C# Klassikaal Basis / Refresher

Inhoud

[Document revisies 4](#_Toc152748163)

[IDE (Visual Studio) 5](#_Toc152748164)

[Aanmaken projecten 5](#_Toc152748165)

[Variables & Datatype 9](#_Toc152748166)

[Primaire Datatypes 9](#_Toc152748167)

[String & Char 10](#_Toc152748168)

[Int 10](#_Toc152748169)

[Long & Short 10](#_Toc152748170)

[Float, Double & Decimal 12](#_Toc152748171)

[Types 12](#_Toc152748172)

[Variables, fields en Properties 14](#_Toc152748173)

[Variable 14](#_Toc152748174)

[Fields 14](#_Toc152748175)

[Properties 15](#_Toc152748176)

[Parse 17](#_Toc152748177)

[If’s en Loops 18](#_Toc152748178)

[If, else if, else 18](#_Toc152748179)

[Logical operators 19](#_Toc152748180)

[Conditional operators 20](#_Toc152748181)

[For, ForEach en While 21](#_Toc152748182)

[While 21](#_Toc152748183)

[For 21](#_Toc152748184)

[ForEach 22](#_Toc152748185)

[Control statements 23](#_Toc152748186)

[Continue 23](#_Toc152748187)

[Break 23](#_Toc152748188)

[Collections 25](#_Toc152748189)

[Array 25](#_Toc152748190)

[Items opvragen, toevoegen, toewijzen en verwijderen 25](#_Toc152748191)

[Items toewijzen 26](#_Toc152748192)

[Items verwijderen 26](#_Toc152748193)

[List<T> 26](#_Toc152748194)

[Items toevoegen 26](#_Toc152748195)

[Items verwijderen 27](#_Toc152748196)

[Items opvragen uit een lijst 27](#_Toc152748197)

[Dictionary<T,U> 27](#_Toc152748198)

[Items toevoegen 28](#_Toc152748199)

[Items verwijderen 28](#_Toc152748200)

[Items opvragen uit de dictionary 28](#_Toc152748201)

[Files en IO 29](#_Toc152748202)

[Namespace 29](#_Toc152748203)

[Simpel lezen van een bestand 29](#_Toc152748204)

[Simpel schrijven van een bestand 30](#_Toc152748205)

[Complex types 30](#_Toc152748206)

[LINQ 32](#_Toc152748207)

[Where 32](#_Toc152748208)

[Select 33](#_Toc152748209)

[Max 33](#_Toc152748210)

[Programming architectuur 34](#_Toc152748211)

[Classes 34](#_Toc152748212)

[Constructors 34](#_Toc152748213)

[Methods 36](#_Toc152748214)

[Afsluiting 38](#_Toc152748215)

# Document revisies

|  |  |
| --- | --- |
| datum | omschrijving |
| 4-9-2023 | Initieel document |
| 11-9-2023 | Toevoeging Collections hoofdstuk. Toevoegen LINQ |
| 18-9-2023 | Toevoeging Revisies |
| 26-9-2023 | Toevoeging contactgegevens einde document.  Uitbreiding Hoofdstuk “IDE (Visual Studio)”.  Toevoeging ‘Byte’ in Primaire datatypes. |
| 4-12-2023 | Toevoegen van IO hoofdstuk. |
| 6-12-2023 | Toevoegen van Constructor hoofdstuk. |
|  |  |

# IDE (Visual Studio)

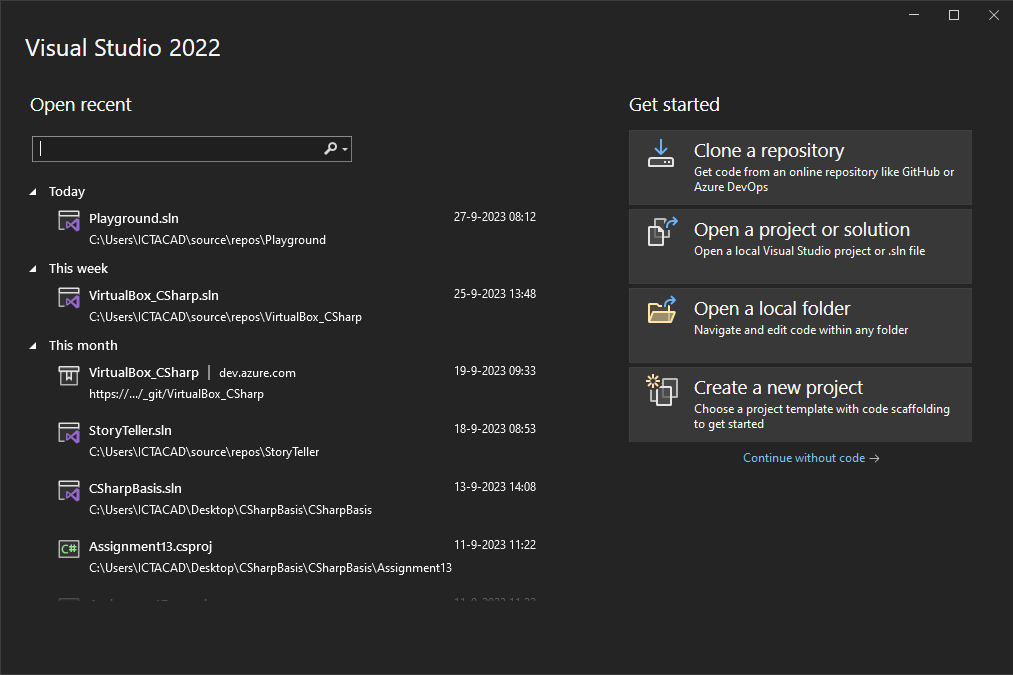
Een IDE is een programma waar je andere programma’s in schrijft. Dat klinkt als een robot die robots bouwt, en dat is ergens ook wel zo. Je kan in een bestaande IDE ook je eigen IDE bouwen (maar da’s super veel werk…).

