeclareerd met de **access modifier public**.



Als je een eigenschap gebruikt om een instantievariabele naar buiten beschikbaar te stellen, dan is het een goede gewoonte om **dezelfde naam als dat veld** te nemen, maar dan geschreven met de **PascalCase** notatie.

Bijvoorbeeld: het veld `kleur` en de eigenschap `Kleur`.

Voorbeeld van een veld en zijn bijhorende property:

```
private string kleur = "rood"; //geschreven met een kleine letter

public string Kleur //geschreven met een hoofdletter
{
    get { return kleur; }
    set { kleur = value; }
}
```

11.6.1 Getters en setters

Properties herken je aan de sleutelwoorden `get` (lezen) en `set` (schrijven) in een klasse:

- **get** geeft de uitgelezen waarde terug met het sleutelwoord `return`;
- **set** leest een waarde in door het sleutelwoord `value` toe te kennen aan het overeenkomstige veld.

Getters en setters bieden de mogelijkheid om een **extra niveau van controle** over de toegang tot de gegevens toe te voegen, en dit in de vorm van **validatie of extra logica**. Indien een waarde niet voldoet aan de opgestelde voorwaarden, kan de waarde namelijk worden aangepast of zelfs niet worden toegewezen aan een veld. Het is aan de programmeur om te bepalen hoe complex de code binnen `set` (en `get`) moet zijn.

Voorbeeld van een property om je leeftijd in te geven **met toegangscontrole**:

```
private int leeftijd;

public int Leeftijd
{
    get { return leeftijd; }
    set
    {
        if(value >= 0)
            leeftijd = value;
    }
}
```

In het bovenstaande voorbeeld accepteert de property Leeftijd alleen positieve waarden omdat een leeftijd niet negatief kan zijn.

Een bijkomend voordeel van properties is dat ze de mogelijkheid bieden om read-only of write-only toegang tot gegevens in te stellen.

11.6.2 Overzicht soorten properties

Er zijn 2 soorten properties in C# met voorbeelden die exact hetzelfde doen:

Full properties	Autoproperties
<p>Deze stijl van properties verplicht ons veel code te schrijven in ruil voor de volledige controle over wat er gebeurt. Hiervoor heb je ook een achterliggende private instantievariabele nodig</p> <pre>private string kleur; public string Kleur { get { return kleur; } set { kleur = value; } }</pre>	<p>Met deze stijl kan je snel properties schrijven met weinig code, maar je hebt dan ook weinig controle over wat er gebeurt. Hiervoor heb je geen achterliggende private instantievariabele nodig omdat C# deze voor ons achter de schermen aanmaakt.</p> <pre>public string Kleur { get; set; }</pre>

In de methode Main() gebruik je de beide soorten properties op dezelfde manier:

```
Auto auto = new Auto();
auto.Kleur = "blauw";                                //set
Console.WriteLine($"De kleur is {Auto.Kleur}.")      //get
```

11.6.2.1 Varianten op full properties

Een property moet niet altijd een getter en een setter bevatten. Dit maakt het mogelijk om verschillende variaties van full properties te schrijven:

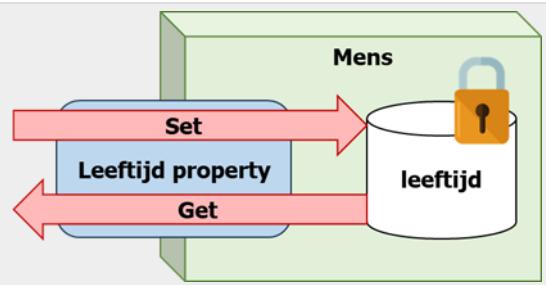
1. Gewone full property: get + set

Een gewone full property kan zowel informatie uit een object halen (get) als informatie in een object plaatsen (set).

Voorbeeld:

```
private int leeftijd;

public int Leeftijd
{
    get { return leeftijd; }
    set { leeftijd = value; }
}
```



2. Write-only property: set

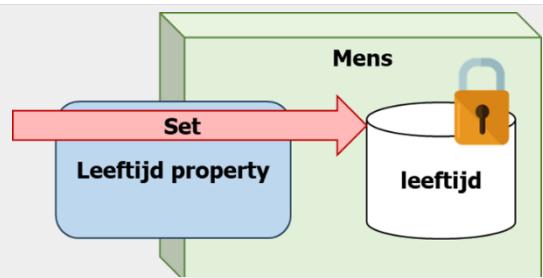
Met een write-only property (of "alleen-schrijven" eigenschap) plaats je informatie in een object dat niet mag worden uitgelezen.

Dit kan handig zijn in situaties waarin gebruikers direct toegang hebben tot bepaalde gegevens om waarden in te stellen, maar dezelfde gegevens niet mogen uitlezen.

Denk bijvoorbeeld aan een klasse waarin je een wachtwoord opslaat. Je wilt niet dat gebruikers het wachtwoord kunnen lezen, maar je wilt wel dat ze het kunnen instellen en wijzigen.

Voorbeeld:

```
private int Leeftijd;  
  
public int Leeftijd  
{  
    set { Leeftijd = value; }  
}
```



3. Read-only property: get

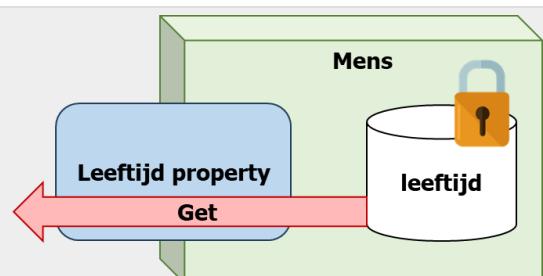
Met een read-only property (of "alleen-lezen" eigenschap) kan je alleen informatie in een object uitlezen, maar niet wijzigen.

Dit kan handig zijn in situaties waarin je wilt voorkomen dat de waarde van een eigenschap wordt gewijzigd.

Denk bijvoorbeeld aan een wiskundige klasse waarin je de (constante) waarde van Pi (π) alleen wilt kunnen uitlezen.

Voorbeeld:

```
private int Leeftijd;  
  
public int Leeftijd  
{  
    get { return Leeftijd; }  
}
```



4. Read-only property met private set: get + private set

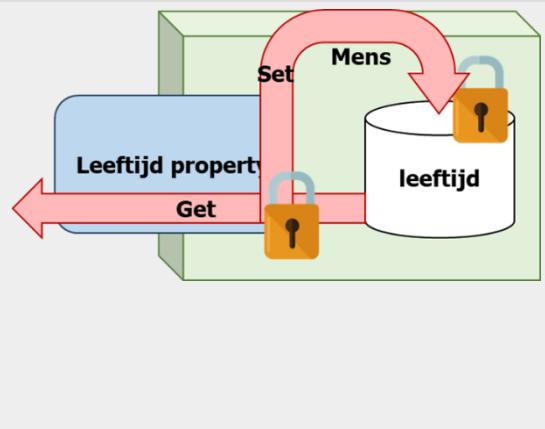
Je kan ook een read-only property definiëren met een `private set` om ervoor te zorgen dat de eigenschap alleen binnenin de klasse zelf gewijzigd kan worden, en niet van buitenaf.

Code die de set van buiten nodig heeft zal een fout geven ongeacht of deze geldig is of niet.

Voorbeeld:

```
private int Leeftijd;

public int Leeftijd
{
    get { return Leeftijd; }
    private set
    {
        if(value >= 0)
            Leeftijd = value;
    }
}
```



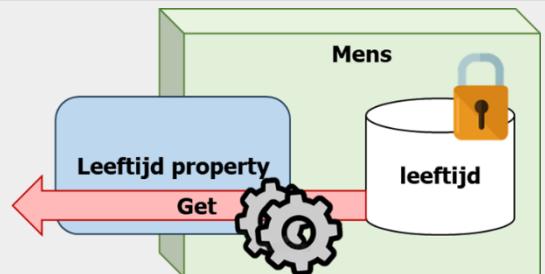
5. Read-only property die data transformeert: get

Je kan nieuwe properties genereren voor gegevens die van andere properties komen, en niet noodzakelijk uit een instantievariabele. Je neemt als het ware de waarde van één of meerdere properties en je maakt er een nieuwe property mee.

Denk bijvoorbeeld aan een klasse die reeds properties bevat van de voor- en achternaam van een persoon. Aan de hand van de waarden in deze properties maak je een nieuwe read-only property aan om de volledige naam te tonen. Op dezelfde manier zou je in dit voorbeeld een property voor een e-mailadres kunnen maken.

Voorbeeld:

```
public string Voornaam {get; set;}
public string Achternaam {get; set;}
public string VolledigeNaam
{
    get { return $"{Voornaam}{Achternaam}"; }
}
```



11.6.2.2 Autoproperties

Automatische eigenschappen (of autoproperties) maken het mogelijk om snel properties te schrijven **zonder** dat je de **achterliggende instantievariabele** moet beschrijven. De instantievariabele is namelijk onzichtbaar in autoproperties en kan dus onmogelijk worden gebruikt.

Bovendien kan je autoproperties **enkel gebruiken als er geen extra logica in de property aanwezig moet zijn**, zoals het geboortejaar controleren op negatieve waarden. Indien je wel bepaalde controles of transformaties wilt opnemen in je property, zal je dit moeten doen met een full property.

Gelukkig kan je wel **beginwaarden** geven aan autoproperties door de waarde erachter te schrijven.

```
public string Leeftijd { get; set; } = 18;
```

Daarnaast kan je wel **read-only autoproperties** definiëren als ze een `private set` bevatten.

```
public string Voornaam { get; private set; }
```

Zonder een private set is een beginwaarde bij read-only autoproperties verplicht. Deze waarde kan achteraf niet meer worden aangepast.

```
public string Voornaam { get; } = "Kristof";
```

11.7 Methoden

Zoals je weet zijn methoden codeblokken die **specifieke taken uitvoeren**. Je kan methoden ook toevoegen aan klassen waarbij ze (meestal) verbonden zijn aan objecten. Zo definiëren ze het gedrag van objecten die tot die klasse behoren.

Een methode bestaat altijd uit een returntype (of `void`) en een methodenaam met daarachter ronde haakjes. Tussen de ronde haakjes kan je extra waarden, beter gekend als parameters, meegeven aan de methode.

Voorbeeld van een instantie-methode:

```
class Mens
{
    public void Praat()
    {
        Console.WriteLine("Ik ben een mens!");
    }
}
```

Tot nu toe heb je geleerd dat je methoden buiten de methode `Main()` moet declareren met het sleutelwoord `static`. Voor methoden van instanties doe je dit echter niet! **Je vervangt static dan met een access modifier zoals public** zodat de buitenwereld deze methode op het object kan aanroepen.



Er zijn situaties waarin je wel het sleutelwoord `static` voor een methode in een klasse plaatst. In het volgende hoofdstuk zal je ontdekken wat `static` juist doet, alsook hoe en (vooral) wanneer je het moet gebruiken.

11.7.1 De scope van variabelen

Een methoden kunnen we gebruik maken van twee soorten variabelen:

- Instantievariabelen (velden),
- lokale variabelen.

Instantievariabelen worden bovenaan in de klasse gedeclareerd en zijn bijgevolg **geldig in heel de klasse**.

Binnen het codeblok van een methoden kunnen we ook **lokale variabelen** declareren. Deze variabelen kunnen alleen gebruikt worden **binnen het bereik (of scope) van dit codeblok**. Bovendien moeten ze expliciet geïnitialiseerd zijn alvorens je ze kan gebruiken. Doe je dit niet, dan

zal de compiler een foutmelding geven. Verder mag een lokale variabele dezelfde naam hebben als een instantievariabele. In dit geval wordt de instantievariabele verborgen door de lokale variabele in de methode.

Voorbeeld van een instantie- en een lokale variabele met dezelfde naam:

```
class Nummers
{
    private int waarde = 5;           //instantievariabele

    public void EenMethode()
    {
        int waarde = 10;             //lokale variabele
        Console.WriteLine(waarde);
    }
}
```

11.7.2 Gegevens doorgeven

Bij het aanroepen van een methode kan men **via parameters** een aantal **gegevens doorgeven** aan de methode. Eigenlijk kan je parameters **beschouwen als lokale variabelen** die onmiddellijk gedeclareerd en geïnitialiseerd worden tijdens het aanroepen van de methode. Het bereik van de parameters is namelijk beperkt tot het codeblok van de methode.

Voorbeeld van parameterwaarden die worden toegekend aan instantievariabelen:

```
class Rechthoek
{
    private int x;
    private int y;

    public void SetPositie(int xPos, int yPos) //2 parameters
    {
        x = xPos;
        y = yPos;
    }
}
```

In dit voorbeeld worden de ingegeven waarden van de parameters xPos en yPos doorgegeven aan de velden x en y.

11.7.2.1 Gegevens doorgeven aan gelijknamige velden met this

Ook parameters en instantievariabelen kunnen dezelfde naam hebben. Alleen kan je op deze manier geen waarde doorgeven van een parameter naar een gelijknamige instantievariabele (bv. x = x). C# zal namelijk denken dat je een variabele aan zichzelf toekent en bijgevolg een standaardwaarde toekennen zoals 0, null of false (afhankelijk van het datatype).

Om toch een referentievariabele te kunnen bereiken, moet je het sleutelwoord **this** voor de instantievariabele plaatsen.

Voorbeeld van parameterwaarden die worden toegekend aan gelijknamige instantievariabelen:

```
class Rechthoek
{
    private int x;
    private int y;

    public void SetPositie(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Vanuit de methode Main() kunnen waarden als volgt worden doorgegeven:

```
Rechthoek eenRechthoek = new Rechthoek();
eenRechthoek.SetPositie(2, 4);
```

Via de methode SetPositie() hebben de instantievariabelen nu de volgende waarde gekregen:
x = 2 en y = 4.

11.7.3 Waarden teruggeven

Zoals je weet heeft iedere methode de mogelijkheid om een bepaalde waarde terug te geven aan het object dat de methode aanroeft.

We gaan verder met het voorbeeld van de klasse Rechthoek. We voegen aan de klasse een methode GetOppervlakte() toe om de oppervlakte van Rechthoek-objecten te berekenen en retourneren.

```
public double GetOppervlakte()
{
    return x * y;
}
```

Vervolgens roepen we de methode GetOppervlakte() aan in de methode Main():

```
double oppervlakte = rechthoek.GetOppervlakte(); //2 * 4 = 8
```

In dit voorbeeld krijgt de variabele oppervlakte de waarde 8.

11.7.4 Method (name) overloading

Ook in klassen kan je **meerdere methoden dezelfde naam** geven op voorwaarde dat er een duidelijk onderscheid is in het aantal en/of het datatype van de parameters per methode. Deze techniek ken je trouwens al als method (name) overloading.

Voorbeeld van de methode Optellen met meerdere parameters die van elkaar verschillen:

```
class Calculator
{
    //2 parameters (int)
    public int Optellen(int a, int b)
    {
        return a + b;
    }

    //3 parameters (int)
    public int Optellen(int a, int b, int c)
    {
        return a + b + c;
    }

    //2 parameters (double)
    public double Optellen(double a, double b)
    {
        return a + b;
    }
}
```

11.7.5 Objecten als parameter

Aangezien een klasse eigenlijk een nieuwe vorm van datatype is, kan je ook **instanties van een klasse meegeven als een parameter**, en dit op dezelfde manier als je met waardetypen.



Dit is weliswaar niet mogelijk met methoden die static zijn! De reden hiervoor leer je in het volgende hoofdstuk.

Wanneer je een object doorgeeft aan een methode, geef je eigenlijk niet het object zelf door, maar een verwijzing naar dat object. Hierdoor zullen alle wijzigingen die door methoden aan het object worden aangebracht, doorwerken in het originele object.

Laten we dit verduidelijken met een Auto-object als voorbeeld. Als je een methode gebruikt om bijvoorbeeld de kleur van de auto te veranderen, wordt de kleur direct aangepast op de originele auto die je hebt doorgegeven. Wanneer je later in je programma opnieuw met het originele object werkt, zal de kleur gewijzigd zijn door de eerdere methodeaanroep. Het is dus belangrijk om te begrijpen wat er met je object gebeurt wanneer je het doorgeeft aan een methode.

Door parameters van instanties kunnen objecten met elkaar communiceren en eventueel een resultaat terugkrijgen. Dit concept wordt nog duidelijker wanneer je oefeningen maakt waarbij bijvoorbeeld een Bankrekening-object geld overschrijft naar een ander Bankrekening-object.

Voorbeeld 1: een methode AttackPokemon() in een erg vereenvoudigde klasse Pokemon om een ander Pokemon-object aan te vallen:

```
public class Pokemon
{
    public string Name { get; private set; }
    public int HP { get; private set; }           //hitpoints (levenspunten)
    public int Attack { get; private set; }
    public int Defense { get; private set; }

    public Pokemon(string name, int hp, int attack, int defense)
    {
        Name = name;
        HP = hp;
        Attack = attack;
        Defense = defense;
    }

    public void AttackPokemon(Pokemon target) //object als parameter
    {
        int damage = Attack - target.Defense;
        if (damage < 0)
            damage = 0;
        target.TakeDamage(damage);
        Console.WriteLine($"{Name} attacks {target.Name} and does {damage} damage!");
        Console.WriteLine($"{target.Name} now has {target.HP} HP.");
    }

    private void TakeDamage(int damage)
    {
        HP -= damage;
        if (HP < 0)
            HP = 0;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Pokemon pikachu = new Pokemon("Pikachu", 100, 50, 20);
        Pokemon bulbasaur = new Pokemon("Bulbasaur", 120, 40, 30);

        bulbasaur.AttackPokemon(pikachu);
    }
}
```

De klasse Pokemon heeft eigenschappen die geïnitialiseerd worden via de constructor.

De methode AttackPokemon() neemt een ander Pokemon-object als parameter en berekent de schade door de Attack waarde van de aanvallende Pokémon te verminderen met de Defense waarde van de verdedigende Pokémon. Als de berekende schade negatief is, wordt deze op nul gezet.

Vervolgens wordt de methode TakeDamage() aangeroepen om de schade af te trekken van de HP (hitpoints) van de verdedigende Pokémon. Deze methode is private zodat het niet kan worden aangeroepen zonder dat er een aanval gebeurt.

Tenslotte wordt een bericht afgedrukt dat aangeeft welke Pokémon aanvalt, hoeveel schade er wordt gedaan en hoeveel HP de andere Pokémon daarna nog heeft.

De code in de methode Main() geeft de volgende uitvoer:

```
Bulbasaur attacks Pikachu and does 20 damage!
Pikachu now has 80 HP.
```

Dit voorbeeld toont hoe bulbasaur de pikachu aanvalt, waarbij de schade wordt berekend als 40 (Attack) - 20 (Defense) = 20. Na de aanval heeft pikachu nog 80 HP over (100 - 20).

Voorbeeld 2: een methode VoegProductToe() van om producten toe te voegen aan een winkelmandje:

```
class Product
{
    public string Naam { get; set; }
    public double Prijs { get; set; }
}

class Winkelmandje
{
    private List<Product> producten = new List<Product>();

    public void VoegProductToe(Product product) //object als parameter
    {
        producten.Add(product);
    }

    public void ToonInhoud()
    {
        Console.WriteLine("Inhoud van het winkelmandje:");
        foreach (var item in producten)
        {
            Console.WriteLine($" - {item.Naam}: {item.Prijs:0.00} EUR");
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Product iphone = new Product();
        iphone.Naam = "iPhone 15";
        iphone.Prijs = 969;
        Product airpods = new Product();
        airpods.Naam = "Airpods 2";
        airpods.Prijs = 125;

        Winkelmandje winkelmandje = new Winkelmandje();
        winkelmandje.VoegProductToe(iphone);
        winkelmandje.VoegProductToe(airpods);
        winkelmandje.ToonInhoud();
    }
}

```

In dit voorbeeld hebben we 2 klassen: Product en Winkelmandje.

We hebben een methode VoegProductToe() in de klasse Winkelmandje gecreëerd die een instantie van de klasse Product als parameter accepteert. Op deze manier kan een Product-object worden toegevoegd aan de List van producten van de klasse Winkelmandje.

- i** Een List (of lijst) is zoals een array waarvan de lengte niet op voorhand moet worden ingegeven. Dit maakt het veel geschikter om dit voorbeeld te illustreren aangezien we nu naar hartenlust producten kunnen toevoegen aan het winkelmandje, zonder op voorhand te moeten weten hoeveel producten je in totaal zal toevoegen aan het winkelmandje.

In het hoofdstuk over collectieklassen zal je hiermee leren werken.

In de methode Main() maken we vervolgens nieuwe producten aan (een iPhone en Airpods) en een nieuw winkelmandje. We voegen vervolgens de producten toe aan het winkelmandje met behulp van de methode VoegProductToe(). Ten slotte tonen we de inhoud van het winkelmandje om te controleren of het product correct is toegevoegd.

De code in de methode Main() geeft de volgende uitvoer:

Inhoud van het winkelmandje:
- iPhone 15: 969,00 EUR
- Airpods 2: 125,00 EUR

11.8 Constructors

In het hoofdstuk over Objecten heb je reeds geleerd dat elk object, en bijgevolg elke klasse, een constructor heeft. Een constructor is een **unieke methode** binnen een klasse die **éénmalig** wordt aangeroepen om een **nieuw object** van die klasse te **creëren en initialiseren**. In een constructor kun je namelijk de beginwaarden van eigenschappen van een object instellen wanneer een nieuw object wordt gemaakt.

Syntax:

```
KlasseNaam ObjectNaam = new KlasseNaam(parameters);
```

Voorbeelden van constructors bij de creatie van objecten:

```
Random willekeurig = new Random(10);
StringBuilder builder = new StringBuilder();
```

Zoals je merkt heeft een constructor **geen retourtype** en altijd **dezelfde naam als de klasse** waarin het zich bevindt. Aangezien je het moet kunnen aanroepen buiten de klasse, krijgt het bij voorkeur de access modifier **public**, maar dit is niet verplicht.

Syntax:

```
[public | protected | private] KlasseNaam(parameters)
{
    //Initialisatie van het object
}
```

11.8.1 Constructor overloading

Net als bij methoden kan een klasse **meerdere constructors** hebben die verschillen in het aantal en het type parameters. Dit noemt men constructor overloading. Afhankelijk van de parameters zal de compiler de juiste constructor gebruiken.

Een klasse moet altijd beschikken over **minstens één constructor**. Indien je een constructor niet expliciet definieert in de klasse, voorziet C# zelf op de achtergrond een **default constructor** zonder parameters.

Voorbeelden van verschillende soorten constructors van de klasse Auto:

```
class Auto
{
    private string merk;
    private int jaar;

    //Zelfgeschreven default constructor zonder parameters
    public Auto()
    {
        merk = "Onbekend";
        jaar = 2024;
    }

    //Overloaded constructor met 2 parameters
    public Auto(string merk, int jaar)
    {
        this.merk = merk;
        this.jaar = jaar;
    }
}
```

Nieuwe objecten worden gecreëerd in de methode Main() met de aanroep van constructors:

```
Auto standaardAuto = new Auto();      //Onbekend merk van het jaar 2024
Auto auto = new Auto("BMW", 2020);
```

11.8.2 Soorten constructors

Zoals je in de klasse Auto al hebt kunnen zien zijn er twee soorten constructors:

- Default constructor,
- Overloaded constructor(s).

11.8.2.1 Default constructor

De default constructor is een constructor **zonder parameters**. Je krijgt in het begin standaard een lege default constructor voor je klasse wanneer je een nieuwe klasse aanmaakt. Deze is voor ons onzichtbaar en bijgevolg kan je deze niet aanpassen.

Van zodra je echter beslist om zelf één of meerdere constructor te schrijven, verdwijnt de oorspronkelijke default constructor! Als je de klasse toch nog van een default constructor wil voorzien, zal je deze zelf moeten schrijven. Je kan dan zelf bepalen wat er in de default constructor moet gebeuren. Daarnaast moet een default constructor **altijd public** zijn.

Voorbeeld van een default constructor waarbij we zelf de standaardwaarden bepalen:

```
class Persoon
{
    //Eigenschappen
    public string Naam { get; set; }
    public int Leeftijd { get; set; }

    //Default constructor met standaardwaarden
    public Persoon()
    {
        Naam = "Onbekend";
        Leeftijd = 18;
    }
}
```

De creatie van een Persoon-object met behulp van een default constructor:

```
Persoon standaardPersoon = new Persoon();
```

11.8.2.2 Overloaded constructors

Met overloaded constructors kan je één of meerdere constructors met **verschillende parameters** hebben in dezelfde klasse. Zo krijgen gebruikers de mogelijkheid om op verschillende manieren objecten te initialiseren.

Voorbeeld van overloaded constructors in de klasse Persoon:

```
class Persoon
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }

    //Default constructor zonder parameters
    public Persoon()
    {
        Naam = "Onbekend";
        Leeftijd = 18;
    }

    //Overloaded constructor met naam als parameter
    public Persoon(string naam)
    {
        Naam = naam;
        Leeftijd = 18;
    }

    //Overloaded constructor met naam en leeftijd als parameters
    public Persoon(string naam, int leeftijd)
    {
        Naam = naam;
        Leeftijd = leeftijd;
    }
}
```

De creatie van een Persoon-object met behulp van een overloaded constructor met 2 parameters:

```
Persoon eenPersoon = new Persoon("Gert", 21);
```

Het object eenPersoon heeft tijdens zijn creatie de naam Gert en de leeftijd van 21 jaar als eigenschappen gekregen via een constructor.

11.8.3 Constructor chaining

Hopelijk heb je in het vorige voorbeeld van de klasse Persoon gemerkt dat **in elke constructor quasi dezelfde code** terugkomt. Dat is toch veel extra werk. Om te voorkomen dat we steeds dezelfde toewijzingen moeten schrijven in elke constructor, gaan we constructors aan elkaar koppelen met een techniek genaamd constructor chaining.

Met constructor chaining is het mogelijk om een constructor een **andere constructor** in dezelfde klasse te laten **aanroepen** met de nodige (standaard) parameters. Hiervoor gebruik je het sleutelwoord **this**. Concreet houdt dit in dat we zelf kiezen welke parameters we meegeven aan de constructor. De compiler zal vervolgens beslissen welke constructor nodig is voor de initialisatie van het object.

Voorbeeld van een herschreven klasse Persoon met constructor chaining:

```
class Persoon
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }

    public Persoon() : this("Onbekend", 18)
    { }

    public Persoon(string naam) : this(naam, 18)
    { }

    public Persoon(string naam, int leeftijd)
    {
        Naam = naam;
        Leeftijd = leeftijd;
    }
}
```

In dit voorbeeld gaan alle constructors de onderste overloaded constructor gebruiken omdat deze voor elke parameter een waarde toekent aan de overeenkomstige eigenschappen. Bij de andere constructors (met minder parameters) wordt tussen haakjes een literal toegevoegd als standaardwaarde voor elke parameter die niet wordt ingevuld. Merk op dat je de naam van de aanwezige parameter(s) letterlijk overneemt tussen de haakjes.

Bijvoorbeeld `Persoon(string naam)`.

Op deze manier hoeft je de eigenlijke **code** voor het initialiseren van het object maar **op één plaats** te schrijven. Dit maakt de code compacter, overzichtelijker en het vermindert het risico op fouten als de code later moet worden aangepast.

11.8.4 Fouten voorkomen met behulp van constructors

Bekijk de onderstaande klasse. Wat is hier het probleem?

```
class Breuk
{
    public int Noemer {get; set;}
    public int Teller {get; set;}

    public double BerekenBreuk()
    {
        return (double) Teller / Noemer;
    }
}
```

Als we de methode `BerekenBreuk()` aanroepen zonder Noemer eerst een waarde te geven, zal dit bij het uitvoeren van het programma een `DivideByZeroException` opleveren. Dit komt doordat een numerieke waarde standaard gelijk is aan 0, en delen door 0 is onmogelijk.

Via een overloaded constructor kunnen we dit soort fouten voorkomen omdat we de gebruiker zo verplichten om eerst een waarde aan de Noemer en Teller toe te kennen. Bovendien kunnen op deze manier geen Breuk-objecten met de default constructor aangemaakt worden omdat deze niet meer bestaat van zodra je zelf een constructor hebt geschreven.

```
Breuk eenBreuk = new Breuk();
```

Daarnaast kunnen we de eigenschap Noemer met een full property voorzien van een foutencontrole op de waarde 0. De gebruiker kan deze immers nog altijd via een constructor ingeven. Door de ingevoerde waarde 0 te veranderen in de waarde 1, zal deze fout niet meer kunnen voorkomen.

Dit geeft het volgende resultaat:

```
class Breuk
{
    public Breuk(int tellerIn, int noemerIn)
    {
        Teller = tellerIn;
        Noemer = noemerIn;
    }

    public int Teller { get; private set; }

    private int noemer;
    public int Noemer
    {
        get { return noemer; }
        private set
        {
            if (value != 0)
                noemer = value;
            else
                noemer = 1; //of werp een Exception op
        }
    }
}
```

Kortom, de gebruiker is nu verplicht om Breuk-objecten als volgt aan te maken:

```
Breuk eenBreuk = new Breuk(21, 7);
```


12.1 Het sleutelwoord static

In een klasse zorgt het **sleutelwoord static** ervoor dat **variabelen en methoden gedeeld** kunnen worden **over alle objecten van die klasse**. Statische leden (of static members) zijn dus niet **gekoppeld** aan een specifiek object, maar **aan de klasse zelf**.

Daarnaast kan je **elk statisch lid van een klasse aanroepen zonder** dat je **een instantie van die klasse** nodig hebt. Dit is handig voor wanneer het (logisch gezien) totaal geen nut heeft om van een klasse objecten aan te maken. Denk bijvoorbeeld maar aan de klassen Math of Console waarvan je tot nu toe regelmatig methoden hebt gebruikt, zonder dat je er een instantie van hebt moeten aanmaken. Dit zijn namelijk zogenaamde “pure” klassen van methoden, ook wel **hulpklassen** (of helper classes) genoemd, waarbij **de klasse zelf wordt gebruikt als een object**.

Kortom, met het sleutelwoord static geven we aan dat het lid bij de klasse hoort en niet bij instanties van die klasse. In het algemeen zal je statische leden minder vaak nodig hebben dan niet-statische leden. Al zullen er bepaalde situaties zijn waarin je er dankbaar gebruik van kan maken.

12.2 Statische leden

Statische leden worden gedefinieerd in een klasse door het sleutelwoord **static** tussen de access modifier en het datatype te plaatsen.

Syntax van een statisch veld:

```
private static datatype veldnaam;
```

Statische methoden kan je rechtstreeks aanroepen op de naam van de bijkomende klasse.

Syntax van een statische methode:

```
[public | protected | private] static returntype MethodeNaam()
{
    //statements
}
```

Je kan zelfs een klasse static maken om te voorkomen dat er objecten van de klasse aangemaakt kunnen worden.

Syntax van statische klasse:

```
public static class KlasseNaam
{
    //Klasse-omschrijving (class body)
}
```

12.2.1 Klasse-variabelen

Klasse-variabelen (of static fields) zijn variabelen die **gemeenschappelijk** zijn **voor alle objecten** van dezelfde klasse. De waarde van een klasse-variabele is dus hetzelfde voor alle objecten.

Voorbeeld van de klasse Leerling **met een static field**:

```
class Leerling
{
    private static int aantalToetsen; //static field

    public int AantalToetsen
    {
        get { return aantalToetsen; }
        private set { aantalToetsen = value; }
    }

    public void MaakToets()
    {
        AantalToetsen++;
    }
}
```

Vervolgens leggen een paar leerlingen enkele toetsen af in de methode Main():

```
Leerling jan = new Leerling();
Leerling marie = new Leerling();

jan.MaakToets();
jan.MaakToets();
marie.MaakToets();
Console.WriteLine(jan.TotaalAantalToetsen);
Console.WriteLine(marie.TotaalAantalToetsen);
```

De uitvoer is voor beide objecten hetzelfde omdat ze dezelfde klasse-variabele delen:

```
3
3
```

Als we het veld aantalToetsen **niet static** hadden gemaakt, dan zou dit de uitvoer zijn geweest:

```
2
1
```

Dat komt omdat objecten bij non-static fields (of instantievariabelen) alleen hun eigen variabelen bijhouden.

12.2.2 Klasse-methoden

Klasse-methoden zijn methoden die je **rechtstreeks kan aanroepen op de klasse**. Je behandelt de klasse zelf dus eigenlijk als een object door de methode aan te roepen op basis van de klasse-naam. Dit is erg handig om een hulp- of nutsklassen te schrijven.

Een voorbeeld van een klasse-methode is de methode Pow() van de klasse Math waarbij je een grondtal en exponent ingeeft om machten te berekenen:

```
Math.Pow(2,10); //210= 1024
```

Merk op dat je klasse-methoden niet kan aanroepen vanuit een instantie van de klasse. Sterker nog, met static classes zoals `Math` kan je zelfs geen instanties aanmaken.

Het onderstaande voorbeeld zal een foutmelding genereren:

```
Math_rekenen = new Math();
rekenen.Pow(2,10);
```

Hier volgt nog voorbeeld van een zelfgedefinieerde klasse-methode:

```
class Rekenmachine
{
    static public int Som(int a, int b)
    {
        return a + b;
    }
}
```

De klasse-methode `Som()` wordt vervolgens aangeroepen in de methode `Main()`:

```
int som = Rekenmachine.Som(1, 2);
Console.WriteLine(som); //= 3
```

12.2.3 Hulp- of nutsklassen

Een **hulpkLASSE** (of helper class), ook wel nutskLASSE (of utility class) genoemd, **bevat meestal alleen maar statische variabelen en methoden**. Ze worden ontworpen om bepaalde functionaliteiten te groeperen die niet specifiek zijn voor een bepaalde klasse of object, maar vaak wel nodig zijn in verschillende delen van de code.

Zo kan je bijvoorbeeld een hulpkLASSE hebben voor het uitvoeren van kansberekeningen, het manipuleren van strings, enzovoort. Het beste voorbeeld van een hulpkLASSE is (opnieuw) de klasse `Math`, maar er bestaan uiteraard veel meer voorgedefinieerde hulpklassen.

Voorbeeld van een zelfgedefinieerde hulpkLASSE:

```
public static class Rekenmachine
{
    private static int aantalBerekeningen;
    public static int AantalBerekeningen
    {
        get { return aantalBerekeningen; }
        private set { aantalBerekeningen = value; }
    }

    public static int Som(int a, int b)
    {
        AantalBerekeningen++;
        return a + b;
    }
}
```

i Merk op dat de klasse zelf statisch is. Je kan bijgevolg dus geen objecten van hulpklassen aanmaken. Probeer je dit wel, dan krijg je een foutmelding van de compiler.

12.3 Static vs. non-static

Zoals je ondertussen weet worden klasse-variabelen en -methoden gedeeld met alle instanties van een klasse. Bijgevolg kan een static method kan geen toegang krijgen tot niet-statistische variabelen en methoden van individuele objecten in de klasse. Dit komt doordat het niet duidelijk is welk individueel object de methode zou moeten benaderen. Kortom, **een statische methode kan alleen andere statische velden en methoden aanspreken**.

Omgekeerd kan wel. Gewone (niet-statistische) methoden kunnen zowel statische als niet-statistische variabelen en methoden aanspreken. De onderstaande afbeelding geeft visueel weer wat kan aangeroepen worden en wat niet:



Bron: Zie Scherp Scherper (Dams, 2022)

Voorbeeld van een static methode die toch gebruik wilt maken van een non-static field :

```
class Leerling
{
    private int aantalGemaakteToetsen = 0;

    public static void MaakToets()
    {
        aantalGemaakteToetsen++;
    }
}
```

Deze code zal de volgende foutmelding geven:

An object reference is required for the non-static field, method, or property
'Leerling.aantalGemaakteToetsen'

12.4 Static Random number generators



Het is **good practice** om een willekeurige getallengenerator in C# `static` te maken zodat alle objecten dezelfde willekeurige getallen gebruiken.

Hierdoor verklein je de kans dat objecten dezelfde willekeurige getallen zullen genereren wanneer ze op quasi hetzelfde moment worden geïnstantieerd of aangeroepen in een methode.

Plaats de generator dus apart (gewoonlijk bovenaan) in de klasse met het sleutelwoord `static` ervoor.

Voorbeeld uit het boek Zie Scherp Scherper (Dams, 2022):

```
class Dobbelensteen
{
    static Random rng = new Random();
    public int Werp()
    {
        return rng.Next(1, 7);
    }
}
```




Opgepast! Enumeraties zijn een onderschat onderwerp bij veel (beginnende) programmeurs omdat je eigenlijk perfect programma's kan schrijven zonder enums. In het begin lijken ze wat raar, onwennig en misschien wel overbodig. Maar van zodra je echter het nut ervan inziet én door hebt hoe het werkt, zal je niet meer zonder kunnen.

13.1 Wat zijn enumeraties?

Een enumeratie (of enumeration) is een mooi woord voor het **opsommingstype**. Het is een "speciale klasse" dat een groep van vooraf gedefinieerde, onveranderlijke **constanten** bevat.

Enerzijds kan je het dus bekijken als een lijst met opsommingen. Anderzijds kan je een **enum** beschouwen als een nieuw **datatype** met een **beperkt aantal constante waarden**. We kunnen namelijk nieuwe variabelen van dit datatype creëren die enkel waarden bevatten die in de enum gedefinieerd zijn.



C# zit vol met ingebouwde enum-types zoals `ConsoleColor`. Deze enum bevat een reeks vaste kleuren waaruit je kan kiezen om de tekst- en/of achtergrondkleur van de console te veranderen.

Bijvoorbeeld: `Console.ForegroundColor = ConsoleColor.Red;`

13.2 De bestaansreden van enumeraties

We illustreren het nut van het enums met een voorbeeld:

Stel dat je een programma moet schrijven voor een online webshop zoals Bol.com waarbij je de status van de bestelling wilt meedelen aan je klanten. De bestelling heeft op elk moment één van de volgende mogelijkheden:

- In afwachting,
- In verwerking,
- Verzonden,
- Afgeleverd,
- Geannuleerd.

13.2.1 Slechte manieren om constanten op te sommen

Zonder enums kunnen we deze constante waarden op **twee foutgevoelige manieren** schrijven:

1. **Met een `int`** (een numerieke waarde) voor elke status:

```
public class Bestelstatus
{
    public const int InAfwachting = 0;
    public const int InVerwerking = 1;
    public const int Verzonden = 2;
    public const int Afgeleverd = 3;
    public const int Geannuleerd = 4;
}
```

```

class Program
{
    static void Main(string[] args)
    {
        int status = Bestelstatus.Verzonden;

        switch (status)
        {
            case 0: Console.WriteLine("Bestelling in afwachting"); break;
            case 1: Console.WriteLine("Bestelling wordt verwerkt"); break;
            case 2: Console.WriteLine("Bestelling verzonden"); break;
            case 3: Console.WriteLine("Bestelling afgeleverd"); break;
            case 4: Console.WriteLine("Bestelling geannuleerd"); break;
        }
    }
}

```

2. Met een **string** die de naam van de status bevat:

```

public class Bestelstatus
{
    public const string InAfwachting = "In afwachting";
    public const string InVerwerking = "In verwerking";
    public const string Verzonden = "Verzonden";
    public const string Afgeleverd = "Afgeleverd";
    public const string Geannuleerd = "Geannuleerd";
}

class Program
{
    static void Main(string[] args)
    {
        string status = Bestelstatus.Verzonden;

        switch (status)
        {
            case "In afwachting":
                Console.WriteLine("Bestelling in afwachting"); break;
            case "In verwerking":
                Console.WriteLine("Bestelling wordt verwerkt"); break;
            case "Verzonden":
                Console.WriteLine("Bestelling verzonden"); break;
            case "Afgeleverd":
                Console.WriteLine("Bestelling afgeleverd"); break;
            case "Geannuleerd":
                Console.WriteLine("Bestelling geannuleerd"); break;
        }
    }
}

```

De bovenstaande manieren hebben echter de volgende **nadelen**:

- **Onveilig:** eender welke waarde kan toegekend worden aan de variabele.
`int status = 100 // deze status bestaat niet`
- **Onleesbaar:** welke waarde hoort ook alweer bij welke naam? Begint de telling vanaf 0 of 1? Wat hoort bij 2?
`if(status == 2) // wat is 2?`
- **Onduidelijkheid:** de afdruk van een constante toont alleen de primitieve waarde.
`Console.WriteLine(Bestelstatus.Geannuleerd); // = 4`
- **Foutgevoeliger:** bijvoorbeeld voor schijffouten bij het gebruik van strings.
`case "In afwachting" of was het "InAfwachting" of was het "in afwachting" of ...?`
- Er is geen automatische toewijzing van waarden aan elke constante.

13.2.2 Enumeraties als oplossing voor het opsommen van constanten

Laten we nu hetzelfde voorbeeld uitwerken **met een enum**:

```
public enum Bestelstatus
{
    InAfwachting,
    InVerwerking,
    Verzonden,
    Afgeleverd,
    Geannuleerd
}

class Program
{
    static void Main(string[] args)
    {
        Bestelstatus status = Bestelstatus.Verzonden;

        switch (status)
        {
            case Bestelstatus.InAfwachting:
                Console.WriteLine("Bestelling in afwachting"); break;
            case Bestelstatus.InVerwerking:
                Console.WriteLine("Bestelling wordt verwerkt"); break;
            case Bestelstatus.Verzonden:
                Console.WriteLine("Bestelling verzonden"); break;
            case Bestelstatus.Afgeleverd:
                Console.WriteLine("Bestelling afgeleverd"); break;
            case Bestelstatus.Geannuleerd:
                Console.WriteLine("Bestelling geannuleerd"); break;
        }
    }
}
```

Zoals je ziet lossen enums de nadelen van de voorbeelden met `int` en `string` op, want nu is het programma:

- **Veilig:** de code kan alleen waarden aannemen die gedefinieerd zijn in de enum.
- **Duidelijk:** de constanten krijgen betekenisvolle namen i.p.v. zogenaamde “magic numbers” waarvan jijzelf en andere programmeurs achteraf moeilijk kunnen nagaan waar die getallen voor staan.
- **Onderhoudbaar:** de logica is gebaseerd op vaste sets van waarden. Dit voorkomt dat de waarden doorheen de uitvoering van het programma kunnen wijzigen.
- **Minder foutgevoelig:** je moet een constante kiezen uit een lijst i.p.v. de constante zelf te schrijven.
- **Gebruiksvriendelijker.**
- **Sneller.**
- ...



Er zijn nog meer voordelen die je gaandeweg zal ontdekken, maar ik geloof dat deze lijst met voordelen wel volstaat om je te overtuigen van het nut van enums.

13.3 Enumeraties aanmaken

13.3.1 Het datatype `enum` definiëren

Om een enumeratie aan te maken gebruik je het **sleutelwoord** `enum` gevolgd door een **zelfgekozen naam** van de enumeratie. Vervolgens som je de **constante waarden** op tussen accolades `{ }` . De constanten worden van elkaar gescheiden met een komma.



Net als bij klassen en methoden, schrijf je de naam van enum in **PascalCase**. Ook de constante waarden in een enum moeten steeds met een hoofdletter starten.

Syntax:

```
enum EnumNaam
{
    Waarde1, Waarde2, ... //constanten
}
```

Voorbeeld:

```
enum Kleuren
{
    Rood,
    Groen,
    Blauw
}
```

Merk op dat je zelf mag kiezen hoe je de constanten in een enumeratie opsomt. Gewoonlijk wordt een enum met weinig waarden op één regel geschreven, anders worden de waarden onder elkaar geschreven. De keuze is afhankelijk van de leesbaarheid van de enum in kwestie.

- i** Een enumeratie kan gedeclareerd worden op klasse-niveau, of als onderdeel van een klasse. Het is echter een goede gewoonte onder programmeurs om alle enumeraties te verzamelen in een aparte klasse. Dit houdt de code gemakkelijk onderhoudbaar en overzichtelijk.

13.3.2 De interne waarde van enumeraties

Intern worden de constanten van een enum **genummerd** met een numerieke waarde van het **type int**, beginnende bij 0.