Microsoft’s Visual Studio is het flagschip voor meeste bedrijven die met .NET programmeren. Met Visual Studio kan je programmeren in talen zoals Visual Basic, C++ en C#. Maar tegenwoordig worden ook talen zoals Python ondersteund en daarnaast is de IDE ook uit te breiden met plugins en extensies voor meer functies en talen.

Er zijn ook andere IDE’s die .NET ondersteunen, zoals Rider. Maar het nadeel is dat deze IDE’s vaak achterlopen op de nieuwe ontwikkeling van Microsoft’s producten.

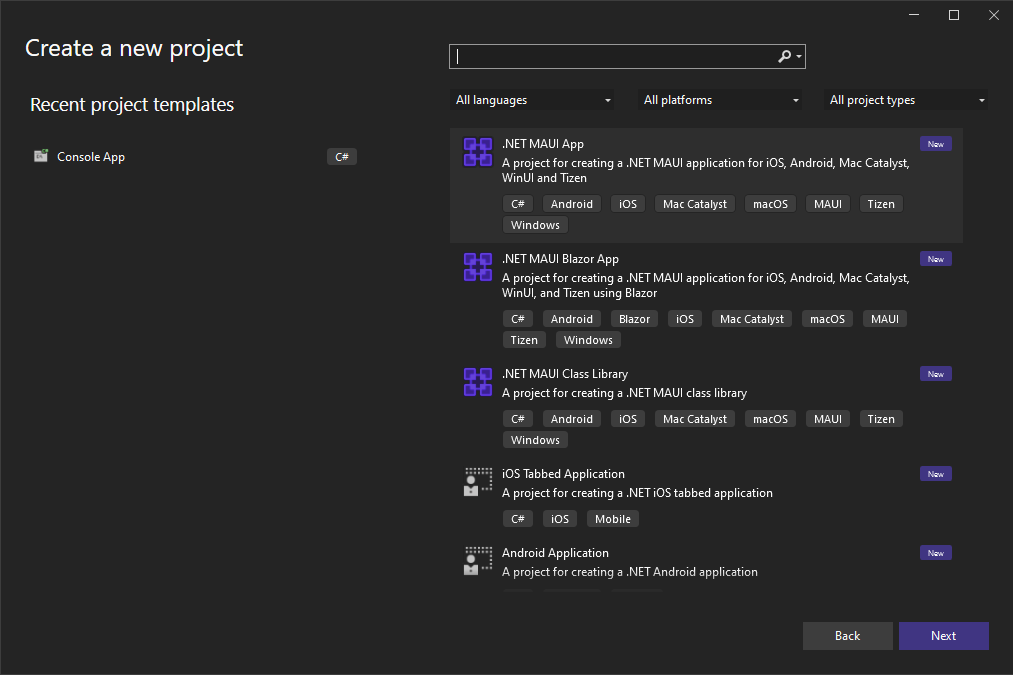
## Aanmaken projecten

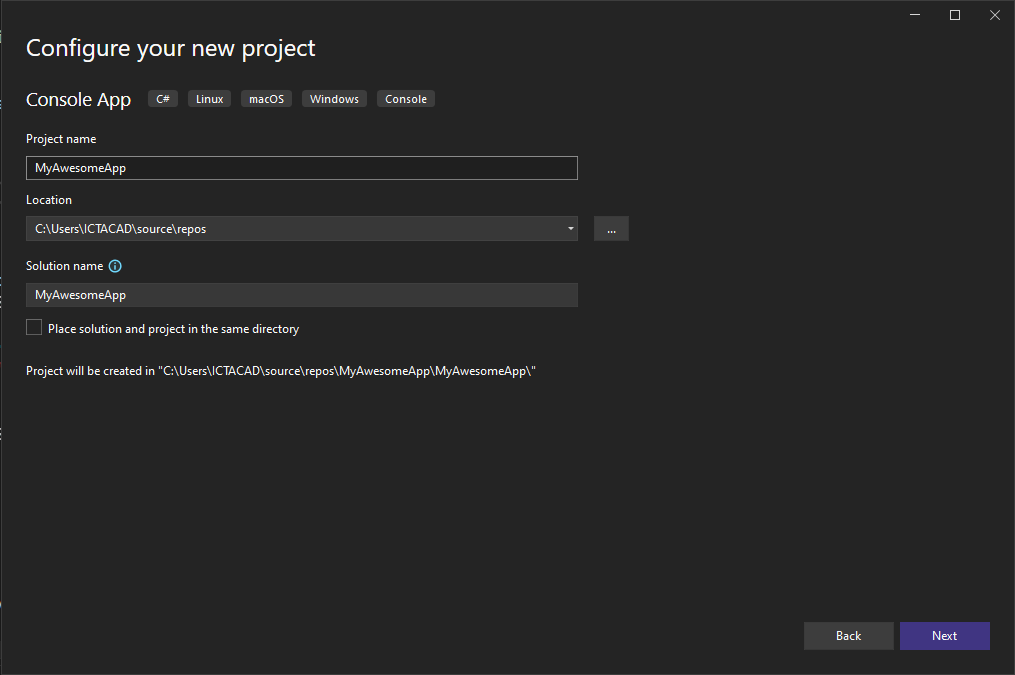
.NET en alle .NET talen werken in een project structuur. Dat houd in dat er een bestandje is die bijhoud welke bestanden onderdeel zijn van jouw project. In dit document gaan we niet diep in op de project structuur.

Als je Visual Studio start, wordt je begroet door de Launcher. Hier staan een aantal opties en je recente projecten.