```
enum Moeilijkheidsgraad
{
    Laag,          //0
    Gemiddeld,     //1
    Hoog           //2
}
```

Je kan de **interne waarde manueel veranderen** door bij een constante expliciet aan te geven wat de nieuwe waarde moet zijn aan de hand van **een toekenning**.

```
enum Moeilijkheidsgraad { Laag = 1, Gemiddeld, Hoog = 100 }
```

Merk op dat de interne waarde van de constanten verder telt op dat van de voorgaande waarde. Aangezien de interne waarde van Laag nu gelijk is aan 1, zal de interne waarde van de constante Gemiddeld gelijk zijn aan 2. De waarde van Hoog is gelijk aan 100 omdat we dit zo hebben toegekend.

13.3.3 Variabelen van het type enum declareren en initialiseren

Neem de volgende enum als voorbeeld:

```
enum WeerType { Zonnig, Bewolkt, Regen, Sneeuw, Mistig, Stormachtig }
```

Net als bij andere datatypes kan je nu een variabele van het type WeerType declareren en initialiseren.

```
WeerType weerMorgen = WeerType.Bewolkt;
```

Uiteraard kan je een variabele ook apart declareren en initialiseren.

```
WeerType weerVandaag;
weerVandaag = WeerType.Zonnig;
```

Syntax:

```
EnumNaam naam = EnumNaam.Item;
```

Hou er wel rekening mee dat je, in tegenstelling tot de andere datatypen, alleen waarden aan de variabelen kan toewijzen die in de enum zijn opgenomen.

13.4 enum-variabelen gebruiken

13.4.1 Enumeraties en beslissingen

Enumeraties **werken perfect samen met beslissingsstructuren** zoals `if-` en (vooral) `switch`-statements. Dit komt omdat een enum over een beperkte en vaste set van waarden beschikt die gebruikt kunnen worden in de voorwaarden van beslissingsstructuren om bepaalde acties uit te voeren. Dit maakt dat de code veel overzichtelijker en leesbaarder wordt weergegeven. Bovendien elimineert dit de kans op het ingeven van onjuiste waarden.

Voorbeeld van een enum in een `if`-statement:

```
if(weerVandaag == WeerType.Sneeuw)
```

Voorbeeld van een enum in een `switch`-statement:

```
switch(weerVandaag)
{
    case WeerType.Zonnig: Console.WriteLine("De zon schijnt!"); break;
    case WeerType.Bewolkt: Console.WriteLine("Het is bewolkt!"); break;
    //...
}
```

13.4.2 enum-variabelen converteren (casten)

Neem de volgende enum als voorbeeld:

```
enum Werkdagen{ Maandag = 1, Dinsdag, Woensdag, Donderdag, Vrijdag }
```

Zoals je weet beschikt een enum-variabele over een interne numerieke waarde van het type `int`. Dit biedt de mogelijkheid om een integer-waarde te converteren naar een enum-waarde met behulp van een cast.

```
int keuze = 3;
Werkdagen dagKeuze = (Werkdagen) keuze; //3 = Woensdag
```

De variabele `dagKeuze` is nu gelijk aan `Woensdag`.

Omgekeerd kunnen we uiteraard ook een enum-waarde converteren naar een integer-waarde.

```
Werkdagen vandaag = Werkdagen.Vrijdag;
int dagVanDeWeek = (int) vandaag; //Vrijdag = 5
```

De variabele `dagVanDeWeek` is nu gelijk aan 5.



Merk op dat het onmogelijk is om decimale waarden zoals een `float`, `double` of `decimal` toe te wijzen aan enums. Je kan de interne waarde namelijk alleen veranderen met gehele getallen.

13.4.3 Enumeraties en keuzemenu's

Enumeraties worden vaak gebruikt om keuzes in een keuzemenu of opties in een programma te definiëren.

Voorbeeld van een eenvoudig programma met keuzemogelijkheden:

```
enum Menu { Inschrijven = 1, Overzicht, Verwijderen, Exit }

static void Main(string[] args)
{
    Console.WriteLine("1. Een leerling inschrijven");
    Console.WriteLine("2. Leerlingenoverzicht tonen");
    Console.WriteLine("3. Een leerling verwijderen");
    Console.WriteLine("4. Programma beëindigen");
    int userkeuze = int.Parse(Console.ReadLine());

    Menu keuze = (Menu) userkeuze;

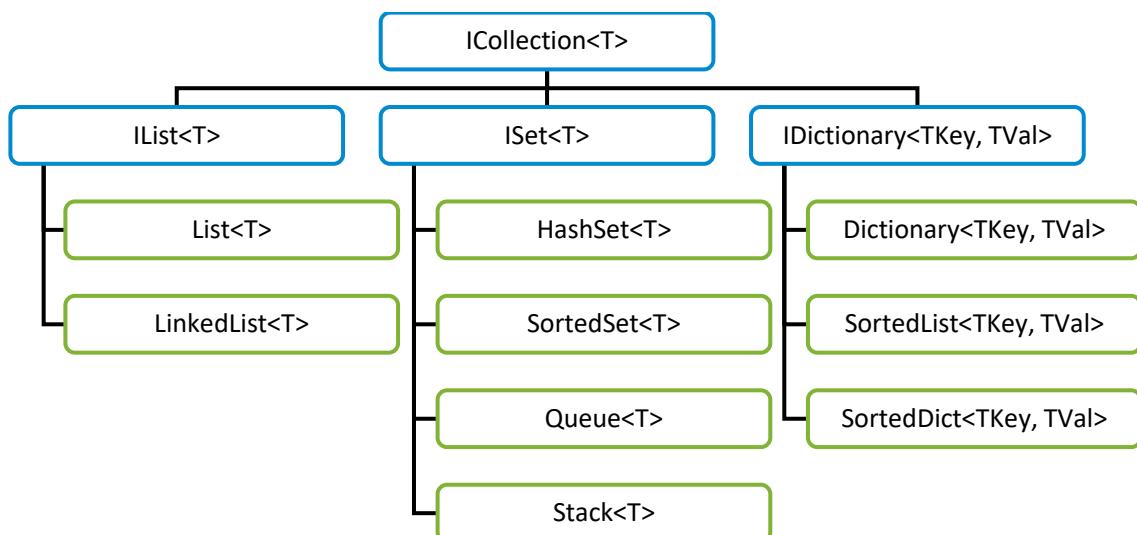
    switch (keuze)
    {
        case Menu.Inschrijven:
            //...
    }
}
```


14.1 Dynamische verzamelingen

Tot nu toe heb je gebruik moeten maken van arrays om meerdere waarden of objecten op te slaan in een verzameling. Arrays hebben echter als nadeel dat ze een vaste grootte hebben die nadien niet meer kan worden aangepast. Als alternatief voor dit probleem bestaan er collectieklassen (of collection classes) waarmee het wél mogelijk is om **de grootte van een verzameling dynamisch te vergroten of verkleinen**.

Een collectie is een **verzameling van objecten**, ook wel elementen genoemd. Over het algemeen zijn collectieklassen nuttig voor het organiseren en bewerken van gegevens. Er zijn verschillende soorten collecties, elk met hun specifieke toepassing. We concentreren ons in deze cursus alleen op de **generieke varianten**, waarbij het **datatype tussen < >** geplaatst wordt. Generieke collectieklassen bieden namelijk een gestructureerde, efficiënte en vooral type-veilige benadering voor het beheren van verzamelingen van objecten. Generieke collecties maken deel uit van de namespace `System.Collections.Generic`.

In het overzicht wordt de `Collection`-interface weergeven met z'n belangrijkste subinterfaces in het blauw, met daaronder enkele van hun implementerende klassen in het groen:



Er bestaan echter nog veel meer collectieklassen dan hierboven weergegeven. Voor een volledig overzicht bekijk je best de officiële documentatie. Al deze klassen overlopen zou teveel tijd in beslag nemen. Daarom beperken we ons in de cursus tot enkele interessante collecties:

- **Arrays**: worden gebruikt wanneer je een vast aantal elementen hebt en ze wilt benaderen via hun index (zie).
- **Lijsten**: worden gebruikt als flexibele arrays; ze zijn interessant als je een onbekend aantal elementen moet opslaan in een verzameling en er verschillende bewerkingen mee moet uitvoeren, zoals toevoegen of verwijderen.
- **Queues en Stacks**: worden gebruikt wanneer je speciale invoeg- en ophaalvolgordes nodig hebt (FIFO voor queues en LIFO voor stacks).
- **Woordenboeken**: worden gebruikt wanneer je sleutels aan waarden moet koppelen en deze waarden snel wilt ophalen aan de hand van hun sleutels.

Hopelijk heb je al wel door dat het belangrijk is om deze verschillende typen verzamelingen goed te begrijpen en te weten wanneer je welke moet gebruiken, afhankelijk van wat je juist wilt doen.

14.2 De collectieklasse List

Een `List<>`-collectie is een geavanceerde versie van arrays. Het doet **hetzelfde als arrays, maar het biedt meer flexibiliteit** omdat je niet van tevoren hoeft aan te geven hoe groot de verzameling zal zijn. Een lijst past zich dus **dynamisch** aan. Je kunt ook specifiek aangeven wat voor soort gegevens er in de lijst zitten.

Kortom, een `List<>` is een krachtige en populaire manier om gegevens te beheren in programmeerprojecten.

14.2.1 Een List aanmaken

Aangezien de klasse `List<T>` een generieke verzameling is, moet je tussen `< >` opgeven welk datatype in de lijst kan worden opgeslagen.

Syntax:

```
List<datatype> naamVanDeLijst = new List<datatype>();
```

Voorbeelden:

```
List<double> prijzen = new List<int>();
List<bool> kopOfMunt = new List<bool>();
List<string[]> lijstVanStringArrays = new List<string[]>();
List<Leerling> leerlingenlijst = new List<Leerling>();
```

Voorbeeld van een lijst met onmiddellijke initialisatie (mogelijk bij elk collectietype):

```
List<string> helloWorld = new List<string>()
{
    { "Hello" },
    { "World" }
};
```

14.2.2 Elementen toevoegen

Met de methode `Add()` kan je elementen toevoegen aan een lijst. Als parameter geef je het element dat je aan de lijst wilt toevoegen. Merk op dat je alleen elementen kan toevoegen van het datatype dat door de lijst aanvaard wordt.

Voorbeeld met het waardetype int:

```
List<int> leeftijden = new List<int>();
leeftijden.Add(16);
```

Voorbeeld met een referentietype van de zelfgedefinieerde klasse Leerling:

```
List<Leerling> leerlingen5AD = new List<Leerling>();
leerlingen5AD.Add(new Leerling());
```

14.2.3 Elementen indexeren

Net als bij arrays, kan je kan elementen van een List<> individueel aanroepen door de **index van een element tussen vierkante haken []** te plaatsen.

Voorbeelden:

```
List<string> namenlijst = new List<string>();  
namenlijst[0] = "Sander";  
Console.WriteLine(namenlijst[0]);
```

Ook de werking van arrays met for en foreach lussen blijft hetzelfde bij List, maar er is wel een verschil. Collectieklassen werken namelijk niet met de eigenschap Length, maar met Count.

Voorbeeld:

```
for(int i = 0 ; i < namenlijst.Count; i++)  
{  
    Console.WriteLine(namenlijst[i])  
}
```

14.2.4 Methoden van de collectieklasse List

In deze tabel staan de meest gebruikte methoden van de klasse List<>:

Returntype	Methode	Beschrijving
void	Clear()	Maakt de volledige lijst leeg. → Count = 0.
void	Insert()	Voegt een nieuw object in op een opgegeven positie in de List.
int	IndexOf()	Zoekt naar het opgegeven object en geeft de index van het element dat als eerste keer gevonden wordt. → Geeft -1 als de waarde niet voorkomt in de List.
void	RemoveAt()	Verwijderd een element op de index die je als parameter meegeeft.

14.3 De collectieklasse Queue

Een Queue (of wachtrij) werkt met het principe van "**First In, First Out**" (FIFO). Het is een collectie waarbij de elementen worden toegevoegd aan het einde van de verzameling en waarbij het eerste/oudste element als eerste behandeld wordt.

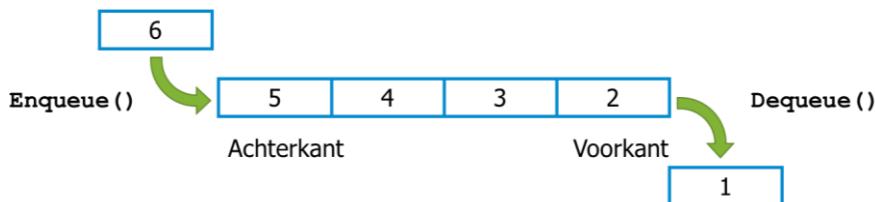
Het is vergelijkbaar met mensen die in de rij staan aan de kassa in een supermarkt. De eerste persoon die in de rij komt, is ook de eerste die bediend wordt en de rij verlaat.



14.3.1 Methoden van de collectieklasse Queue

In deze tabel staan de meest gebruikte methoden van de klasse Queue<>:

Returntype	Methode	Beschrijving
void	Enqueue()	Voegt achteraan de collectie een item toe.
object	Dequeue()	Verwijderd het eerste element in de collectie.
object	Peek()	Geeft het eerste element in de collectie terug, zonder het te verwijderen.



Voorbeeld:

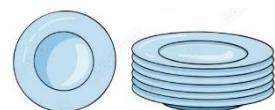
```
Queue<string> wachtrij = new Queue<string>();  
wachtrij.Enqueue("eerste");  
wachtrij.Enqueue("tweede");  
wachtrij.Enqueue("laatste");  
  
Console.WriteLine(wachtrij.Dequeue()); //eerste  
Console.WriteLine(wachtrij.Peek()); //tweede
```

Het eerste element "eerste" in de collectie wordt getoond en meteen verwijderd. Vervolgens wordt het element "tweede" getoond dat nu helemaal vooraan in de rij staat.

14.4 De collectieklasse Stack

Een **Stack** (of stapel) werkt met het principe van "**Last In, First Out**" (**LIFO**). Het is een collectie waarbij het laatst toegevoegde element als eerste wordt behandeld.

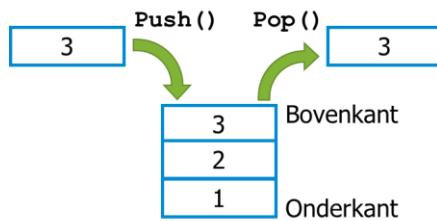
Het is vergelijkbaar met een stapel borden in je servieskast. Je legt een bord bovenop de stapel en wanneer je een bord nodig hebt, pak je ook het bovenste bord.



14.4.1 Methoden van de collectieklasse Stack

In deze tabel staan de meest gebruikte methoden van de klasse Stack<> :

Returntype	Methode	Beschrijving
void	Push()	Voegt bovenaan de stapel een item toe.
object	Pop()	Geeft het eerste element in de stapel terug en verwijdert het uit de stapel.
object	Peek()	Geeft het eerste element in de collectie terug, zonder het te verwijderen.



Voorbeeld:

```
Stack<string> stapel = new Stack<string>();
stapel.Push("eerste");
stapel.Push("tweede");
stapel.Push("laatste");

Console.WriteLine(stapel.Pop());      //laatste
Console.WriteLine(stapel.Peek());    //tweede
```

Het eerste element “laatste” in de collectie wordt getoond en meteen verwijderd. Vervolgens wordt het element “tweede” getoond dat nu helemaal bovenaan de stapel ligt.

14.5 De collectieklasse Dictionary

Een **Dictionary** (of woordenboek) is een collectie dat **sleutel-waarde paren** opslaat, waarbij **elke sleutel uniek** is. Dit maakt snelle data-opzoeken mogelijk omdat elke sleutel direct verwijst naar een waarde.

Dit is vergelijkbaar met een echt woordenboek, waar je een woord (de sleutel) opzoekt om de definitie (de waarde) te vinden. Een ander voorbeeld is een artikellijst waarbij elke artikelcode een unieke sleutel is die gekoppeld is aan een specifiek product.

Artikellijst	
Artikelcode	Artikelnaam
1000104	AKKO kantoorset
1000083	Armleuning, AVANT
1000090	Auto huren
1000105	AVANT volledige kantoorset
1000103	Boekenkast, AKKO EXIT
1000106	Boekenkast, KORO 60
1000073	Boekenkast, PEGASUS 40

14.5.1 Een Dictionary aanmaken

Bij de declaratie plaatsen we tussen de < > eerst het datatype van de sleutel als **key**, en daarnaast het datatype van de waarde als **value**. De opgegeven datatypes hoeven niet hetzelfde te zijn. Bovendien kan je eender welk type ingeven (aangegeven als ‘T’ in de syntax).

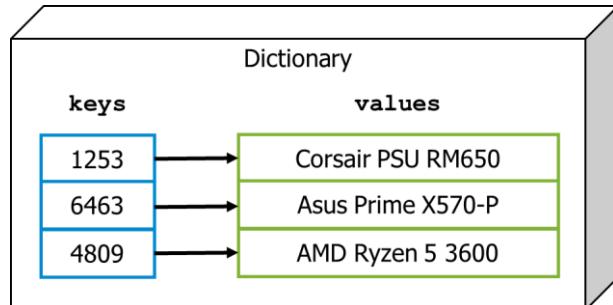
Syntax:

```
Dictionary<Tkey, TValue> woordenboek = new Dictionary<Tkey, TValue>();
```

Voorbeeld:

```
Dictionary<int, string> artikellijst = new Dictionary<int, string>();  
artikellijst.Add(1253, "Corsair PSU RM650");  
artikellijst.Add(6463, "Asus Prime X570-P");  
artikellijst.Add(4809, "AMD Ryzen 5 3600");
```

In het bovenstaande voorbeeld maken we een Dictionary van artikelen aan. Elk artikel heeft een unieke artikelcode (een sleutel van het type int) en een naam (een waarde van het type string).



- i** Een sleutel moet altijd uniek zijn, de waarden daarentegen mogen wel meermaals voorkomen in de collectie. Dit noemen we dan duplicaten. Er bestaan ook collectieklassen waarin duplicaten niet worden opgenomen zodat elke waarde uniek is. Gewoonlijk hebben zulke klassen het woord "hash" in de naam zoals HashSet<>.

14.5.2 Elementen benaderen

We kunnen een specifiek element opvragen door de sleutel te gebruiken op dezelfde manier als de index van een array.

```
Console.WriteLine(artikellijst[1253]); //Corsair PSU RM650
```



Een sleutel heeft géén relatie met de positie van het element in de collectie.

Het doet namelijk dienst als een unieke identifier van een element. Je kan de sleutel bijgevolg niet gebruiken als index om een volledige collectie te doorlopen! Dit komt omdat de indexwaarden elkaar niet altijd met een gelijke, stapsgewijze verhoging opvolgen zoals +1.

Als we toch alle elementen van een Dictionary willen opvragen, doen we dit met een foreach-lus. Dit biedt trouwens het voordeel dat je de eigenschappen Key en Value van elk item individueel kan uitlezen.

```
foreach (var item in artikellijst)  
{  
    Console.WriteLine(item.Key + "\t:" + item.Value);  
}
```

14.5.3 Waarden tonen met beslissingsstructuren

Aangezien elke waarde een sleutel heeft als unieke identifier, kunnen we bepaalde waarden zoeken aan de hand van de sleutel met behulp van een beslissingsstructuur. Het is waarschijnlijk al een gewoonte geworden dat je dit doet met behulp van de operator `==`, maar dat zal helaas niet werken in een Dictionary.

In plaats van de operator `==`, gebruiken we de methode `ContainsKey()` om de waarde van een item te bekomen aan de hand van de sleutel.

```
if (woordenboek.ContainsKey("sleutelwaarde"))
{
    Console.WriteLine(woordenboek["sleutelwaarde"]);
}
```

Voorbeeld:

```
Dictionary<string, double> prijslijst = new Dictionary<string, double>()
{
    {"appel", 0.55},
    {"peer", 0.45},
    {"banaan", 0.75}
};
string product = "appel";

if (prijslijst.ContainsKey(product))
{
    Console.WriteLine($"Prijs van een {product}: €{prijslijst[product]}");
}
```

In dit voorbeeld zoeken we naar de prijs van een appel. Als een appel voorkomt in de prijslijst, wordt de prijs ervan getoond aan de hand van de sleutel die als index werd meegegeven. De uitvoer van dit programma geeft de volgende afdruk in de console:

```
Prijs van een appel: €0,55.
```

14.5.4 Sleutels tonen met beslissingsstructuren

Om een sleutel te vinden, moet je de **volledige collectie doorlopen** en elk item vergelijken met de opgegeven waarde. Dit doen we met behulp van de eigenschap `Value`. Vervolgens gebruiken we de eigenschap `Key` om de sleutel van een gevonden item te tonen.