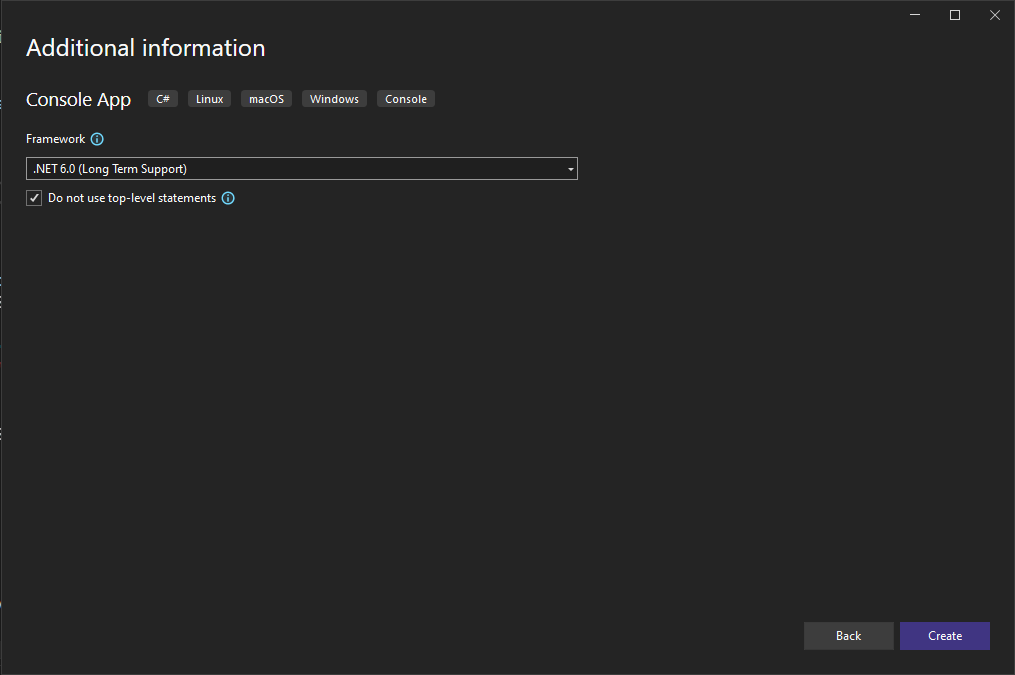
Als je een nieuw project wilt starten, druk je op “Create a new project”. Je kan vanaf dit scherm ook meteen bestaande projecten openen of met GIT projecten klonen van het internet.

Eerst moet je bedenken wat voor project je wilt starten: een console project of WinForms (native Windows), of misschien iets spannender zoals MAUI (cross platform) of Razor (web). Zoek of gebruik de filters om het juiste project type te vinden. Tijdens je opleiding zal je voornamelijk werken met Console en WinForms, tenzij anders aangegeven.

Dit voorbeeld gaat ervan uit dat er is gekozen voor een Console project.

Vervolgens moet je informatie opgeven over je project, zoals naam en opslag locatie.

Soms zijn er extra stappen nodig voordat je kan beginnen. In dit geval moeten we opgeven met welk versie van het framework gaan werken. Ga er altijd vanuit dat je werk met de huidige LTS versie van .NET framework.

Een .NET versie bepaald welke functies en ontwikkelingen er tot je beschikking zijn. Het is achteraf ook mogelijk om de versie aan te passen.

# Variables & Datatype

Variables en datatypes liggen dicht bij elkaar. Een datatype is een representatie van data en/of informatie. Maar je hebt hier 2 soorten van.

* (Primaire) Data-types
* Complex Types

## Primaire Datatypes

Een datatype, ook wel bekend als een primair datatype / primary types, zijn de laagste niveaus van Types in het programmeren. Een aantal voorbeeld van een datatype zijn: string, int, float, decimal, ect. Hieronder staat een tabel waar je de veelgebruikte primaire data types kan terugvinden met simpele voorbeelden.

|  |  |  |
| --- | --- | --- |
| Type | Lengte / Range | Voorbeeld |
| String | 2.000.000.000.000 +/- | “Hello World! Pokeb@ll go?” |
| Char | 1 character of teken | ‘A’, ‘?’, ‘%’, ‘b’, ‘5’ |
| Int | -2.147.483.648 tot 2.147.483.647 | 12345 |
| Short | -32.768 tot 32.767 | 12345 |
| Long | -9.223.372.036.854.775.808 tot 9.223.372.036.854.775.807 | 1234567890 |
| Float | -heel erg veel tot +heel erg veel  Precies tot 9 cijfers achter de komma | -143,123  2847,43 |
| Decimal | -heel erg veel tot +heel erg veel  Precies tot 28 cijfers achter de komma | -143,1238384654641  2847,43684354651 |
| Double | -heel erg veel tot +heel erg veel  Precies tot 17 cijfers achter de komma | -873835.1813854646848  83548.85468464 |
| Boolean | True | False | 1 == 1 (true)  True  False  1 != 1 (false) |
| Byte | 0 tot 255 | De waarde van R in een RGB scherm of plaatje. Een toewijzing van CPU cores op een applicatie. |

### String & Char

Een string is een datatype dat tekst representeert. Je kan alle letters, nummers, leestekens en zelfs emoji's. De waarde van een string geef je aan met dubbele aanhalingstekens aan beide kanten. Je kan geen rekenacties uitvoeren op een string.

Als je gaat proberen te rekenen met strings, komt het verschil of additief resultaat terug, bijvoorbeeld:

“Hello” + “World” = “Hello World”

“Hello World” – “World” = “Hello ”

Een Char aan de andere kant is een specifiek (unicode) character, dit kan een letter, cijfer, leesteken, speciaal teken of een emoticon zijn. Ook op een char kan je geen rekenoperaties uitvoeren. Een char geeft je aan door middel van één aanhalingsteken aan beide kanten. Het is niet mogelijk om meerdere characters in 1 char variable te zetten.

‘a’

‘C’

‘🥲’

“Hello World! I am a textual representation of data in a string/String format!”

### Int

Een Int, ook wel bekend als een Integer, is een heel getal. Een int kan positief of negatief zijn, maar kan nooit decimalen bevatten. Met een Int kan je wel rekenoperaties uitvoeren. Dit datatype is ook wel bekend als Int32.

24 = 12 + 12

24 = 12 \* 2

245845  
-457845

### Long & Short

De *long* en *short* datatypes zijn allebei een variant op de int maar verschillen in het feit dat ze een meer of minder geheugen verbruiken.  
**Short** is ook wel bekend als een Int16, bit-wijs half zo klein als een standaard integer. Je kan maximaal -32000 / + 32000 in een **short** kwijt.

**Long**, ook wel bekend als een Int64, is bit-wijs een keer zo groot als een standaard integer. In een long kan je een getal kwijt wat klinkt als 9.223.372.036.854.775.807 (da’s heel veel). En dit heb je ook in een negatieve form, -9.223.372.036.854.775.808 (da’s heel erg veel nativiteit).

Het is aan de ontwikkelaar om te bepalen welk datatype het beste is en een situatie, vooral als er maar een beperkte hoeveelheid werkgeheugen beschikbaar is (en of de CPU wel beschikt over een 64-bit instructie set).

### Float, Double & Decimal

Nu krijgen we te maken met getallen waar decimalen in voor kunnen komen, wellicht heb je deze al een keertje meegemaakt. Deze 3 datatypes kunnen allemaal decimale getallen bevatten. Het verschil ligt in de precisie in de decimalen. De float, double en decimal slaan getallen niet echt op als echte getallen, zoals de int dat wel doet, maar slaat deze op in een binaire representatie. Problemen doen zich voor wanneer decimalen zo precies worden, dat de binair een patroon gaan vormen die voor eeuwig doorgaat.

**Float:** Ook wel bekend als een *Single*, een float is de kleinste variant om decimale getallen in op te slaan en heeft een precisie van 6 tot 9 getallen achter de komma.

**Double:** Dit is de veel voorkomende variant en heeft een precisie van ongeveer 15 tot 17 getallen achter de komma.

**Decimal:** Dit type is de grootste maat om decimalen in op te slaan en heeft de hoogste precisie van 28 tot 19 getallen achter de komma.

#### Precisie

Je kan natuurlijk meer getallen getallen achter de komma opslaan dan 6 als je een float gebruikt. Het verschil zit namelijk in de getallen die buiten de precisie vallen. In geval van de float, kan het 7e getal steeds weer anders zijn.

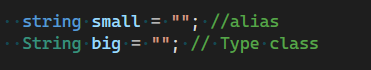
De double en de decimal hebben dit ook, en dit is niet gelimiteerd tot C#. Alle talen die een decimal getal ondersteunen hebben dit gedrag. Dit geldt ook voor databases, omdat die decimalen getallen op dezelfde manier opslaat.

## Types

Zojuist hebben wij het gehad over primaire datatypes. De basis onderdelen binnen programmeren die data representeren. Maar hier zijn 2 soorten van. Namelijk Primaire datatypes en Complexe datatypes. Primaire datatypes hebben we net behandeld, de string, int, bool, ect.  
Complexe datatypes zijn alle andere types. Dit zijn classes, structs, definitions en nog vele andere zaken.

Deze primaire datatypes bestaan al in de source code van het OS waar je mee werkt, deze zijn op een paar specifieke uitzondering na, allemaal op de zelfde wijze geïmplementeerd. Als je .NET gebruikt is er ook nog een extra implementatie laag zodat onafhankelijk waar je op werkt, je de primaire datatypes kunt gebruiken met altijd hetzelfde gedrag. Bijna alle programmeertalen hebben een eigen implementatie voor primaire datatypes.

Het verschil tussen deze twee lagen kan je terug vinden in de Typed classes en de short-hand alias. De Typed-class is geschreven met een hoofdletter en (in Visual Studio) geelachtig gekleurd. Een alias daarin tegen is geschreven met een kleine letter en is vaak blauw gekleurd.



Types daarin tegen zijn wat breder en we gaan er in dit document niet al te diep op in, dat gaan we doen in de C# Advanced reader. Maar voor nu mag je het volgende onthouden:

Alle primaire datatypes == Types

Alle primaire datatypes zijn types. Maar niet alle types zijn primaire datatypes.  
Onderstaand zie je de Types van de bovenstaande datatypes. De Console class is ook een soort Type.

Tegenwoordig is er geen verschil tussen bijvoorbeeld ‘string’ en ‘String’. Op technische wijze zou het niet uitmaken welk van de twee je gebruikt. Maar de algemene regel is dat we de primaire datatypes gebruiken.

Hieronder een voorbeeld van een variable met een primair datatype en een complex type. De class `MySuperAwesomeClass` is een zelf gemaakte class met zelf geschreven properties. Classes en properties gaan we verderop in het document behandelen.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

*Bijna alle elementen die je terug vind in het programmeren zoals Classes, Structs, Enums, ect zijn ook Types.*

# Variables, fields en Properties

Nu gaan we wat meer aan het programmeren, het vorige hoofdstuk was vooral theorie. We starten met iets meer theorie, en wellicht hebben jullie het al wel eerder gehoord: Scopes.

Scopes zijn eigenlijk heel simpel, en je kan ze herkennen aan de accolades: ‘{’ en ‘}’. Alles wat tussen een setje accolades staat is een scope. Je kan min of meer zo veel mogelijk scopes maken als je zelf wilt.

## Variable

De variable maakt het mogelijk om informatie van een bepaald type voor een korte tijd op te slaan. Je kan alleen variables maken in methods (methods gaan we later behandelen). Een variable bestaat uit maximaal 3 delen.

**Type:** Dit verteld wat voor informatie de variable zal bevatten. Een type kan een primair data-type zijn, maar ook een complex type zoals een class. Dit deel vertelt alleen over de form van de data, niet wat de waarde van die data is.

**Naam:** Dit is de naam van de variable, deze gebruik je om op een later moment de data weer op te vragen die je erin stopt.

**Toewijzing / Waarde:** Hier geven we aan wat de daadwerkelijke data van de variable is. Dat kan een directe toewijzing zijn, zoals bij een string of int. Maar het kan ook het resultaat zijn van een andere methode. Deze is optioneel om meteen op te geven, maar als je de waarde van te voren al weet, is het wel aangeraden deze meteen op de geven.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Fields

Nu word het verwarrend, dus let goed op. Een Field, is een stukje data die **binnen een class staat, maar buiten een methode**. Fields bevat data waarmee de methodes van de class mee kunnen werken, dit zijn zowel primaire datatypes als complexe datatypes.