```
foreach (var item in woordenboek)
{
    if (item.Value == "waarde")
    {
        Console.WriteLine(item.Key);
    }
}
```

het is dus niet mogelijk om rechtstreeks een sleutel te verkrijgen aan de hand van de daaraan gekoppelde waarde. Dit is logisch, want sommige waarden kunnen meerdere sleutels hebben in een Dictionary als duplicaten. Bijgevolg kunnen er mogelijks meerdere sleutels gevonden worden die met de opgegeven waarde overeenkomen.

Voorbeeld:

```
Dictionary<string, double> prijslijst = new Dictionary<string, double>()
{
    {"appel", 0.55},
    {"peer", 0.45},
    {"banaan", 0.75}
};
double prijs = 0.45;

foreach (var item in prijslijst)
{
    if (item.Value == prijs)
    {
        Console.WriteLine($"Prijs van een {item.Key}: €{item.Value}");
    }
}
```

In dit voorbeeld zoeken we naar alle artikelen met een prijs van € 0,45. Elk gevonden artikel wordt met samen met de prijs afgedrukt op een nieuwe regel. De uitvoer van dit programma geeft de volgende afdruk in de console:

```
Prijs van een peer: €0,45.
```

15.1 Nog meer OOP-concepten

Ondertussen heb je al wat ervaring opgedaan met objecten, klassen en inkapseling. Hoog tijd dus om eens dieper te duiken in enkele van de meest krachtige concepten van object georiënteerd programmeren (OOP), namelijk **overerving**, **polymorfisme** en **abstractie**.

Zoals je weet streeft OOP naar efficiëntie en onderhoudsvriendelijkheid door zoveel mogelijk **dubbele code te vermijden**, waardoor het **aantal potentiële fouten (bugs) verminderd**. Door overerving, polymorfisme en abstractie correct te implementeren, zal je efficiënte, flexibele en krachtige software-architecturen kunnen ontwikkelen die gemakkelijk aangepast en uitgebreid kunnen worden (deze uitspraak zal je de komende hoofdstukken vaak horen).

15.2 Overerving

Overerving (of inheritance) stelt ons in staat om nieuwe klassen te maken die **eigenschappen en methoden** kunnen **overnemen van een andere klasse**. Dit maakt het mogelijk om een hiërarchie van klassen te creëren en het hergebruik van code zoveel mogelijk te bevorderen. Bovendien kunnen eigenschappen en methoden in overervende klassen verder worden **aangepast** en **uitgebreid**.

15.2.1 Introductie tot overerving

We illustreren de kracht van overerving aan de hand van het volgende voorbeeld:

```
public class Brommer
{
    public int AantalWielen { get; set; }
    public double Snelheid { get; set; }
    public void Rijden()
    {
        Console.WriteLine("Je begint te rijden.");
    }
}

public class Auto
{
    public int AantalWielen { get; set; }
    public double Snelheid { get; set; }
    public void Rijden()
    {
        Console.WriteLine("Je begint te rijden.");
    }
}
```

Zoals je ziet komt dezelfde code voor in de klassen `Brommer` en `Auto`. Dit kan echter tot problemen leiden wanneer blijkt dat er fouten in deze code zitten. In dat geval zijn we helaas genoodzaakt om in meerdere klassen dezelfde aanpassingen door te voeren. Het spreekt dus voor zich dat dit

tijdrovend en verre van efficiënt is. Kortom, de bovenstaande code is zeer slecht en vloekt tegen de principes van OOP.

Gelukkig kunnen we dit probleem oplossen met overerving door de gemeenschappelijke code van beide klassen op te nemen in een “overkoepelende” klasse Voertuig. Vervolgens laten we de klassen Auto en brommer overerven van de klasse Voertuig.

```
public class Voertuig
{
    public int AantalWielen { get; set; }
    public double Snelheid { get; set; }
    public void Rijden()
    {
        Console.WriteLine("Je begint te rijden.");
    }
}

public class Brommer : Voertuig
{ }

public class Auto : Voertuig
{ }
```

Merk op dat we geen nieuwe code hoeven te schrijven in de klassen Brommer en Auto. Dit komt doordat ze alle eigenschappen en methoden van de superklasse Voertuig hebben overgeërfd.

- (i) Deze introductie doet uitschijnen dat overerving enkel z'n nut heeft om dubbele code te vermijden, wat niet zo is. Dubbele code vermijden dankzij overerving is eerder een gevolg ervan. Overerving is een erg krachtig concept dat in de komende hoofdstukken telkens zal terugkomen.

15.2.2 Super en subklassen

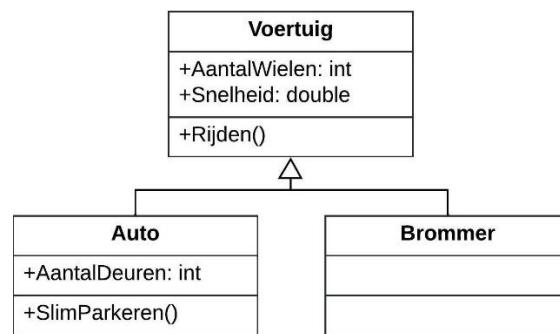
Zoals je ziet kunnen klassen relaties hebben met elkaar. Zo onderscheiden we superklassen en subklassen die samen deel uitmaken van een klasse-hiërarchie.

Een **superklasse** (of parent class) is een klasse dat **alle gemeenschappelijke code** van verschillende overervende klassen in zich opneemt. De hoogste klasse in een klasse-hiërarchie wordt ook wel een **basisklasse** (of base class) genoemd.

Subklassen (of child class) zijn klassen die alle eigenschappen en methoden overerven van een superklasse. Deze eigenschap maakt dat overerving in C# **transitief** is. Vervolgens is het mogelijk om de implementatie van de overgeërfde leden te **vervangen** met behulp van een override.

Daarnaast kan je aan elke subklasse code **toevoegen** dat specifiek is voor deze klasse. Omwille van die reden noemt men bepaalde subklassen soms ook wel specialisatieklassen. Het is namelijk een klasse dat alles kan wat de superklasse kan, met daarbovenop nog eens extra functionaliteiten die uniek zijn voor deze klasse. Alle klassen die afgeleid zijn van een basisklasse noemt men trouwens een **afgeleide klasse** (of derived class).

Onderstaande UML class diagram toont van de klasse-hiërarchie met een uitbreiding in de klasse Auto:

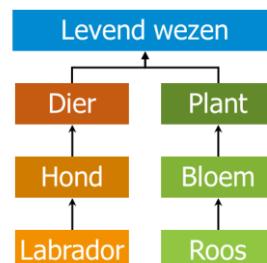


De superklasse **Voertuig** bevat alle gemeenschappelijke code. De subklasse **Brommer** neemt deze allemaal over zonder er iets aan toe te voegen. De subklasse **Auto** daarentegen heeft daarbovenop nog een unieke eigenschap voor het aantal deuren, alsook een unieke methode om de auto zelfstandig te laten parkeren.

15.2.3 Is-een relatie

Wanneer je twee klassen kunt beschrijven als een "x is een y"-relatie, duidt dit op de **mogelijkheid van overerving**. Bijvoorbeeld:

- Een labrador is een hond → een hond is een dier → een dier is een levend wezen.
- Een roos is een bloem → een bloem is een plant → een plant is een levend wezen.



De kans is dus groot dat overerving mogelijk is als je ergens in een programmeeropdracht een vervoeging van het **werkwoord 'zijn'** tegenkomt (bv. was, is, zijn, zal zijn, zijnde, ...).



Overerving is enkel interessant wanneer er een realistische "is een"-relatie bestaat.

Het feit dat twee klassen gelijkaardige (of zelfs identieke) code delen, betekent niet automatisch dat overerving een goede oplossing is.

Vandaar dat overerving slechts een mogelijke oplossing is. Gelukkig zijn er nog andere manieren om dubbele code te voorkomen, zelfs als er geen "is een"-relatie bestaat. Meer hierover in de volgende hoofdstukken.

15.2.4 Subklassen definiëren

In C# wordt overerving in een subklasse aangeduid in de klasse definitie (of class header) met een dubbele punt : en daarachter de naam van de superklasse.

Syntax:

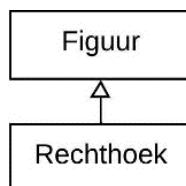
```
class Subklasse : Superklasse
{ }
```

Voorbeeld 1:

```
class Figuur           //Superklasse
{ }

class Vierhoek : Figuur //Subklasse
{ }
```

De klasse Vierhoek is als het ware een uitbreiding van de klasse figuur:



Voorbeeld 2:

```
class Dier
{
    public void Eet()
    {
        Console.WriteLine("Het dier eet");
    }
}

class Hond : Dier
{
    public bool KanBlaffen{ get; set; }
}
```

De subklasse Hond erft de methode Eet() over van de superklasse Dier en voegt aan z'n eigen klasse omschrijving (of class body) de eigenschap KanBlaffen toe.

Objecten van het type Dier kunnen alleen de methode Eet() aanroepen, terwijl objecten van de klasse Hond ook de methode Eet() kunnen aanroepen én beschikken over de eigenschap KanBlaffen. De subklasse Hond is dus een specialisatie van de superklasse Dier, want een hond kan alles wat een dier kan en wat het zelf kan.

15.2.5 Subklassen gebruiken

We gaan verder met de voorbeeld van de klassen Dier en Hond. In de methode Main() kan je de klassen gebruiken als volgt:

```

Dier dier = new Dier();
Hond hond = new Hond();

dier.Eet();
hond.Eet();

dier.KanBlaffen = true; //Gaat niet
hond.KanBlaffen = true;

```

Merk op dat het object dier de property KanBlaffen niet kan gebruiken omdat deze uniek is voor de klasse Hond. Objecten van **superklassen hebben dus geen toegang tot** de properties en methoden van **hun subklassen**.

15.2.6 Klasse-hiërarchie

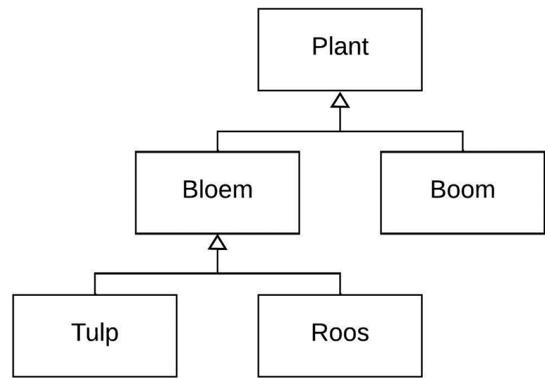
Een klasse-hiërarchie is een organisatie van klassen in een overervingsstructuur. Helemaal bovenaan in de hiërarchie heb je de basisklasse (of base class) waarvan alle andere klassen afgeleid zijn. Dit toont aan dat het dus perfect mogelijk is om meermaals over te erven. In een klasse-hiërarchie zal je in de bovenste klassen meer algemene code terugvinden. Des te lager je gaat, des te specieker de code in de subklassen zal zijn.

Voorbeeld van een klasse-hiërarchie:

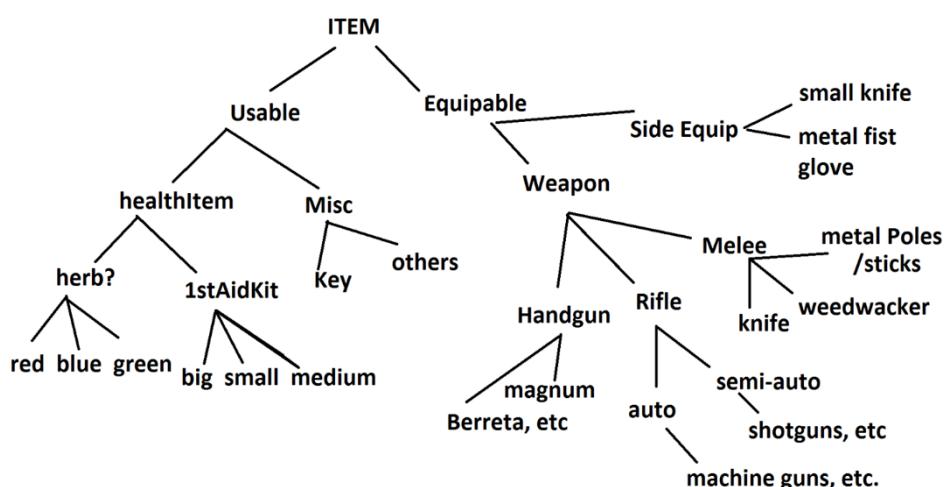
```

class Plant
{
}
class Boom : Plant
{
}
class Bloem : Plant
{
}
class Roos: Bloem
{
}
class Tulp : Bloem
{
}

```



Voorbeeld van een ‘Inheritance tree’ (of overervingsboom) in een videogame met de klasse Item als base class:



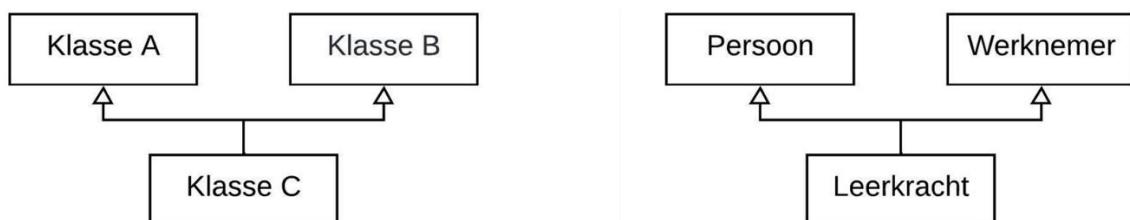


Zoals je ziet kan een klasse-hiërarchie snel uitgebreid en complex worden. Dit brengt het risico met zich mee dat je na een tijd door de bomen het bos niet meer ziet. Overdrijf dus niet met het aantal subklassen dat je aanmaakt. Tracht ten allen tijden je code zo goed georganiseerd en overzichtelijk mogelijk te houden.

En als dat niet lukt met overerving? Geen paniek, er zijn nog andere manieren om je code goed te organiseren. Deze oplossingen zullen in de volgende hoofdstukken aan bod komen.

15.2.6.1 Multiple inheritance

We komen even terug op de uitspraak dat het perfect mogelijk is om meermaals over te erven. Sommige talen, zoals Python en C++, bieden aan een subklasse namelijk de mogelijkheid om **over te erven van meerde superklassen**. Een klasse **Leerkracht** zou dan bijvoorbeeld kunnen overerven van de klassen **Persoon** en **Werknemer**. Dit fenomeen noemt men multiple inheritance.



Multiple inheritance is echter niet mogelijk in C#! Dit komt omdat het gewoonweg amper voorkomt in de echte wereld. Ja, er zijn uitzonderingen zoals het vogelbekdier dat zowel eigenschappen heeft van vogels als van zoogdieren, maar hoe vaak ga je dat nu modelleren in een project? Je zal dit probleem trouwens vaak kunnen oplossen door gebruik te maken van compositie en interfaces.



In C# kan elke subklasse slechts overerven van één superklasse.

15.2.7 Transitiviteit en toegankelijkheid

Transitiviteit bij overerving verwijst naar het vermogen van een subklasse om **alle leden over te erven** van zowel zijn directe superklasse als alle superklassen hoger in de klasse-hiërarchie.

Dit geldt dus **ook voor alle private variabelen en methoden van de superklasse(n)**. Dit wilt echter niet zeggen dat de subklasse opeens deze private leden kan benaderen! De access modifier **private** beperkt namelijk de zichtbaarheid van een lid tot de klasse waarin het is gedeclareerd. Hierdoor kan het lid alleen gebruikt worden binnen diezelfde klasse en is het niet toegankelijk van buitenaf, zelfs niet voor afgeleide klassen.

Toegangs niveau	Eigen klasse	Subklassen	Alle klassen
Private	X		
Protected	X	X	
public	X	X	X

Voorbeeld met een private lid in de superklasse Dier:

```
class Dier
{
    private int leeftijd;
}

class Hond : Dier
{
    public void MaakOuder()
    {
        leeftijd++;
    }
}
```

Deze code zal dus niet werken in de klasse Hond, want de instantievariabele `leeftijd` is gedeclareerd als `private` in de superklasse Dier.

15.2.7.1 De access modifier `protected`

Gelukkig kan je het probleem van private leden in superklassen oplossen door ze te vervangen met de access modifier `protected`. Dit toegangsniveau werkt namelijk als een uitbreiding van `private`, waarbij het lid zichtbaar en toegankelijk is binnen de klasse waarin het is gedeclareerd, én binnen al zijn subklassen.

Voorbeeld met een `protected` lid in de superklasse:

```
class Dier
{
    protected int leeftijd;
}

class Hond : Dier
{
    public void MaakOuder()
    {
        leeftijd++;
    }
}
```

Deze code zal wel werken omdat de instantievariabele `leeftijd` nu zichtbaar en toegankelijk is in alle subklassen van de superklasse Dier.

15.2.7.2 Sealed klassen

Gewoonlijk staan helemaal onderaan in de klasse-hiërarchie klassen waarvan je niet wilt dat daarvan overgeërfd kan worden. Met het sleutelwoord `sealed` kan je voorkomen dat je collega-programmeurs, of misschien wel jijzelf, later toch proberen over te erven van deze klasse.

Voorbeeld:

```
sealed class MagNietOvererven
{
    //Klasse-omschrijving (class body)
}
```

Als je toch probeert over te erven van een sealed klasse ...

```
class Subklasse : MagNietOvererven
{
    //Klasse-omschrijving (class body)
}
```

... zal dit de volgende foutmelding genereren:

```
'Subklasse': cannot derive from sealed type 'MagNietOvererven'
```

15.2.8 Constructors bij overerving

15.2.8.1 Default constructors bij overerving

Bekijk even het volgende voorbeeld van (default) constructors in een klasse-hiërarchie:

```
class Dier
{
    public Dier()
    {
        Console.WriteLine("Het dier is aangemaakt");
    }
}

class Zoogdier : Dier
{
    public Zoogdier()
    {
        Console.WriteLine("Het zoogdier is aangemaakt");
    }
}

class Kat : Zoogdier
{
    public Kat()
    {
        Console.WriteLine("De kat is aangemaakt");
    }
}
```

Elke constructor die wordt aangeroepen bij de creatie van een object, zal een boodschap genereren.

In de methode Main() creëren we vervolgens een nieuw object van het type Kat:

```
Kat fluffy = new Kat();
```

Dit levert de volgende boodschap wanneer we het programma uitvoeren:

```
Het dier is aangemaakt  
Het zoogdier is aangemaakt  
De kat is aangemaakt
```

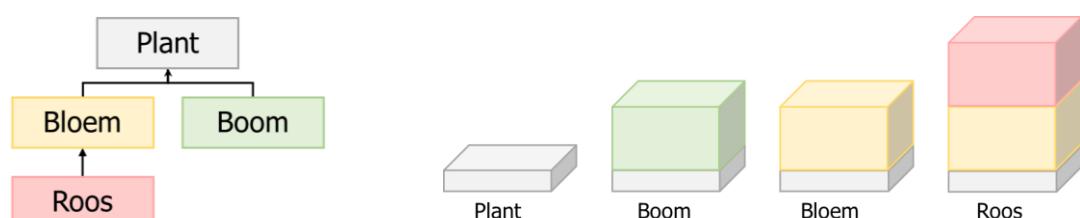
Wanneer we een object van een subklasse aanmaken, stellen we het volgende vast:

1. Eerst wordt de constructor aangeroepen van de basis-klasse.
2. Vervolgens worden de constructors aangeroepen van alle opvolgende superklassen.
3. Tenslotte wordt de constructor van de subklasse zelf aangeroepen.

Er wordt hier dus verondersteld dat er een default constructor in de basis-klasse aanwezig is.

De constructors van superklassen worden niet geërfd door de subklassen, maar alle constructors zijn wel gekoppeld aan de constructor van de basisklasse. De subklasse heeft dus als het ware de "fundering" van zijn superklasse(n) nodig om te kunnen functioneren. Dit garandeert namelijk dat de toestand van het object correct wordt geïnitialiseerd.

Op de volgende afbeelding zie je een visuele interpretatie van klassen in het geheugen. Zoals je ziet bouwt elke subklasse verder op de superklassen waar het van overerft. Dit verklaart meteen waarom de constructors van alle superklassen eerst worden doorlopen voordat de constructor van de eigen klasse wordt aangeroepen. Kortom, alle superklassen worden eerst doorlopen voordat de eigen klasse aan bod komt.



15.2.8.2 Overloaded constructors bij overerving

Elke constructor van een afgeleide klasse moet de constructor van de basisklasse aanroepen. Maar als de basisklasse niet over een default constructor beschikt, is een aanroep naar de juiste overloaded constructor van de basisklasse vereist.

Volgende code zou dus een probleem geven wanneer je een object van het type Roos wilt aanmaken via new Roos():

```
class Bloem  
{  
    public string Kleur { get; set; }  
    public Bloem(string kleur)          //Overloaded constructor  
    {  
        Kleur = kleur;  
    }  
}
```