Let goed op hoe je een Field schrijft. Het lijkt sterk op een variable, met wat dingen om rekening mee te houden.

**Accessor:** Een accessor vertelt of een buitenstaande class dit field mag aanroepen. Bij fields is dit doorgaans private. Dit houd in dan alleen onderdelen uit dezelfde class deze informatie mogen aanroepen.

**Type:** Het type van het field. Als een field *private* is, schrijven we deze vaak met een underscore aan het begin ‘\_’. Dit doen we zodat we ze kunnen onderscheiden van de property (waar we het zo over gaan hebben).

**Naam:** De naam van de field.

**Waarde / Toewijzing:** Dit is de daadwerkelijke waarde van de field. Het is bij fields vaak niet gezegd dat je van te voren weet welke informatie erin gaat.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Properties

Nu komen we bij de echte meat van het programmeren. Properties zijn elementen, vaak public, die stukken informatie bevatten over een complex type. Properties zijn contextueel verschillend van Fields. Properties bevatten stukken informatie over een type waar andere classes kennis over mogen hebben. Fields daarin tegen, zijn vaak alleen intern te benaderen.

In het volgende voorbeeld hebben we 3 properties. Eén van het type `string`, één van het type `int` en één van het type `bool`. Als ik deze class elders instantieer, kan ik deze properties van buitenaf benaderen.

Een Property bestaat uit de volgende onderdelen:

**Accessor:** Bij Properties is dit altijd public. Dit betekend dat andere classes en systemen toegang hebben tot de informatie van een property.

**Type:** Het type van de Property. Dit kan een primair datatype zijn of een complex type zoals een class, struct, enum, etc.

**Naam:** Dit is de naam die je gebruikt om te refereren naar de informatie in de property. Deze begin je altijd met een hoofdletter. Bij een samengesteld woord, schrijf je alle woorden met een hoofdletter.

**get; set; :** Nu worden properties interessant. De get; en set; zijn speciale mechanismes die een aantal zaken regelen. Hieronder zal ik er een aantal uitleggen, maar het is niet cruciaal om uit je hoofd te weten.

* *Automatische backing-field:* 
  + Een backing-field is een andere naam voor een Field, die we eerder hebben besproken. Normaal dien je een Field en een Property aan elkaar te verbinden. Maar dat hoeft tegenwoordig niet meer. Dit werkt alleen voor non-scoped get;set;. Scopes worden aangegeven met ‘{’ en ‘}’. Onderstaand staat een voorbeeld van een scoped get;set;.
* Validatie
  + Get en Set zijn onderwater function-calls, dus net als dat je een methode aan zou roepen (bijvoorbeeld ReadLine()). In de scope van get; en set; kan je interacties uitvoeren met de `value`.

Afbeelding met tekst

Automatisch gegenereerde beschrijvingAfbeelding met tekst

Automatisch gegenereerde beschrijvingAfbeelding met tekst

Automatisch gegenereerde beschrijving

## Parse

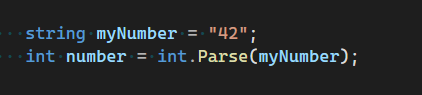
Het kan zijn dat je een stukje tekst hebt, een string, en die wilt omzetten naar een ander type, zoals een int. Er zijn een aantal verschillende manieren voor, in dit document gaan we de volgende methodes behandelen.

De `Parse` is de snelste manier om datatypes om te zetten. Als alle datatypes een vormpje hebben, zoals een rondje, vierkantje, driehoekje, ect, probeert `Parse` andere datatypes te forceren een andere vorm aan te nemen.

Het nadeel is dat Parse geen controles uitvoert, en dus niet controleert of het omzetten ook daadwerkelijk mogelijk is.

Een `Parse` doe je altijd vanuit het datatype van het resultaat; dus als je een string wilt omzetten naar een int, gebruik je de `Parse`-methode uit het int-datatype.

Alle primaire datatypes bevatten de `Parse`-methode.



# If’s en Loops

In dit hoofdstuk gaan we het hebben over control-statements en loops. Dit zijn fundamentele elementen die het mogelijk maakt voor programma’s om te reageren op verschillende situaties.

## If, else if, else

Wellicht al bekend. De if-statement. Een controle op een `true` of `false` waarde waarbij je wel of niet een bepaald stuk code uitvoert.

Je start een if-statement met het keyword `if`, gevolgd door de expressie/statement.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Je kan de if-statement ook uitbreiden met een alternatieve actie met een ‘else’.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

But wait! There is more!  
Het kan namelijk nog gekker. Je kan een statement toevoegen aan een else clause, namelijk door er direct een `if` achter te zetten.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Logical operators

Nu is het een goed moment om het te hebben over de logical operators. Klinkt lastig, valt best mee. Een logical operator is niks anders dan een token die LINKS en RECHT met elkaar vergelijkt, (even heel krom gezegd). Je ziet al 2 voorbeelden van logical operators in de afbeelding hierboven.  
Hieronder een tabel met nog een aantal andere operators.

|  |  |  |
| --- | --- | --- |
| Operator | Betekenis | Voorbeeld |
| == | <LINKS> GELIJK AAN <RECHT> | 1 == 2 |
| > | <LINKS> GROTER DAN <RECHT> | 1 > 2 |
| < | <LINKS> KLEINDER DAN <RECHTS> | 1 < 2 |
| >= | <LINKS> GROTER DAN OF GELIJK AAN <RECHTS> | 1 >= 2 |
| <= | <LINKS> KLEINER OF GELIJK AAN <RECHT> | 1 <= 2 |

## Conditional operators

Het kan voorkomen dat je een IF statement hebt waarbij je meerdere controles in één expressie wilt controleren, dus een IF statement waarbij 2 condities waar of onwaar moeten zijn, of 1 conditie waar en 1 conditie onwaar.  
Hiervoor kan je de volgende operators gebruiken.

|  |  |  |
| --- | --- | --- |
| Operator | Betekenis | Voorbeeld |
| && | <EXPRESSIE LINKS> EN <EXPRESSIE RECHTS> | 1 < 5 && 1 > 0 |
| || | <EXPRESSIE LINKS> OF <EXPRESSIE RECHTS> | 1 < 5 || 1 == 0 |

Afbeelding met tekst

Automatisch gegenereerde beschrijving

In het voorbeeld doe ik maar 2 condities controleren, maar dit kan je ook meerdere keren doen!