```

class Roos : Bloem
{
    public Roos() //Default constructor
    {
        Console.WriteLine("De roos is aangemaakt");
    }
}

```

Aangezien de basisklasse Bloem niet over een default constructor beschikt, zal de volgende foutmelding verschijnen wanneer je een nieuw Roos-object wilt aanmaken met behulp van een default constructor:

```
There is no argument given that corresponds to the required parameter 'kleur' of 'Bloem.Bloem(string)'
```

Gelukkig kan je met het **sleutelwoord** `base` een expliciete aanroep doen naar de constructor van de basisklasse, wanneer je met overloaded constructors werkt in een klassehiërarchie. We onderscheiden in dat geval de volgende mogelijke situaties waarbij de afgeleide klasse de overloaded constructor van de basisklasse aanroept:

1. De aanroep gebeurt vanuit een default constructor

Wanneer een afgeleide klasse de aanroep doet met een default constructor, zal je voor elke parameter van de overloaded constructor in de basisklasse een standaardwaarde moeten ingeven.

Voorbeeld:

```

class Bloem
{
    public string Kleur { get; set; }
    public Bloem(string kleur) //Overloaded constructor
    {
        Kleur = kleur;
    }
}

class Roos : Bloem
{
    public Roos() : base("rood") //Default constructor
    {
        Console.WriteLine("De roos is aangemaakt");
    }
}

```

In dit voorbeeld zal de default constructor van Roos de waarde van de actuele parameter `kleur` steeds op “rood” zetten. Er moet hoe dan ook steeds een geldige waarde worden meegegeven om de constructor te kunnen aanroepen.

2. De aanroep gebeurt vanuit een eigen overloaded constructor

Wanneer een afgeleide klasse een eigen overloaded constructor aanroept, kan je hier ook voor elke parameter een standaardwaarde ingeven. Dit is echter niet vereist aangezien je nu via parameters de waarden kunt meegeven tijdens de creatie van een object. Hiervoor moet wel een

verwijzing worden gemaakt naar de parameters van de constructor van de basisklasse. Hiervoor gebruik je opnieuw gebruik van het sleutelwoord `base`.

Voorbeeld:

```
class Bloem
{
    public string Kleur { get; set; }
    public Bloem(string kleur) //Overloaded constructor
    {
        Kleur = kleur;
    }
}

class Roos : Bloem
{
    public Roos(string kleur) : base(kleur) //Default constructor
    {
        Console.WriteLine("De roos is aangemaakt");
    }
}
```

In het bovenstaande voorbeeld wordt de parameter `kleur` in de afgeleide klasse duidelijk doorverwezen naar de gelijknamige parameter `kleur` in de basisklasse. Hierdoor kan je parameters met waarden ingeven tijdens de creatie van een object van het type `Roos`.

3. Een hybride aanpak

Situaties waarbij de parameters verdeeld worden tussen de sub- en superklasse zijn ook mogelijk, zolang de constructor in kwestie maar bestaat in de superklasse.

Voorbeeld:

```
class Persoon
{
    public string Naam { get; set; }
    public Persoon(string naam)
    {
        Naam = naam;
    }
}

class Werknemer : Persoon
{
    public int WerknemersID { get; set; }
    public Werknemer(string naam, int werknemersID) : base(naam)
    {
        WerknemersID = werknemersID;
    }
}
```

Het bovenstaande voorbeeld toont de klassen `Person` en `Employee` waarbij we de constructor van `Employee` zodanig beschrijven dat deze bepaalde parameters zoals `employeeID` "voor zich houdt" en de andere parameters als het ware doorsluist naar de aanroep van de superklasse `Person`.

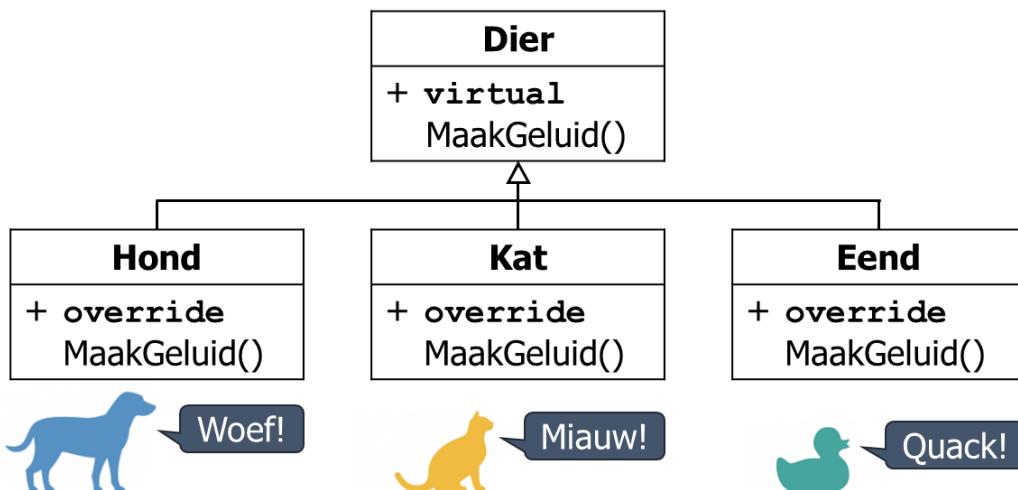
15.3 Polymorfisme

15.3.1 Sleutelwoorden `virtual` en `override`

Het woord 'polymorfisme' is afgeleid van de Latijnse woorden 'poly' en 'morfisme', wat letterlijk '**meerderen vormen**' betekent. Het is een belangrijk en krachtig programmeerconcept waarbij objecten van subklassen de **implementatie van een overgeërfde eigenschap of methode** kunnen **aanpassen** en/of **uitbreiden** met extra functionaliteit, naargelang de noden van elke klasse. Hiervoor gebruiken we de sleutelwoorden `virtual` en `override`.

Gewoonlijk wil je voorkomen dat een subklasse zomaar de implementatie van om het even welke eigenschap en methode van een superklasse kan aanpassen. Dit zou namelijk kunnen leiden tot fouten in het programma. Daarom moet je altijd expliciet in de parent-klasse aangeven welke leden door child-klassen aangepast en uitgebreid **mogen** worden, met het sleutelwoord `virtual`. Het is dus een optie en zeker **geen verplichting**.

Een child-klasse kan trouwens alleen een lid van een parent-klasse aanpassen of uitbreiden wanneer het voorzien is van het sleutelwoord `override`. En uiteraard is dit alleen toegestaan als hetzelfde lid in de superklasse aangeduid staat met het `virtual`.



Voorbeeld van de subklasse `Hond` die de implementatie van de methode `MaakGeluid()` van de superklasse `Dier` overschrijft/aanpast:

```
class Dier           //Parent-klasse
{
    public virtual void MaakGeluid()
    {
        Console.WriteLine("Dit dier maakt een geluid");
    }
}
```

```

class Hond : Dier          //Child-klasse
{
    public override void MaakGeluid()
    {
        Console.WriteLine("De hond zegt: Woef!");
    }
}

```

Nu laten we een object van beide klassen de methode MaakGeluid() aanroepen:

```

Dier dier = new Dier();
Hond hond = new Hond();

dier.MaakGeluid();
hond.MaakGeluid();

```

Dit geeft de volgende uitvoer:

```

Dit dier maakt een geluid!
De hond zegt: Woef!

```

Syntax (van eenzelfde methode):

```

public virtual void MethodeNaam()      //methode van de parent-klasse
{
    //statements
}

public override void MethodeNaam()     //methode van de child-klasse
{
    //Aanpassing en/of uitbreiding van deze methode
}

```

Zoals je ziet moet de signatuur van de eigenschap of methode van de child-klasse identiek zijn aan die van de parent-klasse. Het enigste dat je moet doen is het sleutelwoord `virtual` vervangen door `override` in de child-klasse.

Er zijn wel enkele voorwaarden waar de parent-klasse zich aan moet houden om een lid `virtual` te maken. Zo kan je alleen `public` properties of methoden die niet statisch zijn als `virtual` worden ingesteld.

15.3.2 Sleutelwoord `base`

Ondertussen weet je al dat het sleutelwoord `base` gebruikt kan worden om de constructor van de basis-klasse aan te roepen vanuit de constructor van een afgeleide klasse.

Daarnaast kan `base` gebruikt worden om bij de `override` van een eigenschap of methode van een subklasse toch de **implementatie van de superklasse toe te passen**. Dit is handig voor wanneer je de implementatie van de superklasse verder wilt uitbreiden in de subklasse.

Voorbeeld:

```
class HorecaPersoneel
{
    protected double Loon { get; set; } = 0;
    public virtual void OntvangLoon()
    {
        Loon += 1900;
    }
}

class ZaalVerantwoordelijke : HorecaPersoneel
{
    public override void OntvangLoon()
    {
        base.OntvangLoon();
        Loon += 150;
    }
}
```

Wanneer in de superklasse Horecapersoneel het basisloon wordt aangepast, wordt deze aanpassing ook automatisch doorgevoerd naar de subklasse Zaalverantwoordelijke dankzij het sleutelwoord `base`.



Dit voorbeeld is trouwens veel veiliger dan wanneer we in de subklasse `Zaalverantwoordelijke` het statement `Loon += 1900 + 150;` zouden gebruiken. Je zou dan namelijk de lonen van het horecapersoneel en de zaalverantwoordelijke apart moeten aanpassen als de lonen veranderen. Bovendien bestaat dan de kans dat je het loon vergeet aan te passen in één van de klassen. Dat is allemaal dubbel werk en foutgevoeliger, wat indruist tegen de principes van OOP.

15.4 Abstractie

Abstractie is een fundamenteel programmeerconcept dat **alleen essentiële informatie** toont aan de gebruiker, terwijl de interne en ingewikkelde details verborgen worden gehouden. In C# wordt abstractie bereikt door middel van abstracte klassen en interfaces.

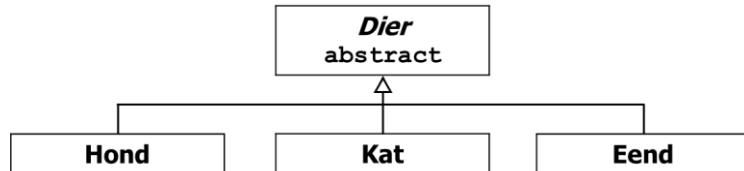
We kunnen klassen, eigenschappen en methoden abstract maken door ze te voorzien van het gelijknamige sleutelwoord `abstract`. Van zodra je een lid abstract maakt, moet de klasse ook abstract worden.

15.4.1 Abstracte klassen

Een abstracte klasse is een klasse waarvan je **geen objecten** kan **aanmaken**, maar die wel dienst kan doen als superklasse om te worden uitgebreid door andere klassen.

Neem nu klassen zoals `Dier`, `Voertuig`, `Apparaat`, `Figuur`, ... Deze klasse hebben **een duidelijke bestaansreden** in een klasse-hiërarchie omdat ze aardig wat gemeenschappelijke code kunnen bevatten. Maar heb je ooit al eens afgevraagd wat we ons moeten voorstellen bij een

object van bijvoorbeeld het type Dier? Wat voor een dier is het dan juist? Een vogel? Een kat? Een olifant? Je merkt dus al dat er abstracte zaken zijn die gebruikt worden om een **algemene verzameling** van objecten voor te stellen **met gemeenschappelijke eigenschappen en gedragingen**. Echter, objecten maken van deze klassen is zinloos en zelfs onrealistisch. Er bestaan immers geen Dier-, Voertuig-, Figuur-, ... objecten in de echte wereld.



Syntax:

```

public abstract class KlasseNaam {
    //klasse-omschrijving (class body)
}
  
```

Voorbeeld:

```

public abstract class Dier
{
    public string Naam { get; set; }
}

public class Hond : Dier
{
    //klasse-omschrijving (class body)
}
  
```

15.4.2 Abstracte methoden

Abstracte klassen kunnen abstracte methoden bevatten die zelf geen implementatie hebben en die geïmplementeerd moeten worden door de afgeleide klassen. Sterker nog, het **verplicht** afgeleide klassen om de **methode aan te passen of uit te breiden**. Dit in tegenstelling tot het sleutelwoord **virtual** dat geen verplichting oplegt, maar wel aangepast mag worden.

Voorbeeld:

```

abstract class Dier
{
    public abstract string MaakGeluid();
}

class Hond : Dier
{
    public override string MaakGeluid()
    {
        return "Woef!";
}
  
```

In dit voorbeeld heeft de abstracte methode MaakGeluid() geen implementatiecode, want wat voor geluid maakt een dier nu juist? Dat kunnen we niet zeggen zonder de diersoort te kennen. Dit is een goed voorbeeld van de bestaansreden van abstracte methoden. We willen namelijk dat elke afgeleide klasse een eigen implementatie geeft aan deze methode, maar om dat te kunnen doen is eerst meer specifieke informatie nodig.

Kortom, het volstaat om enkel de methodesignatuur in te geven van abstracte methoden.

15.4.3 Abstracte eigenschappen

Net als bij abstracte methoden, zijn afgeleide klassen **verplicht** om een **eigen implementatie** van een abstracte property te schrijven.

Voorbeeld:

```
abstract class Dier
{
    abstract public int AantalPoten { get; }

}

class Hond : Dier
{
    public override int AantalPoten
    {
        get { return 4; }
    }
}
```

Tijdens het aanmaken van een abstracte property moet je aangeven of het een read-only (getter), write-only (setter), of een combinatie van de twee (getter en setter) gaat.

Voorbeelden:

```
public abstract int Authenticatiecode { get; }      //read-only
public abstract int Wachtwoord { set; }             //write-only
public abstract int Gebruikersnaam { get; set; }    //combinatie
```

16.1 **Associaties**

In object-georiënteerd programmeren zijn er naast overerving nog andere manieren om relaties tussen klassen te modelleren, en bijgevolg code te hergebruiken, zoals **associatie**, **aggregatie** en **compositie**.

- i** Eigenlijk is de kans groot dat je deze concepten reeds hebt gebruikt zonder er bij stil te hebben gestaan wat ze juist inhouden. Bovendien bieden ze vaak meer flexibiliteit en bruikbaarheid dan overerving in complexere programma's. Juist daarom is het belangrijk om deze concepten eens nader te bekijken en te vergelijken met overerving, voordat we dieper ingaan op andere OOP concepten.

Associatie (of association) is de meest **algemene vorm** van een **relatie tussen twee klassen**. Het vertegenwoordigt een situatie waarin objecten van de ene klasse verbonden zijn met objecten van een andere klasse. Dit werkt in beide richtingen, maar dit hoeft niet altijd het geval te zijn. Eigenlijk komt het erop neer dat de ene klasse gebruik maakt van de functionaliteit uit een andere klasse.

Voorbeeld: objecten van de klassen Student en Cursus hebben een associatie omdat een student zich kan inschrijven voor meerdere cursussen en een cursus meerdere studenten kan hebben.



```

class Student
{
    public string Naam { get; set; }
    public List<Cursus> Cursussen { get; set; } //List van klasse Cursus

    public Student(string naam)
    {
        Naam = naam;
        Cursussen = new List<Cursus>();
    }
}

class Cursus
{
    public string Titel { get; set; }
    public List<Student> Studenten { get; set; } //List van klasse Student

    public Cursus(string titel)
    {
        Titel = titel;
        Studenten = new List<Student>();
    }
}
  
```

16.2 Aggregatie en compositie

Aggregatie (of **aggregation**) en compositie (of **composition**) zijn specifieke vormen van associatie waarbij we **een object in een ander object gebruiken**. Het object van de ene klasse heeft dan intern een referentie naar het object van de andere klasse en kan daardoor gebruik maken van diens functionaliteit.

Kortom, het is het gebruik van objecten als variabelen in een andere klasse.

16.2.1 Aggregatie

Aggregatie beschrijft een relatie waarbij **objecten onafhankelijk van elkaar kunnen bestaan**. Het kan worden gezien als een relatie tussen een geheel en zijn delen, maar waarbij de delen onafhankelijk kunnen bestaan van het geheel. Kortom, objecten met een aggregatie relatie zijn “**een deel van**” een ander object.

Voorbeeld: objecten van de klassen Team en Speler hebben een aggregatie relatie omdat spelers bij een team kunnen horen, maar ze ook kunnen ook op zichzelf bestaan zonder deel uit te maken van team. Het is niet zo dat een speler opeens ophoudt met bestaan wanneer die het team ploeg, of wanneer het team ophoudt met bestaan.



```
class Team
{
    public string Naam { get; set; }
    public List<Speler> Spelers { get; set; }

    public Team(string naam)
    {
        Naam = naam;
        //Spelers kunnen worden toegevoegd aan het team, maar worden niet
        //geinitialiseerd in de klasse Team
        Spelers = new List<Speler>();
    }
}

class Speler
{
    public string Naam { get; set; }

    public Speler(string naam)
    {
        Naam = naam;
    }
}
```

16.2.2 Compositie

Compositie is een sterkere vorm van aggregatie, want in deze relatie is de **levensduur van de delen wel afhankelijk van de levensduur van het geheel**. Als het geheel wordt vernietigd, worden de delen dus ook vernietigd. Kortom, een object met compositie relatie “**bestaat volledig**” uit andere objecten.

Voorbeeld: objecten van de klassen Huis en Kamer hebben een compositie relatie omdat kamers niet zonder het huis kunnen bestaan. Wanneer een huis wordt vernietigd, verdwijnen de kamers dus ook.



```
class Huis
{
    public List<Kamer> Kamers { get; set; }

    public Huis()
    {
        Kamers = new List<Kamer>();
    }

    public void VoegKamerToe(string naam)
    {
        //een nieuwe kamer wordt geïnitialiseerd in de klasse Huis
        Kamers.Add(new Kamer(naam));
    }
}

class Kamer
{
    public string Naam { get; set; }

    public Kamer(string naam)
    {
        Naam = naam;
    }
}
```

16.2.3 Heeft-een relatie

Hopelijk is het je al opgevallen dat alle klassen in de gegeven voorbeelden sowieso niet in aanmerking zouden komen voor overerving. Overerving is namelijk alleen van toepassing bij zinvolle "is een"-relaties tussen klassen, wat hier duidelijk niet het geval is.

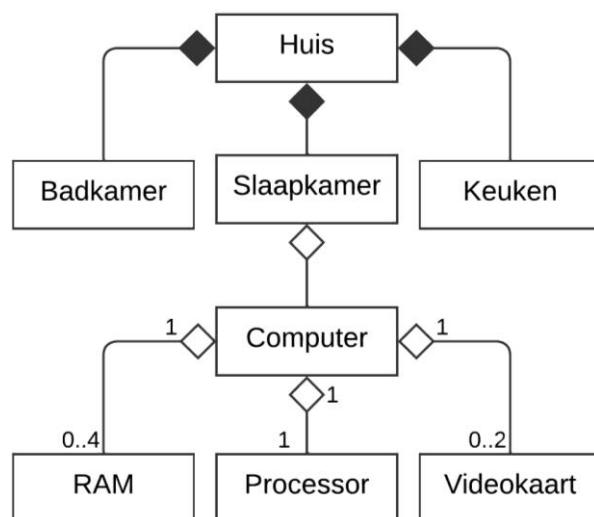
Wat we hier wél kunnen opmerken is dat we de relaties van deze klassen allemaal kunnen beschrijven als een "heeft een"- / "heeft meerdere"-relatie (of "has a" / "has multiple"-relation):

- Een student heeft één of meerdere cursussen, en vice versa.
- Een team heeft één of meerdere spelers.
- Een huis heeft één of meerdere kamers.

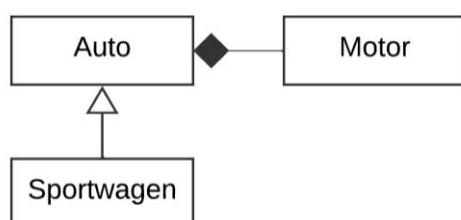
Compositie en aggregatie kunnen we dus detecteren door een "heeft-een/meerdere"-relatie te beschrijven tussen klassen. Deze relaties geven aan dat een object bestaat uit een of meer andere objecten.

16.2.4 Combinaties van relaties

Uiteraard zijn ook combinaties van relaties zoals compositie en aggregatie mogelijk. Neem bijvoorbeeld een huis met in de slaapkamer een computer. Een computer kan je uit een brandend huis reden, maar de kamer zelf uiteraard niet. In het onderstaande UML diagram staat bij de computer ook beschreven hoeveel objecten in een ander object opgenomen mogen worden.



Ook de combinatie van overerving en associatie is perfect mogelijk. Neem bijvoorbeeld de parent-klasse Auto die een object van de klasse Motor heeft ingebouwd. Aangezien de child-klasse Sportwagen overerft van Auto, heeft een sportwagen bijgevolg ook een motor ingebouwd.



Uiteraard zijn er nog veel meer combinaties mogelijk. Dit zijn slechts een paar voorbeelden om je een idee te geven van de mogelijkheden.

16.3 Compositie en aggregatie toepassen

In de praktijk zijn er weinig verschillen tussen aggregatie en compositie. Het kan op verschillende manieren worden toegepast, afhankelijk van de specifieke behoeften van de applicatie. Er is met andere woorden **geen exacte toepassingsstrategie** die je kan volgen. Laat je dus niet beperken door de gegeven voorbeelden; er zijn namelijk veel manieren om compositie en aggregatie toe te passen.

16.3.1 Via instantievariabelen (velden)

Compositie: een Motor-object wordt ogenblikkelijk gecreëerd als een essentieel onderdeel van een Auto-object.

```
class Auto
{
    private Motor motorblok = new Motor();
```

16.3.2 Via constructors

Compositie: het Motor-object wordt op hetzelfde moment gecreëerd als het Auto-object.

```
class Auto
{
    private Motor motorblok;

    public Auto()
    {
        motorblok = new Motor();
    }
}
```

Via de constructors kunnen we trouwens **meer controle inbouwen**.

Voorbeeld 1: controleren of een motorblok moet worden voorgemonteerd:

```
class Auto
{
    private Motor motorblok;

    public Auto(bool motorIsVoorgemonteerd)
    {
        if (motorIsVoorgemonteerd)
        {
            motorblok = new Motor();
        }
        else
        {
            motorblok == null;
        }
    }
}
```

De lijn `motorblok == null` lijkt overbodig omdat de variabele `motorblok` standaard `null` is. Toch is het aangeraden om dit expliciet te doen! Hiermee zeg je immers uitdrukkelijk: "als we via de overloaded constructor een auto aanmaken en er is geen voormontage vereist is, dan zit er geen motorblok in de auto". Zo garanderen we namelijk dat het motorblok niet door voorgaande code een objectreferentie bevat.

Voorbeeld 2: controleren of een adres is gegeven:

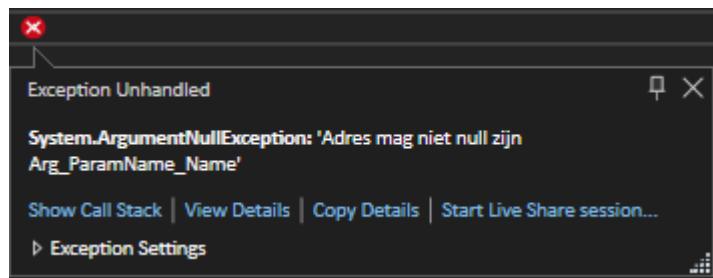
```
class Persoon
{
    public string Naam { get; }
    public Adres Adres { get; }

    public Persoon(string naam, Adres adres)
    {
        if (adres == null)
        {
            throw new ArgumentNullException(nameof(adres), "Adres mag niet
            null zijn!");
        }
        Naam = naam;
        Adres = adres;
    }
}
```

Zoals je ziet is het ook ideaal om een exception op te gooien. In dit geval voor als het adres null blijkt te zijn.



Het sleutelwoord `nameof` werd in dit voorbeeld gebruikt om een aangepaste foutmelding te tonen tijdens de uitvoering van het programma. Anders zou er de standaard boodschap "Value cannot be null." staan in plaats van "Adres mag niet null zijn".



16.3.3 Via properties

Compositie: een Motor-object wordt ogenblikkelijk gecreëerd als een essentieel onderdeel van een Auto-object, maar deze keer via een autoproperty.

```
class Auto
{
    public Motor Motorblok { get; set; } = new Motor();
```

Aggregatie: Tot hiertoe hebben we alleen maar voorbeelden gezien van compositie waarbij het Motor-object verdwijnt als het Auto-object verdwijnt. Willen we echter de motor in auto's kunnen verwijderen of vervangen, dan is het interessanter om deze in te bouwen via aggregatie.

Hiervoor moeten we op een andere, externe plaats het Motor-object aanmaken en deze vervolgens in het Auto-object plaatsen.

```
public class Auto
{
    public Motor Motorblok { get; set; }
}

internal class Program
{
    static void Main(string[] args)
    {
        Motor mijnMotorblok = new Motor()
        Auto mijnAuto = new Auto();
        mijnAuto.Motorblok = mijnMotorblok;
    }
}
```

Op deze manier zal de motor niet verdwijnen wanneer de auto wordt verwijderd. Bovendien kan je, zoals reeds aangegeven, zo gemakkelijk de motor in een auto vervangen. Laat ons dit ook maar eens illustreren aan de hand van een voorbeeld.

Voorbeeld: We willen een defecte motor in onze bedrijfswagen vervangen met de motor uit onze gezinswagen (want we moeten op het werk geraken met de bedrijfswagen).

```
static void Main(string[] args)
{
    //Haal beide wagens erbij
    Auto gezinswagen = new Auto();
    Auto bedrijfswagen = new Auto();

    //Verwijder defecte motor uit de bedrijfswagen
    bedrijfswagen.Motorblok = null;

    //Plaats de motor van de gezinswagen in de bedrijfswagen
    bedrijfswagen.Motorblok = gezinswagen.Motorblok;

    //De gezinswagen heeft nu geen motor meer
    gezinswagen.Motorblok = null;
}
```

16.3.4 “Heeft meerdere”-relaties toepassen

Wanneer een object meerdere objecten van een specifiek type heeft, gebruiken we een array of een List als compositie-object. Laten we verder gaan op het voorbeeld van de auto en meerdere auto's in een garage stoppen.

Voorbeeld 1: met een **array** (met 50 parkeerplaatsen).

```
class Auto { }
```

```

class Garage
{
    public Auto[] AlleAutos { get; set; } = new Auto[50];

    public void AutoToevoegen(Auto auto, int nummmerParkeerplaats)
    {
        AlleAutos[nummmerParkeerplaats] = auto;
    }
}

```

Voorbeeld 2: met een List<>

```

class Auto { }

class Garage
{
    public List<Auto> AlleAutos { get; private set; } = new List<Auto>();

    public void AutoToevoegen(Auto auto, int nummmerParkeerplaats)
    {
        AlleAutos.Insert(nummmerParkeerplaats, auto);
    }
}

```



Dit voorbeeld met de List is vanuit een OOP-standpunt een slechte aanpak. Het vereist namelijk dat programmeurs, die de klasse Garage gebruiken, weten dat intern met een List wordt gewerkt.

Om toch het principe van abstractie zo correct mogelijk door te voeren, is het interessanter om AlleAutos af te schermen, liefst met private, en alleen te benaderen met behulp van methoden. Zo behouden we meer controle.

Stel je maar eens voor dat iemand de collectie volledig leeg zou maken omdat hij/zij de methode Clear() heeft aanroepen. Dit zou grote gevolgen kunnen hebben op de werking van het programma! Het is dan ook omwille van die reden dat we de methode AutoToevoegen() hebben toegevoegd aan onze code, in plaats van de methode Insert() van de List te gebruiken.

In de methode Main() kunnen we vervolgens auto's toevoegen aan de garage.

```

Auto bmwI5 = new Auto();
Auto audiR8 = new Auto();
Garage garageVanMossel = new Garage();

garageVanMossel.AlleAutos[0] = bmwI5;           //aggregatie
garageVanMossel.AutoToevoegen(audiR8, 1);       //aggregatie
garageVanMossel.AlleAutos[2] = new Auto();        //compositie
garageVanMossel.AutoToevoegen(new Auto(), 3);    //compositie

```

16.4 Compositie- en aggregatie-objecten gebruiken

We bekijken even terug de klasse Motor die een autoproperty Vermogen (uitgedrukt in pK) heeft. Omdat een motorblok dankzij compositie is ingebouwd in een auto, kan de klasse Auto deze property ook gebruiken.

```
class Auto
{
    private Motor motorblok = new Motor();

    public override string ToString()
    {
        if (motorblok != null)
        {
            return $"Vermogen: {motorblok.Vermogen} pK";
        }
        else
        {
            return "Er is geen motorblok aanwezig";
        }
    }
}
```

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`. Maak er dus zeker een goede gewoonte van om zoveel mogelijk te controleren op null telkens je het object gaat gebruiken bij compositie en aggregatie.

16.5 Code hergebruik: overerving of compositie?

In de praktijk worden compositie en aggregatie veel vaker gebruikt dan overerving omdat ze een "heeft een"-relatie aanduiden, terwijl overerving beperkt is tot "is een"-relaties. Associaties laten ons dus toe om verschillende soorten objecten met elkaar te laten samenwerken. Bovendien creëert overerving vaak starre klasse-hiërarchieën die achteraf moeilijk aangepast of uitgebreid kunnen worden.

We illustreren dit laatste aan de hand van een kortverhaal (geschreven met behulp van ChatGPT):

Stel je een bibliotheek voor die begon met het beheren van boeken. In het begin gebruikte de bibliotheek overerving om verschillende soorten boeken te structureren. De basisklasse Book had subklassen zoals HardcoverBook, PaperbackBook en EBook. Deze aanpak werkte goed voor de initiële collectie van boeken.

Naarmate de tijd vorderde, groeide de vraag naar verschillende soorten media. Leden wilden niet alleen boeken, maar ook tijdschriften, audioboeken, en video's lenen. De eerste uitdaging ontstond bij het toevoegen van tijdschriften. Omdat overerving een "is-een" relatie vereist, was het moeilijk om tijdschriften als subklasse van boeken te definiëren. Tijdschriften hebben andere eigenschappen en structuren dan boeken, wat resulteerde in een complexe en onlogische hiërarchie.

De situatie verergerde toen de bibliotheek video's wilde toevoegen. De bestaande hiërarchie was niet ontworpen om videomateriaal te bevatten. De bibliotheek stond voor de keuze: moesten video's worden geïntegreerd als een subklasse van boeken, of moest er een volledig nieuwe tak in de hiërarchie worden gecreëerd? Beide opties leidden tot complicaties. Het opnemen van video's als subklasse van boeken zou een kunstmatige en onlogische hiërarchie creëren, terwijl een nieuwe tak zou leiden tot een onsaamhangend en moeilijk beheersbaar systeem.

Deze problemen benadrukken de beperkingen van overerving in een dynamische en uitbreidende context. De rigide structuur van overerving maakte het moeilijk om flexibel te reageren op veranderende behoeften en nieuwe mediatypes. De bibliotheek raakte vast in een starre klasse-hiërarchie die moeilijk aan te passen en te onderhouden was.

Om deze uitdagingen te overwinnen, besloot de bibliotheek om over te stappen op een benadering gebaseerd op compositie en aggregatie. In plaats van media te structureren via een hiërarchie van klassen, werden media-items samengesteld uit verschillende componenten zoals TextContent, AudioContent, en VideoContent. Hierdoor konden ze eenvoudig nieuwe mediatypes toevoegen door relevante componenten te combineren.

Met deze nieuwe benadering kon de bibliotheek gemakkelijk tijdschriften, audioboeken en video's integreren in hun collectie. De flexibiliteit van compositie en aggregatie stelde hen in staat om een modulair en schaalbaar systeem te creëren, waardoor ze beter konden inspelen op de veranderende vraag en toekomstige uitbreidingen.

Dit scenario toont aan dat compositie en aggregatie, door hun vermogen om een "heeft-een"-relatie te modelleren, vaak beter geschikt zijn voor situaties waar flexibiliteit en aanpassingsvermogen cruciaal zijn. Overerving, met zijn "is-een"-relatie, kan namelijk snel leiden tot starre en moeilijk te onderhouden systemen wanneer de complexiteit en diversiteit van objecten toenemen.

Conclusie:

Met associaties kunnen programmeurs flexibele en onderhoudbare systemen ontwerpen. De code kan hierdoor beter gestructureerd en modulair worden geschreven, wat het eenvoudiger maakt om deze te begrijpen, te testen en uit te breiden.

Overerving blijft echter een fantastische tool voor eenvoudige, minder complexe programma's die niet meer uitgebreid of aangepast hoeven te worden. Het is dus zeker niet zinloos om overerving te leren, want in de juiste situaties kan het een wereld van verschil maken. Overerving kan immers ook leiden tot mooie, onderhoudbare en gemakkelijk te begrijpen code.

Uiteindelijk moet je altijd rekening houden met de interessante mogelijkheden die combinaties van overerving en associaties bieden. Overweeg daarom de verschillende oplossingsstrategieën en houd rekening met eventuele aanpassingen die je later in het programma zou moeten doorvoeren.



het gebruik van interfaces had ook een uitstekende oplossing kunnen zijn voor het probleem in het voorbeeld van de bibliotheek. Meer hierover in het hoofdstuk over interfaces.

17.1

Polymorfisme (bis)

In het hoofdstuk over overerving en klasse-hiërarchie hebben we het reeds gehad over polymorfisme, wat letterlijk “veelvormigheid” of “meerdere vormen” betekent. Het is een belangrijk en krachtig OOP-concept dat ervoor zorgt dat verschillende klassen hun **methoden** kunnen **delen** en hun **eigen specifieke implementatie** van deze methoden kunnen bieden.

Nu je echter weet dat klassen elkaars functionaliteit kunnen gebruiken via associaties, zal je merken dat polymorfisme niet alleen gebonden is tot afgeleide klassen. Hoog tijd dus om te ontdekken wat polymorfisme nog allemaal te bieden heeft.

17.1.1 Objecten en polymorfisme

Objecten van subklassen kunnen beschouwd en behandeld worden als objecten van hun superklassen. Dit heeft tot gevolg dat een object van een superklasse zich kan voordoen als een object van zijn verschillende subklassen.

We illustreren dit aan de hand van het klassieke voorbeeld met de diersoorten die elks hun eigen, unieke geluid produceren.

```
abstract class Dier
{
    public abstract string MaakGeluid();
}

class Hond : Dier
{
    public override string MaakGeluid()
    {
        return "Woef!";
    }
}

class Kat : Dier
{
    public override string MaakGeluid()
    {
        return "Miauw!";
    }
}
```

Dankzij polymorfisme kunnen we nu objecten van de subklassen Hond en Kat in een variabele van het type Dier bewaren, maar ze toch hun eigen geluid laten reproduceren:

```
Dier dier1 = new Hond();
Dier dier2 = new Kat();
Console.WriteLine(dier1.MaakGeluid()); //Woef!
Console.WriteLine(dier2.MaakGeluid()); //Miauw!
```

Het is belangrijk om op te merken dat deze objecten **alleen toegang hebben tot de eigenschappen en methoden die zijn beschreven in de superklasse** Dier vanwege hun declaratie. Toch kan elk object zijn eigen implementatie van de methode MaakGeluid() uitvoeren vanuit zijn subklasse dankzij override en polymorfisme.

17.1.2 Collecties en polymorfisme

Dit principe kunnen we ook toepassen op collecties zoals arrays en Lists door objecten van verschillende subklassen te verzamelen in een collectie van een gemeenschappelijke superklasse.

```
List<Dier> huisdieren = new List<Dier>();
huisdieren.Add(new Hond());
huisdieren.Add(new Kat());

foreach (var dier in huisdieren)
{
    Console.WriteLine(dier.MaakGeluid());
}
```

Aangezien de methode MaakGeluid() in de superklasse Dier gedefinieerd staat, is deze methode ook beschikbaar voor alle objecten van zijn subklassen. Daardoor kunnen we van elk dier hun unieke geluid produceren, ongeacht welk soort dier het is. Kortom, we gebruiken onze objecten even als één geheel, zonder dat ze allemaal van hetzelfde type moeten zijn.

17.1.3 Upcasting en downcasting

De sleutelwoorden `is` en `as` kunnen zeer nuttig zijn in polymorfisme, waar verschillende klassen een gemeenschappelijke superklasse of dezelfde interface (zie volgend hoofdstuk) delen.

17.1.3.1 Het sleutelwoord `is`

Met behulp van het sleutelwoord `is` kan je controleren of een object van een bepaald type is voordat je een bepaalde handeling uitvoert. Dit is handig als je code moet uitvoeren die specifiek is voor een bepaald type, maar je zeker moet zijn dat het object van dat type is om een exception te voorkomen. Het resultaat is altijd een booleanse waarde: `true` or `false`.

Voorbeeld van klassen waarvan we willen controleren of het dieren zijn:

```
class Dier { }
class Hond : Dier { }
class Persoon { }
```

In de methode Main() kunnen we nu met `is` controleren welke objecten van het type Dier zijn:

```
Hond lassie = new Hond();
Persoon billGates = new Persoon();
if (lassie is Dier)
    Console.WriteLine("Lassie is een dier");
if (billGates is Dier)
    Console.WriteLine("Bill Gates is een dier");
```

Dit zal als uitvoer alleen de volgende boodschap geven:

```
Lassie is een dier
```

Met polymorfisme het sleutelwoord `is` pas echt interessant, want nu kunnen we bijvoorbeeld alleen de honden iets laten doen in een verzameling van dieren:

```
List<Dier> Huisdieren = new List<Dier>();
Huisdieren.Add(new Kat());
Huisdieren.Add(new Hond());
Huisdieren.Add(new Dier());

foreach (var dier in Huisdieren)
{
    if (dier is Hond)
        //statements die iets doen met de huidige hond
}
```

17.1.3.2 Het sleutelwoord `as`

Tot nu toe hebben we altijd gebruik gemaakt van casting om een object van het ene type om te zetten naar het andere type:

```
Hond rex = new Hond();
Dier jack = (Dier)rex;
```

Het probleem bij casting is dat als de conversie niet mogelijk is, er een uitzondering gegenereerd wordt.

Gelukkig kunnen we met het sleutelwoord `as` een object op een veilige manier van type converteren, want als de conversie mislukt, wordt `null` gereturneerd in plaats van een uitzondering op te werpen.

De bovenstaande code herschrijven we als volgt:

```
Hond rex = new Hond();
Dier jack = rex as Dier;
```

Rex zal de waarde `null` krijgen als blijkt dat de klasse `Hond` geen subklasse is van `Dier`. Als dit echter wel het geval is, mag je er zeker van zijn dat de type conversie succesvol is.

De bovenstaande code kan je dan uitbreiden als volgt:

```
Hond rex = new Hond();
Dier jack = rex as Dier;

if (jack != null)
    //statements die "Dier-zaken" doen
```

17.1.4 Polymorfisme in de praktijk

Stel dat je een garage hebt waarin je een eenvoudig onderhoudssysteem voor voertuigen wilt simuleren.

De klasse Garage heeft toegang tot verschillende voertuigen die elk specifieke onderhoudstaken nodig hebben. Zonder de voordelen van polymorfisme zou onze klasse er zo kunnen uitzien:

```
public class Garage
{
    public Auto MijnAuto { get; set; } = new Auto();
    public Motorfiets MijnMotorfiets { get; set; } = new Motorfiets();
    public Vrachtwagen MijnVrachtwagen { get; set; } = new Vrachtwagen();

    public void Onderhoud()
    {
        // voertuigen krijgen hun onderhoud
        // Auto: Onderhoudstaken
        MijnAuto.VerversOlie();
        MijnAuto.ControleerBandenspanning();
        MijnAuto.ReinigInterieur();

        // Motorfiets: Onderhoudstaken
        MijnMotorfiets.SmeerKetting();
        MijnMotorfiets.ControleerRemmen();
        MijnMotorfiets.ControleerBanden();

        // Vrachtwagen: Onderhoudstaken
        MijnVrachtwagen.LaadPallets();
        MijnVrachtwagen.VerversOlie();
        MijnVrachtwagen.ControleerLading();
    }
}

public class Auto
{
    public void VerversOlie()
    { Console.WriteLine("Auto: Olie ververst!"); }
    public void ControleerBandenspanning()
    { Console.WriteLine("Auto: Bandenspanning gecontroleerd!"); }
    public void ReinigInterieur()
    { Console.WriteLine("Auto: Interieur gereinigd!"); }
}

public class Motorfiets
{
    public void SmeerKetting()
    { Console.WriteLine("Motorfiets: Ketting gesmeerd!"); }
    public void ControleerRemmen()
```

```

    { Console.WriteLine("Motorfiets: Remmen gecontroleerd!"); }
    public void ControleerBanden()
    { Console.WriteLine("Motorfiets: Banden gecontroleerd!"); }
}

public class Vrachtwagen
{
    public void LaadtPallets()
    { Console.WriteLine("Vrachtwagen: Pallets geladen!"); }
    public void VerversOlie()
    { Console.WriteLine("Vrachtwagen: Olie ververst!"); }
    public void ControleerLading()
    { Console.WriteLine("Vrachtwagen: Lading gecontroleerd!"); }
}

```

In dit voorbeeld moet de garage (of de programmeur van deze klasse) veel specifieke kennis hebben van de verschillende voertuigen. De bovenstaande code is dus zeer slecht en vloekt tegen het abstractie-principe van OOP. Onze klasse moet namelijk te veel weten over andere klassen, wat vermeden moet worden. Telkens er iets verandert in een specifieke voertuigklasse, moet dit ook in de klasse Garage aangepast worden.

Dankzij polymorfisme en overerving kunnen we dit alles veel mooier oplossen!

Ten eerste: We verplichten alle voertuigen om over te erven van de abstracte klasse Voertuig die één abstracte methode VoerOnderhoudUit() heeft:

```

public abstract class Voertuig
{
    public abstract void VoerOnderhoudUit();
}

public class Auto : Voertuig
{
    public override void VoerOnderhoudUit()
    {
        VerversOlie();
        ControleerBandenspanning();
        ReinigInterieur();
    }
    public void VerversOlie() { ... }
    public void ControleerBandenspanning() { ... }
    public void ReinigInterieur() { ... }
}

public class Motorfiets : Voertuig
{
    public override void VoerOnderhoudUit()
    {
        SmeerKetting();
    }
}

```

```

        ControleerRemmen();
        ControleerBanden();
    }
    public void SmeerKetting() { ... }
    public void ControleerRemmen() { ... }
    public void ControleerBanden() { ... }
}

public class Vrachtwagen : Voertuig
{
    public override void VoerOnderhoudUit()
    {
        LaadtPallets();
        VerversOlie();
        ControleerLading();
    }
    public void LaadtPallets(){ ... }
    public void VerversOlie() { ... }
    public void ControleerLading() { ... }
}

```

Ten tweede: Het leven van de garage wordt plots véél makkelijker. Het kan gewoon de methode VoerOnderhoudUit() aanroepen van elk voertuig:

```

public class Garage
{
    public Voertuig MijnAuto { get; set; } = new Auto();
    public Voertuig MijnMotorfiets { get; set; } = new Motorfiets();
    public Voertuig MijnVrachtwagen { get; set; } = new Vrachtwagen();

    public void Onderhoud()
    {
        MijnAuto.VoerOnderhoudUit();
        MijnMotorfiets.VoerOnderhoudUit();
        MijnVrachtwagen.VoerOnderhoudUit();
    }
}

```

Ten derde: We kunnen de klasse Garage nog verder verbeteren door een array of List<Voertuig> te gebruiken, zodat het ook niet steeds de "namen" van zijn voertuigen hoeft te kennen.

Dankzij polymorfisme mag dit:

```

public class Garage
{
    public List<Voertuig> AlleVoertuigen { get; set; } = new
List<Voertuig>();

```

```

public Garage()
{
    AlleVoertuigen.Add(new Auto());
    AlleVoertuigen.Add(new Motorfiets());
    AlleVoertuigen.Add(new Vrachtwagen());
}

public void Onderhoud()
{
    foreach (Voertuig voertuig in AlleVoertuigen)
    {
        voertuig.VoerOnderhoudUit();
    }
}

public class Program
{
    public static void Main()
    {
        Garage mijnGarage = new Garage();
        mijnGarage.Onderhoud();
    }
}

```

Nu kan de garage elk type voertuig gemakkelijk beheren en uitbreiden zonder de code van de klasse Garage aan te passen, wat het systeem flexibeler en onderhoudsvriendelijker maakt. Polymorfisme zorgt hier dus voor een elegantere en meer schaalbare oplossing.

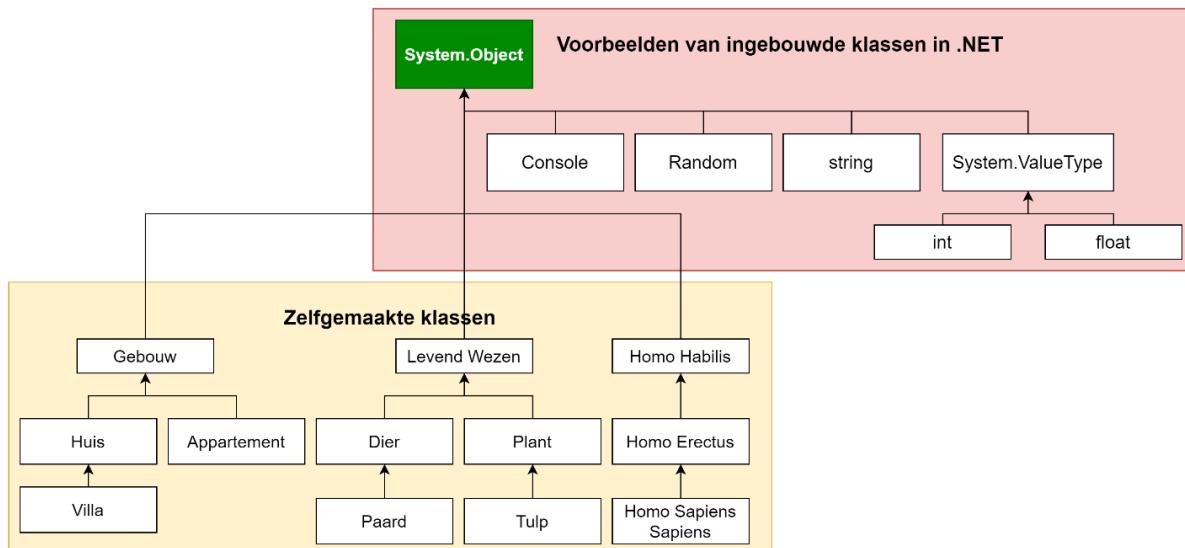
-  Dit voorbeeld is volledig gebaseerd op een voorbeeld uit het boek Zie Scherp Scherper (Dams, 2022) als antwoord op de vraag: "What is polymorphism, what is it for, and how is it used?".

Ik raad je aan om het originele voorbeeld ook eens te bekijken op de website:
<https://apwt.gitbook.io/zie-scherp-scherper/h16-polymorfisme/polypraktijd>

17.2 De oerklasse Object

C# is een programmeertaal die speciaal is ontworpen om optimaal gebruik te maken van object georiënteerd programmeren. Daarom is de taal **volledig opgebouwd uit klassen**.

Bovenaan deze klasse-hiërarchie staat de klasse Object, waarvan alle andere klassen in C# direct of indirect erven. Dit geldt voor zowel zelfgemaakte als alle ingebouwde klassen in het .NET-framework. Zelfs waardetypes erven uiteindelijk van de klasse Object, via de tussenliggende klasse ValueType.



Bron: Zie Scherp Scherper (Dams, 2022)

17.2.1 Impliciete overerving

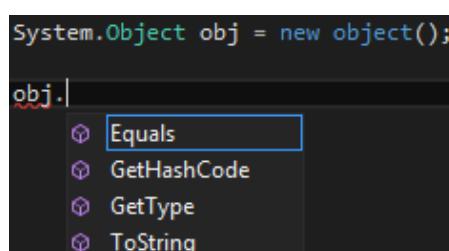
Wanneer je een klasse aanmaakt gebeurt er namelijk een impliciete overerving van de klasse Object. Dit wilt zeggen dat de compiler intern de volgende code zal toevoegen aan de declaratie van de klasse zoals bij een expliciete overerving:

```
class Voorbeeld : Object
{ }
```

Dit kan je trouwens ook expliciet zelf schrijven, maar dat heeft weinig zin omdat deze overerving toch automatisch gebeurd.

17.2.2 Methoden van de klasse Object

De klasse Object beschikt over enkele methoden die door alle subklassen worden overgeërfd. Je zal ze waarschijnlijk al wel gezien hebben wanneer je de methoden van een object van een nieuwe klasse hebt willen aanroepen.



In deze tabel worden de meest gebruikte methoden van de klasse Object weergegeven:

Returntype	Methode	Beschrijving
bool	Equals(obj o)	Geeft aan of twee objecten aan elkaar gelijk zijn. Dit is het geval indien de referentievariabelen gelijk zijn.
int	GetHashCode()	Geeft de unieke hashcode van het object; nuttig om o.a. te sorteren.
Type	GetType	Geeft het datatype (de klasse) van het object terug.
String	ToString()	Geeft een string terug waarin het object beschreven wordt.

De meeste van deze methoden zijn aanpasbaar met een override. Pas nadat je deze methoden een eigen implementatie hebt gegeven, zullen ze nuttig zijn voor jouw klassen en objecten. Sterker nog, heel wat .NET bibliotheken rekenen er zelfs op dat je deze methoden op de juiste manier hebt aangepast, zodat ook jouw nieuwe klassen perfect kunnen samenwerken met deze bibliotheken.

 De klasse Object bevat ook nog de methoden ReferenceEquals(), MemberWiseClone() en Finalize(), maar deze zijn niet virtual en behandelen we dus niet in deze cursus .

Laten we eens zien hoe we deze methoden kunnen gebruiken en/of aanpassen. Dit doen we aan de hand van de klasse Auto dat de autoproperties Merk, Model en Bouwjaar bevat.

```
class Auto
{
    public string Merk { get; set; }
    public string Model { get; set; }
    public int Bouwjaar { get; set; }
}
```

17.2.2.1 De methode GetType()

Elke instantie van Auto kan de methode GetType() gebruiken om als uitvoer de namespace gevuld door het type van het object op het scherm te tonen.

```
Auto mijnAuto = new Auto();
Console.WriteLine(mijnAuto.GetType());
```

Dit zal als uitvoer de volgende boodschap geven:

```
MyApp.Auto
```

Deze methode kan niet worden overschreven omdat deze is gedefinieerd als sealed in de klasse Object.

17.2.2.2 De methode ToString()

Deze methode wordt gebruikt om een tekstuele voorstelling van een object terug te geven. Het toont met andere woorden de informatie die we willen weergeven wanneer we het object zelf in

een `Console.WriteLine()` stoppen. Er wordt dan eigenlijk een impliciete aanroep naar de methode `ToString()` gedaan.

```
Console.WriteLine(mijnAuto);           //impliciete aanroep
Console.WriteLine(mijnAuto.ToString());  //expliciete aanroep
```

Beide codevoorbeelden zullen exact dezelfde uitvoer genereren, namelijk:

MyApp.Auto

Dit komt doordat de methode `GetType()` wordt aangeroepen zolang we de methode `ToString()` nog niet hebben **vervangen door een eigen implementatie**.

Het zou natuurlijk fijner zijn dat deze methode nuttigere info teruggeeft, zoals bijvoorbeeld het merk, model en bouwjaar van de auto. Dit doen we met een override in de klasse Auto.

```
class Auto
{
    public string Merk { get; set; }
    public string Model { get; set; }
    public int Bouwjaar { get; set; }

    public override string ToString()
    {
        return $"Deze auto is een {Merk} {Model} ({Bouwjaar}).";
    }
}
```

Wanneer we nu `Console.WriteLine(mijnAuto);` zouden schrijven, krijgen we als mogelijke uitvoer:

Deze auto is een Audi R8 (2024).

17.2.2.3 De methode `Equals()`

Deze methode wordt gebruikt om te bepalen of twee objecten hetzelfde zijn. De standaardimplementatie vergelijkt of de objecten naar hetzelfde geheugenadres verwijzen.

```
if (auto1.Equals(auto2))
```

Het is echter de bedoeling dat we **zelf beslissen wanneer twee objecten van eenzelfde type gelijk zijn** aan elkaar door de `Equals()`-methode te overriden.

Voorbeeld: wij willen dat auto's gelijk zijn aan elkaar ze van hetzelfde Merk en Model zijn.

Slechte manier: met een cast:

```
public override bool Equals(Object o)
{
    Auto temp = (Auto)o;
    return (Merk == temp.Merk && Model == temp.Model);
}
```

We hebben reeds gezien waarom het casten van o naar Auto een **foutgevoelige manier** is. Als de cast mislukt, zal dit namelijk leiden tot een uitzondering. Daarom is het veiliger om eerst te controleren of we wel mogen casten, voor we het effectief doen, met behulp van de operator `is` of `as`.

Goede manier 1: met een `is` operator:

```
public override bool Equals(Object o)
{
    if(o is Auto)
    {
        Auto temp = o as Auto;
        return (Merk == temp.Merk && Model == temp.Model);
    }
    return false;
}
```

Goede manier 2: met een `as` operator:

```
public override bool Equals(Object o)
{
    Auto temp = o as Auto;
    if (temp != null)
    {
        return (Merk == temp.Merk && Model == temp.Model);
    }
    return false;
}
```

17.2.2.4 De methode `GetHashCode()`

De methode `GetHashCode()` wordt gebruikt om een hashcode voor een object te berekenen.

Een hashcode is een numerieke waarde die wordt gebruikt om objecten te identificeren in hash-gebaseerde collecties zoals `Dictionary`, `HashSet` en `HashTable`. Dit zijn collecties die **geen duplicaten** toestaan.

```
public override int GetHashCode()
{
    return HashCode.Combine(Merk, Model); //HashCode is een struct
}
```

i Een struct is zoals een klasse, met het belangrijkste verschil dat een struct wordt opgeslagen als een **waardetype** in plaats van een referentietype. Het wordt meestal gebruikt voor eenvoudige objecten, zoals punten of kleuren, omdat ze ideaal zijn voor kleine, onveranderlijke objecten zonder complexe gedragspatronen.

Structs vallen niet in de scope van dit handboek omdat je ze in de praktijk amper nodig zal hebben. Meestal geniet een `class` namelijk de voorkeur.

De methode `HashCode.Combine()` genereert een enkele hashcode door de eigenschappen Merk en Model te combineren. Dit zorgt ervoor dat twee Auto-objecten die als gelijk worden beschouwd door de `Equals()`-methode, ook dezelfde hashcode hebben. Het vervangen van de methode `Equals()` moet dus altijd samengaan met het vervangen van de methode `GetHashCode()` omdat gelijke objecten altijd dezelfde hashcode moeten hebben.

Als je nu bijvoorbeeld meerdere keren een Audi R8 toevoegt aan een hash-collectie, wordt deze slechts één keer toegevoegd:

```
HashSet<Auto> autos = new HashSet<Auto>();  
  
Auto auto1 = new Auto() { Merk = "Audi", Model = "R8" };  
Auto auto2 = new Auto() { Merk = "Audi", Model = "R8" };  
  
autos.Add(auto1);  
autos.Add(auto2); // Dit wordt niet toegevoegd
```

De variabele `auto2` wordt niet toegevoegd aan de `HashSet` omdat `auto1` en `auto2` gelijk zijn volgens de `Equals()`-methode en bijgevolg dezelfde hashcode hebben.

18.1 Wat is een interface?

18.1.1 Interfaces in de echte wereld

Een interface is een schakel tussen systemen die met elkaar communiceren of samenwerken. We gebruiken interfaces dagelijks zonder het te beseffen, en dit principe zien we terug in de manier waarop we apparaten gebruiken.

Voorbeeld 1: Besturing van voertuigen

Bij het besturen van een auto of vrachtwagen maken bestuurders gebruik van verschillende interfaces om het voertuig te bedienen:

- De contactsleutel dient om de motor te starten;
- Het gaspedaal om sneller te rijden;
- Het stuur om de rijrichting te bepalen.

Hoewel de **interne werking** (of implementatie) van een auto of vrachtwagen **kan verschillen**, beschikken beide over dezelfde bedieningselementen om bestuurd te worden. Hierdoor kan de bestuurder op een gelijkaardige manier met een auto of vrachtwagen rijden, zonder zich te moeten verdiepen in technische details.

Voorbeeld 2: Hardware-interfaces van een computer

Bij computers zijn interfaces van cruciaal belang om componenten en randapparatuur te verbinden:

- USB-poorten, HDMI-poorten, CPU-sockets, ... fungeren als fysieke koppelingen tussen verschillende onderdelen;
- Elk component kan communiceren via een gestandaardiseerde interface, zonder afhankelijk te zijn van de specifieke technische implementatie.

Dankzij deze interfaces kunnen hardware fabrikanten componenten eenvoudig met elkaar compatibel maken, zonder voor elk type aparte communicatiekanalen te ontwikkelen. Interfaces **beloven** dus een gestandaardiseerde communicatie op een afgesproken manier.

18.1.2 Interfaces in OOP

In OOP is een interface een contract dat voorschrijft welke methoden en eigenschappen een klasse moet bevatten, zonder te specificeren hoe deze precies werken. Klassen die een interface implementeren, moeten dus zelf een invulling geven aan de opgelegde leden.

Dit lijkt sterk op het implementeren van abstracte eigenschappen en methoden uit een abstracte superklasse, maar interfaces vereisen geen overerving. Hierdoor kunnen klassen interfaces implementeren naast andere vormen van erfelijkheid, wat flexibiliteit biedt.

Voordelen:

- Interfaces maken **polymorfisme** mogelijk door verschillende klassen met dezelfde interface op een uniforme manier te behandelen. Bijvoorbeeld, klassen die de interface **IPayable**

implementeren kunnen betalingen verwerken, ongeacht hun interne werking. Dit maakt het mogelijk om verschillende betalingsmethoden zonder onderscheid te gebruiken.

- Interfaces bieden **abstractie** door de interne werking van een klasse te verbergen. Gebruikers hoeven alleen te weten dat de interface bepaalde functies biedt, zonder details over de implementatie. Bijvoorbeeld, een interface **IStorable** kan verschillende soorten opslagmedia beschrijven, zonder dat de gebruiker hoeft te weten *hoe* de data wordt opgeslagen.
- Klassen kunnen **meerdere interfaces** implementeren, waardoor het mogelijk is om de voordelen van “multiple inheritance” te benutten. Een klasse kan bijvoorbeeld zowel **IPrintable** als **IDownloadable** implementeren, waardoor deze zowel een afdruk- als downloadfunctionaliteit krijgt.

18.1.3 Interfaces in C#

In C# kun je interfaces declareren zoals een klasse, maar dan met het sleutelwoord **interface**. Bovendien bevatten ze alleen de declaraties van eigenschappen en methoden.

Syntax:

```
interface INaamInterface
{
    returntype Propertynaam { get; set; }
    returntype Methodenaam();
}
```

Voorbeeld:

```
interface IFile
{
    string FileName { get; set; }
    void ReadFile();
    void WriteFile(string text);
}
```



Regels en best practices:

- De naam van een interface begint altijd met een hoofdletter **I**, gevolgd door een beschrijvende naam (bijvoorbeeld **IFile**, **IDownloadable**).
- Interfaces bevatten **geen access modifiers**. Alle leden zijn standaard **public**.
- Interfaces mogen **geen velden of constructors** bevatten, maar wel eigenschappen en methoden.
- Een interface bevat **alleen de declaraties** van eigenschappen en methoden, **zonder de implementatie**. Het specificeert namelijk **wat** een klasse moet doen, maar **niet hoe**. De daadwerkelijke implementatie van de methoden wordt in de klassen zelf neergeschreven.
- Een interface kan **enkel overerven van andere interfaces**, maar niet van klassen. Dit maakt het mogelijk om interfaces te combineren, zonder de complexiteit van meervoudige overerving van klassen.



Tip: Bij het ontwerpen van interfaces, is het belangrijk om de principes van SOLID te volgen. Een van de SOLID-principes stelt dat interfaces klein en specifiek moeten zijn. Dit betekent dat een interface idealiter slechts één taak of verantwoordelijkheid heeft, wat zorgt voor een flexibele en onderhoudbare code.

18.2 Interfaces en klassen

De interface(s) die de klasse belooft te implementeren, plaatsen we bovenaan de klasse na een dubbele punt.

Syntax:

```
public class NaamKlasse : INaamInterface  
{ }
```

Voorbeeld 1: Chat

```
public interface IChat  
{  
    int AantalBerichten { get; }  
    void StuurBericht(string bericht);  
    string OntvangLaatsteBericht();  
}  
  
public class WhatsApp : IChat  
{  
    public int AantalBerichten { get; private set; } = 0; //IChat  
    private List<string> berichtenLijst = new List<string>();  
  
    public void StuurBericht(string bericht) //IChat  
    {  
        berichtenLijst.Add(bericht);  
        AantalBerichten++;  
        Console.WriteLine($"Bericht verstuurd: {bericht}");  
    }  
  
    public string OntvangLaatsteBericht() //IChat  
    {  
        if (AantalBerichten > 0)  
        {  
            return berichtenLijst[AantalBerichten - 1];  
        }  
    }  
}
```

Door aan de klasse WhatsApp de interface IChat toe te kennen, geven we aan programmeurs de zekerheid dat deze klasse gegarandeerd de methoden StuurBericht(), OntvangLaatsteBericht(), en de eigenschap AantalBerichten bevat. Zo weten we dat de implementerende klassen over een chatfunctie beschikken.

Voorbeeld 2: Documenteigenschappen