## For, ForEach en While

Voor herhalende taken kunnen we gebruik maken van loop. Deze komen in een aantal smaken, maar de essentie is altijd hetzelfde: een stuk code herhaaldelijk uitvoeren op bases van een voorafgaande expressie.

### While

De while-loop is de simpelste loop in het programmeren. Je start een while-loop met de `while`-keyword, gevolgd door een expressie. De expressie moet een bool resultaat teruggeven. Zolang dit resultaat `true` is.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Met dit mechanisme moet je wel zelf ervoor zorgen dat je ooit weer uit deze eeuwige while-loop komt. Dit kan je doen op een aantal manieren.

1. Je kan je while-expressie aanpassen zodat deze niet meer een `true` teruggeeft.
   1. `annoyingTeacher = false;`
2. Je kan een control statement gebruiken om geforceerd de uitvoering van de loop te onderbreken. Hier gaan we het zo verder over hebben.

### For

Een for-loop is ook wel bekend als een iteratieve-loop, en werkt op basis van een teller. Maar een for-loop moet je op een bepaalde manier opbouwen.

Een for-loop start je met de `for`-keyword, gevolgd door de volgende 3 onderdelen.

**(local) variable:** Dit is de iteratieve teller die bijhoud hoe vaak de loop zich al heeft herhaald. Dit is altijd int, long of short. Je kan geen decimalen gebruiken!

**Bool expression:** Dit deel bepaald of de loop wel of niet moet worden uitgevoerd. Dit is vaak expressie die zich uit als een boolean.

**Post-iteratieve expressie:** Dit deel word uitgevoerd als nadat de loop elke keer tot het einde is gekomen. Vaak wordt hier een incrementation gedaan van de local variable.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst, overdekt, sinaasappel

Automatisch gegenereerde beschrijving

### ForEach

Een ForEach-loop lijkt sterk op een for-loop, maar deze is wel degelijk anders. Een ForEach-loop itereert door objecten in een collectie en geeft toegang tot het specifieke object waar doorheen geïtereerd wordt.

Een foreach-loop start je met de foreach-keyword, gevolgd door de volgende onderdelen.

**Local variable:** Als eerste maak je een nieuwe variable met het hetzelfde type van de items in de collectie. Dit kunnen primaire datatypes zijn, maar ook complex types.

**Collectie:** Dit is de collectie waar je doorheen gaat itereren.

Onderstaand zie je een List<T>, waar <T> een string is. Dit houd in dat de List alleen objecten van het type `string` kan bevatten. <T> kan je ook vervangen met andere types.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Er is wel een ding wat je moet weten over de foreach. Je kan binnen een foreach niet de collectie aanpassen! In C# krijg je hier een exception voor, de reden hiervoor is om, onder andere, te voorkomen dat je een eeuwig durende foreach loop krijgt. Technisch gezien heeft het te maken met geheugen allocatie, maar dat is voor nu even te ver.

## Control statements

Het kan zijn dat, terwijl je in een loop zit, soms executie wilt overslaan of de hele loop wilt stoppen. Vaak doe je dit alleen als je weet dat er iets goed fout gaat en het veilig is om eerder te stoppen. Dit doe je met control-statement, ook wel control-syntax genoemd.

De control statements die je hieronder ziet werken voor alle loops!

### Continue

De continue syntax zorgt ervoor dat de huidige iteratie stopt, en de loop meteen doorgaat naar de volgende iteratie. Bij een for-loop word meteen de post-expressie aangeroepen en de loop gaat weer verder.

In een foreach betekend dit dat we de executie voor het huidige object stoppen, en meteen verder gaan met het volgende object in de collectie.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

### Break

Waar `continue` verder gaat in de iteratie, doet `break` dit niet. Break beëindigd de hele loop per direct!

In onderstaand voorbeeld, als er nog students zijn na de student met de naam “my evil twin”, het maakt niet uit. Zij worden dan niet meer behandeld! #sad

Afbeelding met tekst

Automatisch gegenereerde beschrijving

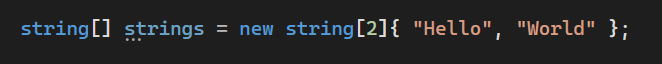
# Collections

In programmeren kan het vaak voorkomen dat je gaat werken met collecties, verzamelingen van types. We gaan een aantal veel voorkomende collecties behandelen.

Collecties werken nauw samen met de `for` en `foreach` loops.

## Array

De Array is de simpelste vorm van een collectie, en is heel letterlijk een plat geslagen lijstje van objecten. Bij het aanmaken van een array, moet van te voren aangeven welke objecten de array moet bevatten. Het is ook best lastig om nieuwe objecten toe te voegen en te verwijderen.  
Onderstaand zie je een voorbeeld van een string-array.



Een array-declaratie bestaat uit de volgende onderdelen.

**Identifier:** representeert welk typede arrayzal bevatten. Dit kunnen primaire als complexe datatypes zijn.

**Array-Identifier:** dit zijn de blokhaken die je na het datatype. Dit geeft aan dat iets een array is.

**Name**: de naam van de array waarmee je deze refereert.

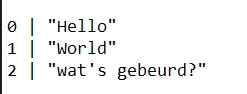
**New-keyword:** Creëert een nieuwe instantie van de array, deze is van zichzelf een complex type.