```
public interface IEigenschappen
{
    int AantalWoorden { get; }
    int AantalPaginas { get; }
    void ToonEigenschappen();
}

public class WordDocument : IEigenschappen
{
    private string tekst;
    public string Titel { get; set; }

    public int AantalWoorden          //IEigenschappen
    {
        get { return tekst.Split(new[] { ' ', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries).Length; }
    }

    public int AantalPaginas         //IEigenschappen
    {
        get { return (AantalWoorden / 300) + 1; }
    }

    public void VoegTekstToe(string tekst)
    {
        this.tekst += tekst;
    }

    public void ToonEigenschappen() //IEigenschappen
    {
        Console.WriteLine($"Titel: {Titel}");
        Console.WriteLine($"Aantal woorden: {AantalWoorden}");
        Console.WriteLine($"Aantal pagina's: {AantalPaginas}");
    }
}
```

Een klasse die de interface IEigenschappen implementeert, moet de eigenschappen AantalWoorden en AantalPaginas, en de methode ToonEigenschappen() bevatten en definiëren. Zo garandeert de interface consistentie tussen verschillende documenttypen zoals een Word- of PDF-document.

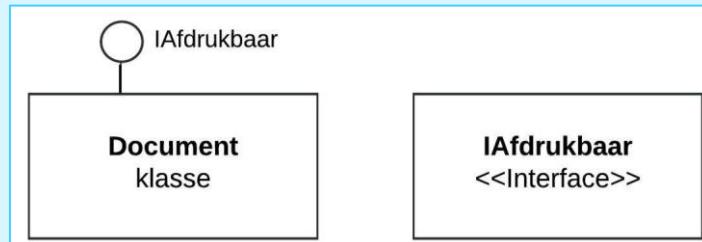


Als een klasse de verplichte leden van een interface niet implementeert, kan de code niet worden gecompileerd. Bijgevolg zal het programma niet werken.

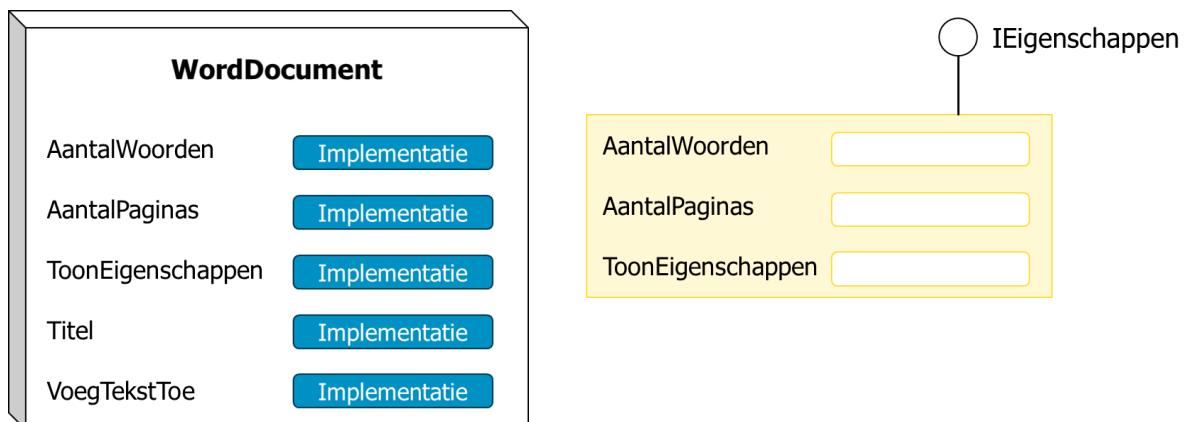
18.2.1 Visuele voorstelling

(i) In UML wordt een klasse die een interface implementeert aangeduid met een streepje en een cirkel bovenaan. Naast de cirkel wordt de naam van de interface geschreven.

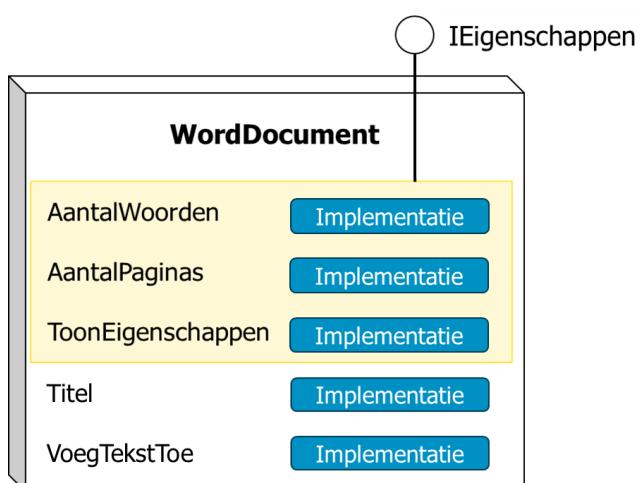
Voorbeeld: IAfdrukbaar wordt geïmplementeerd door de klasse Document.



Een handige manier om interfaces te visualiseren, is om deze voor te stellen als een transparant sjabloon dat je bovenop een klasse legt. Op dit sjabloon staan de vereiste methoden en eigenschappen van de interface weergegeven.



Wanneer je dit sjabloon precies over een klasse legt die deze interface implementeert, sluiten de gaten van het sjabloon mooi aan op de methoden en eigenschappen van de klasse.



18.3 Interfaces en overerving

Zoals je weet kan een klasse in C# slechts van één superklasse overerven. Dit helpt om een klasse-hiërarchie eenvoudiger en minder complex te houden.

Een klasse kan echter wel **meerdere interfaces implementeren**. Daardoor kan het verschillende eigenschappen en gedragingen van verschillende bronnen overnemen. De enigste voorwaarde is dat de superklasse altijd eerst wordt overgeërfd, gevolgd door de implementatie van de interface(s).

Een voorbeeld hiervan is een Word-document dat de klasse Bestand als parent-klasse heeft, en daarnaast meerdere interfaces implementeert die verschillende soorten documentopmaak ondersteunen.

```
class WordDocument : Bestand, ITekenopmaak, ITabel, IPaginaIndeling
```

Interfaces kunnen van elkaar overerven, waardoor programmeurs interfaces geleidelijk aan kunnen uitbreiden. Dit betekent dat een interface aanvullende methoden en eigenschappen kan erven van een basisinterface.

Zo kunnen bijvoorbeeld interfaces voor verschillende soorten media worden uitgebreid met extra functionaliteiten die specifiek zijn voor dat type media.

```
interface IMedia
{
    void Start();
    void Stop();
}

interface IAudio : IMedia
{
    int Volume { get; set; }
    void VolumeDempen (bool volumeStaatAan);
}

interface IVideo : IAudio
{
    int Resolutie { get; set; }
    void ToonVideo();
}
```

18.3.1 Abstracte klasse of interface?

Bij het kiezen tussen een abstracte klasse en een interface, is het belangrijk om na te denken over de aard van de relatie die je wilt definiëren, alsook welke eigenschappen en methoden nodig zijn.

Abstracte klasse:

- Kan leden als instantievariabelen, eigenschappen, methoden en constructors bevatten.
- Bevat methoden die, indien gewenst, al volledig geïmplementeerd kunnen zijn.
- Klassen kunnen slechts van één superklasse overerven, waarbij sprake is van een sterke "Is een"-relatie.
- Voordeel: gedeelde methoden moeten niet voor elke subklasse opnieuw geschreven worden. Het volstaat om een methode slechts éénmaal uit te schrijven.
- Een subklasse breidt een superklasse verder uit.

Interface:

- Bevat alleen methoden en/of eigenschappen, zonder implementatie.
- Fungeert als een lijst van vereiste methoden en eigenschappen voor implementerende klassen.
- Staat een klasse toe om meerdere interfaces te implementeren. Dit biedt een vorm van multiple inheritance waarmee klassen functionaliteiten uit meerdere interfaces kunnen combineren.
- Geschikt voor klassen met meerdere "rollen" of verantwoordelijkheden, dankzij de mogelijkheid om verschillende interfaces te combineren.
- Handig wanneer verschillende klassen dezelfde methoden en eigenschappen moeten delen, zonder een duidelijke "Is-een"-relatie tussen deze klassen.

18.4 Interfaces en polymorfisme

18.4.1 De interface als een datatype

Door een interface te definiëren, maakt men in feite een **nieuw datatype (referentietype)** dat objecten kan beschrijven. Net als bij superklassen kunnen objecten dus behandeld worden als exemplaren van een interface, mits ze deze implementeren. Dit betekent dat een object dat aan een interface wordt toegewezen, **alleen toegang heeft tot de methoden en eigenschappen die in die interface beschreven staan**.

We illustreren dit aan de hand van een voorbeeld met verschillende soorten magiërs die de interface `IMagicUser` implementeren.

```
public interface IMagicUser
{
    int Mana { get; set; }
    void CastSpell();
}

public class Necromancer : IMagicUser { ... }

public class Mage : IMagicUser { ... }

internal class Program
{
    static void Main()
    {
        IMagicUser necromancer = new Necromancer();
        IMagicUser mage = new Mage();

        necromancer.CastSpell();
        mage.CastSpell();

        int totalRemainingMana = necromancer.Mana + mage.Mana;
    }
}
```

18.4.2 Praktijkvoorbeeld

We nemen het voorbeeld van de garage en het onderhoudssysteem uit het vorige hoofdstuk er terug bij.

Een groot nadeel van de gekozen aanpak is dat alleen objecten van de superklasse voertuig op deze manier een onderhoudsbeurt kunnen krijgen. Het zou echter handiger zijn als ook andere objecten, zoals machines en computers, via hetzelfde systeem onderhouden kunnen worden. Overerving gebruiken is echter geen optie aangezien machines en computers geen voertuigen zijn. We moeten "onderhoud" dus beschouwen als een aparte taak, en niet als de hoofdreden voor het bestaan van een klasse.

Dit probleem kunnen we oplossen door de volgende stappen uit te voeren:

Ten eerste: Maak de interface IOnderhoud, die de methode onderhoud() bevat.

```
public interface IOnderhoud
{
    void Onderhoud();
}
```

Ten tweede: Elke klasse die deze interface implementeert, kan nu onderhouden worden, terwijl de klasse zelf ook andere leden kan bevatten.

```
public class Computer : IOnderhoud
{
    public void Onderhoud()
    {
        Stofverwijdering();
        BesturingssysteemUpdaten();
        OpslagmediaOpschonen();
    }

    public void Stofverwijdering()
    {
        Console.WriteLine("De interne hardwarecomponenten worden
        gereinigd.");
    }
    public void BesturingssysteemUpdaten()
    {
        Console.WriteLine("het besturingssysteem wordt geüpdatet.");
    }
    public void OpslagmediaOpschonen()
    {
        Console.WriteLine("Tijdelijke en onnodige bestanden worden
        verwijderd.");
    }

    //gevolgd door de reeds bestaande leden van de klasse Computer
}
```

Ten derde: Uiteraard moeten ook alle voertuigklassen de interface IOnderhoud implementeren.

Aangezien de methode Onderhoud() momenteel de enigste bestaansreden is voor de superklasse Voertuig, is het beter om deze klasse te verwijderen. De voertuigen erven bijgevolg niet langer over van een superklasse.

```
public class Auto : IOnderhoud
{
    public void Onderhoud()
    {
        VerversOlie();
        ControleerBandenspanning();
        ReinigInterieur();
    }

    public void VerversOlie() { ... }
    public void ControleerBandenspanning() { ... }
    public void ReinigInterieur() { ... }
}

public class Motorfiets : IOnderhoud
{
    public void Onderhoud()
    {
        SmeerKetting();
        ControleerRemmen();
        ControleerBanden();
    }

    public void SmeerKetting() { ... }
    public void ControleerRemmen() { ... }
    public void ControleerBanden() { ... }
}

public class Vrachtwagen : IOnderhoud
{
    public void Onderhoud()
    {
        LaadtPallets();
        VerversOlie();
        ControleerLading();
    }

    public void LaadtPallets() { ... }
    public void VerversOlie() { ... }
    public void ControleerLading() { ... }
}
```

Ten vierde: De klasse Garage moet aangepast worden zodat deze een lijst van het type IOnderhoud bevat, in plaats van een lijst met Voertuigen.

```
public class Garage
{
    public List<IOnderhoud> VasteActiva = new List<IOnderhoud>()
    {
        new Auto(),
        new Motorfiets(),
        new Vrachtwagen(),
        new Computer()
    };

    public void Onderhoud()
    {
        foreach (var item in VasteActiva)
        {
            item.Onderhoud();
        }
    }
}
```

18.4.3 Het sleutelwoord `is`

Soms willen we specifieke code alleen uitvoeren wanneer een object een bepaalde interface implementeert. Dit kunnen we nagaan met het sleutelwoord `is`.

In het volgende voorbeeld voeren we alleen code uit wanneer een object de interface IAfdrukbaar implementeert. De bijhorende leden en hun implementatie zijn hier niet van belang.

```
public interface IAfdrukbaar { ... }

public class Document : IAfdrukbaar { ... }

public class Rekenblad { ... }

public class Program
{
    static void Main()
    {
        Document cv = new Document();
        Rekenblad verkoopcijfers = new Rekenblad();

        if(cv is IAfdrukbaar)
            Console.WriteLine("Uw cv wordt afgedrukt");
        if(verkoopcijfers is IAfdrukbaar)
            Console.WriteLine("De verkoopcijfers worden afgedrukt");
    }
}
```

Dit zal als uitvoer de volgende boodschap geven:

```
Uw cv wordt afgedrukt
```

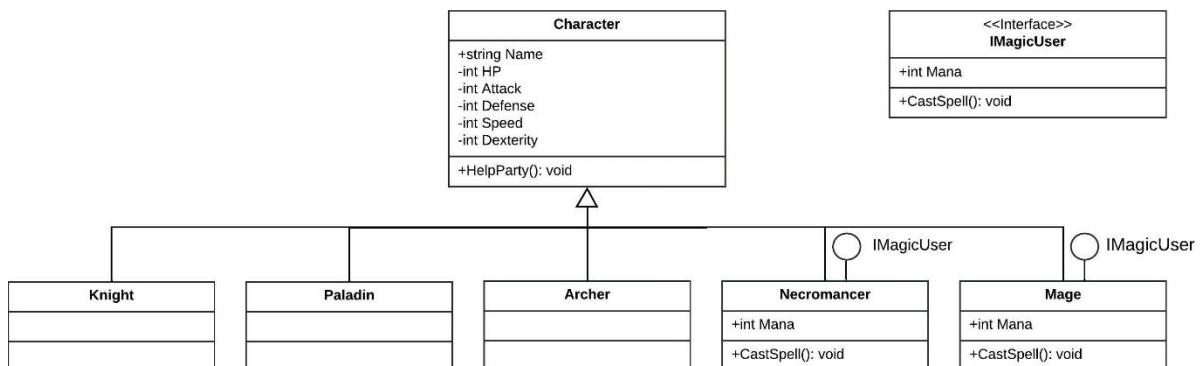
18.4.4 Eigen interfaces

Het combineren van de eigenschappen van polymorfisme en interfaces kan leiden tot bijzonder krachtige code. Wanneer we vervolgens de `is` en `as` sleutelwoorden gebruiken, komt de kracht van OOP pas echt tot zijn recht.

Dit principe kan geïllustreerd worden aan de hand van een voorbeeld uit een fictieve videogame, waarin een groep (of party) bestaat uit vijf leden. Elk lid beschikt over een specialiteit, ook wel een "class" of "job" genoemd.

In dit scenario heeft ieder lid van de party een unieke rol, waarbij sommige personages *magie* kunnen gebruiken:

- Een Necromancer roept een skelet op bij het aanroepen van de functie `CastSpell()`.
- De Mage kan alle "stats" van de groepsleden met +2 verhogen (of buffen) door `CastSpell()` aan te roepen.
- *Alle* jobs kunnen andere leden van de groep beschermen tegen aanvallen door de `HelpParty()` functie aan te roepen.



```
public interface IMagicUser
{
    int Mana { get; set; }
    void CastSpell();
}

public abstract class Character
{
    public string Name { get; set; }
    public int HP { get; private set; }
    public int Attack { get; private set; }
    public int Defense { get; private set; }
    public int Speed { get; private set; }
    public int Dexterity { get; private set; }

    public void HelpParty()
    {
        Console.WriteLine("This character protects its party members.");
    }
}
```

```

public class Knight : Character { }
public class Paladin : Character { }
public class Archer : Character { }

public class Necromancer : Character, IMagicUser
{
    public int Mana { get; set; } = 150;

    public void CastSpell()
    {
        Console.WriteLine("The necromancer conjures a skeleton warrior.");
        Mana -= 25;
    }
}

public class Mage : Character, IMagicUser
{
    public int Mana { get; set; } = 100;

    public void CastSpell()
    {
        Console.WriteLine("The mage casts a spell that increases all stats
by +2.");
        Mana -= 50;
    }
}

```

Vervolgens maken we in de methode Main() een groep in de vorm van een array aan (dit kon even goed in een List<Character>, maar dat maakt op zich niet uit).

```

Character[] partyMembers = new Character[5];
partyMembers[0] = new Mage();           //IMagicUser
partyMembers[1] = new Paladin();
partyMembers[2] = new Archer();
partyMembers[3] = new Necromancer();   //IMagicUser
partyMembers[4] = new Knight();

```

Het **probleem** is nu als volgt: hoe zorg je ervoor dat in een array van verschillende datatypes alleen de jobs die magie gebruiken een spreek uitspreken?

Oplossing 1: met sleutelwoord is:

We doorlopen de array en controleren voor elk object of het de IMagicUser interface implementeert, met behulp van het sleutelwoord **is**.

```

for (int i = 0; i < partyMembers.Length; i++)
{
    if (partyMembers[i] is IMagicUser)
    {
        IMagicUser temp = partyMembers[i] as IMagicUser;
        temp.CastSpell();
    }
    else
    {
        partyMembers[i].HelpParty();
    }
}

```

Oplossing 2: met sleutelwoord as:

Hier proberen we het object te converteren naar een tijdelijk object van het type `IMagicUser`, met behulp van het sleutelwoord `as`. Vervolgens gaan we na of de conversie geslaagd is door te controleren of het tijdelijke object niet null is. Tenslotte kunnen we het tijdelijke object een spreuk laten uitspreken, of de groepsleden laten helpen wanneer de conversie mislukt is.

```

for (int i = 0; i < partyMembers.Length; i++)
{
    IMagicUser temp = partyMembers[i] as IMagicUser;
    if (temp != null)
    {
        temp.CastSpell();
    }
    else
    {
        partyMembers[i].HelpParty();
    }
}

```

Beide oplossingen zullen de volgende uitvoer geven:

```

The mage casts a spell that increases all stats by +2.
This character protects its party members.
This character protects its party members.
The necromancer conjures a skeleton warrior.
This character protects its party members.

```

Door polymorfisme te combineren met interfaces kunnen we de verschillende acties van deze personages op een uniforme manier aanroepen. Dit maakt het mogelijk om de logica van de game te vereenvoudigen en flexibeler te maken.

18.5 Overzicht populairste .NET-interfaces

Het .NET-framework bevat een breed scala aan interfaces die veel gebruikt worden om specifieke functionaliteiten mogelijk te maken. Zelfgemaakte klassen kunnen deze interfaces implementeren om compatibel te zijn met bestaande methoden en functionaliteiten binnen .NET.

Enkele van de meest gebruikte .NET-interfaces zijn:

.NET-interface	Beschrijving
IEnumerable & IEnumerator	Voor het doorlopen van collecties met foreach en LINQ.
IDisposable	Maakt opruimen van bronnen mogelijk, zoals bestanden of netwerkverbindingen.
IQueryable	Uitvoeren van query's tegen queryable databronnen, zoals databases.
INotifyPropertyChanged	Voor databinding, updatet UI bij wijziging van eigenschapwaarden.
IComparable & IComparer	Vergelijken en sorteren van objecten in collecties.
IEquatable & IEqualityComparer	Vergelijkt objecten voor gelijkheid, vooral nuttig in zoekoperaties.
IList & ICollection	Beheert verzamelingen, biedt mogelijkheden om items toe te voegen of te verwijderen.
IEnumerable & IEnumerator	Voor het doorlopen van collecties met foreach en LINQ.

De kans is groot dat je deze interfaces vroeg of laat zal moeten implementeren in een project. Voor meer informatie over deze informatie, raden we je aan om de [officiële documentatie](#) van de specifieke .NET-interface te lezen.



Je hebt nu alle verplicht te kennen leerstof van de richting Applicatie- en databaseheer over programmeren in C# (hopelijk) onder de knie. Proficiat met het behalen van deze mijlpaal!

Met deze kennis en vaardigheden beschik je alvast over een stevige basis om gemakkelijk nieuwe programmeerconcepten op te pikken. In de wereld van programmeren valt er namelijk nog veel meer te ontdekken. In de bijgevoegde lijst geef ik je alvast enkele richtingen die de moeite waard zijn om te verkennen ([bron: Zie Scherp Scherper](#)):

- **Geavanceerde C# concepten:** Verdiep je verder in de taal zelf, zoals werken met async, events, en LINQ. LINQ, een krachtige tool voor gegevensbeheer, komt in vrijwel alle .NET-toepassingen van pas.
- **Desktop-applicaties:** Breid je kennis uit door desktop-apps te maken met frameworks zoals WPF of UWP. Dit vereist enige kennis van event-gebaseerd programmeren en XAML, maar je huidige basis helpt je snel op weg.
- **Mobiele applicaties:** Hoewel native Android- en iOS-apps niet in C# worden geschreven, biedt .NET MAUI een uitstekende optie voor cross-platform ontwikkeling. Met één codebase maak je apps voor Windows, Android, iPhone en meer.
- **Webontwikkeling:** Ontdek de .NET-backendstack met ASP.NET en Entity Framework. Voor wie liever geen JavaScript gebruikt, biedt Blazor een unieke kans om met C# interactieve web-apps te bouwen.
- **Game development:** Voor game-ontwikkeling kun je aan de slag met Unity, een populaire op C# gebaseerde game-engine, of Monogame, waar bekende indie-games zoals Stardew Valley mee zijn ontwikkeld. Een andere optie is Godot, een toegankelijke engine voor beginners.
- **Azure en de cloud:** Duik in Azure, Microsofts cloudplatform. Hier ontdek je een breed scala aan diensten en technologieën waarin .NET en C# centraal staan.
- **Gevorderde programmeerconcepten:** Verken taal-agnostische concepten zoals Design Patterns, Dependency Injection, en SOLID-principes. Deze helpen je efficiëntere en flexibele code te schrijven, ongeacht de programmeertaal.

Ik raad je aan om nieuwsgierig te blijven en je kennis voortdurend te verdiepen. Het mooie van programmeren is dat het een continu leerproces is: hoe meer je onderzoekt, experimenteert en bouwt, hoe sterker je vaardigheden worden. Ga op ontdekkingstocht, duik in nieuwe uitdagingen en blijf groeien als programmeur.

Veel succes met de rest van jouw programmeeravontuur!



- Cursus C Sharp.* (s.d.). Opgehaald van C-SHARP.BE: <https://www.c-sharp.be/c-sharp/>
- Dams, T. (2022). *Zie Scherp Scherper*. Brave New Books. Opgehaald van <https://apwt.gitbook.io/zie-scherp-scherper/>
- Hamedani, M. (s.d.). *The Ultimate C# Mastery Series*. Opgehaald van Code With Mosh: <https://codewithmosh.com/p/the-ultimate-csharp-mastery-series>
- Microsoft. (s.d.). *C# documentation*. Opgehaald van Learn Microsoft: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>
- TutorialsTeacher. (s.d.). *Learn C# Programming*. Opgehaald van TutorialsTeacher: <https://www.tutorialsteacher.com/csharp>
- Vaes, N. (2014). *OO Programmeren in Java*. Hasselt.
- Van Deuren, D. (2015). *Computersystemen, algoritmisch denken*. Antwerpen: Karel de Grote-Hogeschool.
- W3 Schools. (s.d.). *C# Tutorial*. Opgehaald van W3 Schools: <https://www.w3schools.com/cs/index.php>