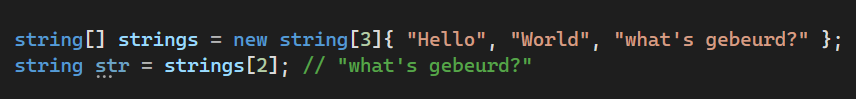
**Array-Type:** De array-type bepaald opnieuw de type en de lengte van de array.

**Initializer expression:** In de array expression schrijf je alle data op die in je array moeten. Het aantal objecten die je opschrijft MOET gelijk zijn aan de lengte die je hebt opgegeven.

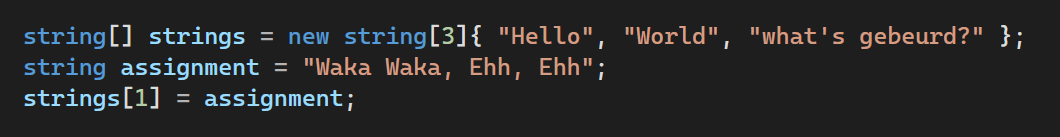
## Items opvragen, toevoegen, toewijzen en verwijderen

Om een item uit een array op te vragen, gebruiken we de index. De index start bij 0, dat is het eerst in een array. Een index schrijf je altijd in blokhaken na de variable naam.





### Items toewijzen

Waardes toewijzen doen we op dezelfde manier als bij het opvragen, namelijk door middel van de indexer.  


Op index 1, is “World” nu vervangen met de inhoud van de variable “assignment”.

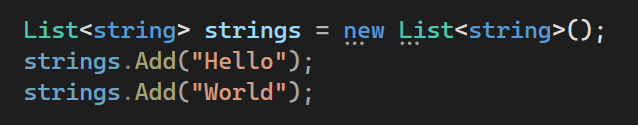
### Items verwijderen

Het verwijderen van items uit een array gaat niet zomaar. Een array is altijd een aantal items lang, en dit is niet meer aan te passen.  
Maar afhankelijk van het type, kan je wel `null` waardes invoeren.

## List<T>

De List is een van de populairste collecties. De List bevat veel functies om objecten te toe te voegen, te bewerken en te verwijderen. Een List is een soort schil met verschillende functies die om de verzameling heen zit. T is het type waarvan de collectie bestaat, dit kan een primair datatype zijn, maar ook een complex type.

Het aanmaken van een List<T> bestaat uit de volgende onderdelen. Hieronder staat een voorbeeld van een List<string>.



**Identifier:** Dit is altijd List.

**Type declaration:** Geeft aan wat voor type de objecten zullen hebben die in de List komen te staan.

**Name:** De naam van de list die je gaat gebruiken om later naar de list te refereren.

**New-keyword:** Creëert een nieuwe instantie van de List<string> in het geheugen.

**Initializer expression:** Hier schrijf je opnieuw de Identifier en de type gevolgd met ‘()’. Dit zorgt ervoor dat je de List kan gebruiken.

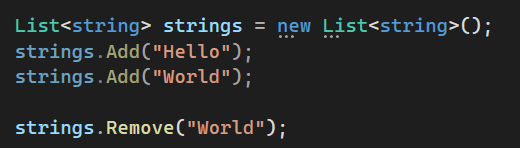
### Items toevoegen

Het toevoegen van items aan een List, doe je met de `Add()` methode. In de haakjes schrijf je de waarde van het object wat je toevoegt. Let op dat dit wel in het juiste type is.



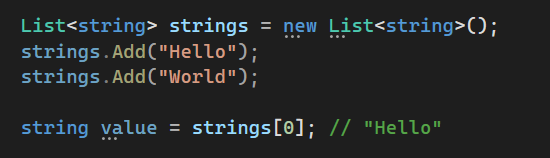
### Items verwijderen

Het is ook mogelijk om items uit een List te verwijderen. Dit doe je met de `Remove()` methode. In de haakjes, schrijf je dan de waarde van het item wat je wilt verwijderen.



### Items opvragen uit een lijst

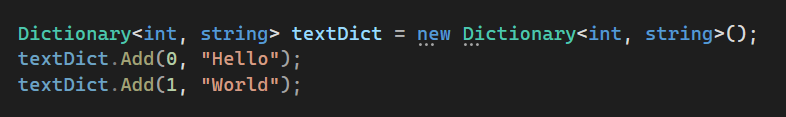
Het opvragen van een waarde uit een lijst gaat op de zelfde manier als bij de Array, namelijk op basis van index.



## Dictionary<T,U>

Een Dictionary is de meest gebruikte “complexe” collectie. Deze bestaat namelijk uit 2 types (die je zelf opgeeft in de Type-placeholders (T en U)). Een dictionary bestaat uit een KeyValueType, deze bestaat uit een sleutel (key), en een waarde (value).

Hieronder een voorbeeld van hoe je een Dictionary aanmaakt.



**Identifier:** Dit is altijd Dictionary.

**Type declaration:** Hier geeft je aan welk type zal worden gebruikt in de Key kant, en welk type zal worden gebruikt in de value kant. Deze kunnen verschillen of gelijk zijn. Je moet altijd twee types opgeven.

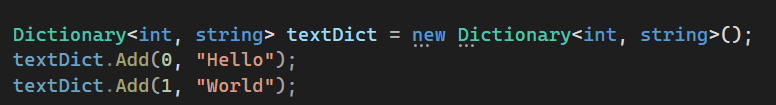
**Name:** De naam van de dictionary die je gaat gebruiken om later naar de collectie te refereren.

**New-keyword:** Creëert een nieuwe instantie van de Dictionary<T, U> in het geheugen.

**Initializer expression:** Hier schrijf je opnieuw de Identifier en de type gevolgd met ‘()’. Dit zorgt ervoor dat je de dictionary kan gebruiken.

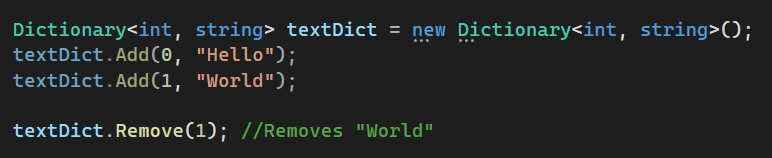
### Items toevoegen

Net als bij de List, maak je hier ook gebruik van de `Add()` methode. Hier dien je dan twee waardes op te geven. Namelijk een waarde voor de key, en een waarde voor de value. De types moeten overeenkomen met de types die je bij het aanmaken hebt aangegeven.



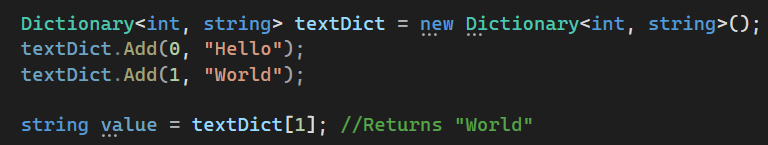
### Items verwijderen

Het verwijderen van een item uit de dictionary gaat met de `Remove()` methode. Hier vul je de waarde van de key in om het item te verwijderen.



### Items opvragen uit de dictionary

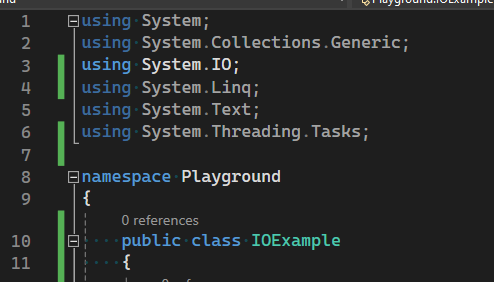
Net als bij de Array en de List, gebruik je bij Dictionary ook de indexer. Maar hier is het van belang om de juiste type te gebruiken voor het items wat je wilt opvragen.



# Files en IO

Het werken met files, directories, paths en het eeuwig opvullen van de HDD of SSD is een vaardigheid die je veel gaat gebruiken bij programmeren; en het kan op de extreem moeilijke manier, namelijk het zelf bepalen welke bytes op welke plek gaan in een file-stream. Maar het kan ook op de makkelijke manier. De makkelijke manier gaan we nu behandelen.

## Namespace

Als je wilt werken met files, moet je eerst de juiste namespace toevoegen aan de C# bestand, namelijk de `System.IO`-namespace.

## Simpel lezen van een bestand

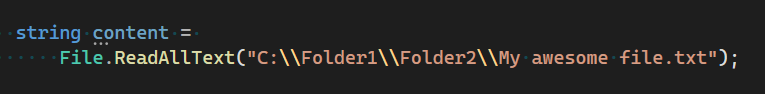
Nu we de `IO` namespace hebben, kunnen we gaan werken met bestanden. Maar we gaan maar een paar basis classes behandelen.

Om een bestand te lezen, moet je 1 ding van te voren weten:

* **De locatie van het bestand.**

Afhankelijk van wat je gebruikt, moet je zelf even uitzoeken hoe je aan het pad van een bestand komt. Voor Windows kan je het onderstaande doen:

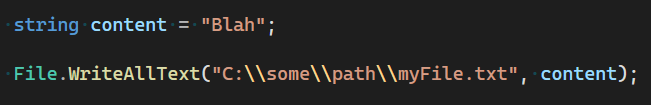
Navigeer in Verkennen naar je gewenste bestand, selecteer deze, houd de Shift-toets ingedrukt en druk op de rechtermuisknop. Selecteer in het context-menu de optie voor “Als pad kopiëren”.  
Nu staat het pad in je klembord.

Nu gaan we terug naar het programmeren. Om een bestand uit te lezen, gebruiken we de `File`- class, die te vinden is in de `System.IO`-namespace. In de `File`-class kan je een aantal handige methodes vinden, we zijn nu geïnteresseerd ‘ReadAllText()’.

## Simpel schrijven van een bestand

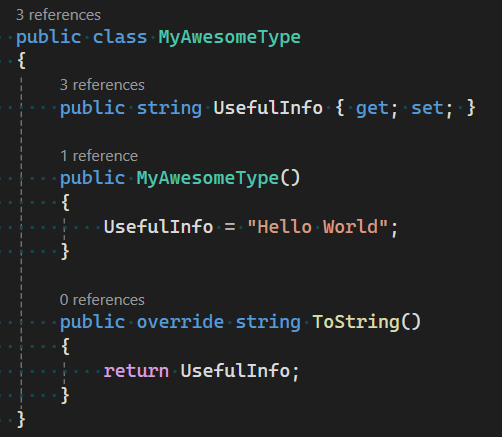
Voor het schrijven van een bestand, kunnen we dezelfde namespace en dezelfde class gebuiken. Het enige wat we aanpassen is de methode. Als we een bestand willen schrijven, moeten we twee dingen weten:

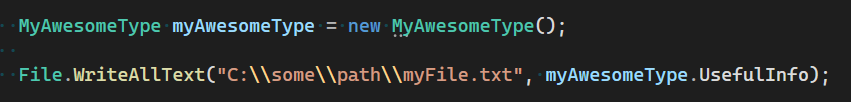
* **De locatie waar we ons bestand willen opslaan.**
* **De inhoud van ons bestand.**

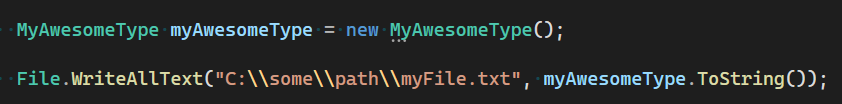
De inhoud van het bestand moet altijd een string zijn. Voor Primaire datatypes kan je makkelijk een ‘ToString’ methode aanroepen.  
Voor Complex Types is dit al lastiger; dan zal je specifiek een override moeten gebruiken op het type wat je wilt gebruiken, of een property moeten aanroepen waar nuttige informatie in staat.

### Complex types

We gaan het nog even snel hebben over het gebruiken van complex types. Als herrinnering, een complex type kan van alles zijn, een class, struct, enum, ect.

Op types zoals een class kan je overrides implementeren. Deze overschrijven standaard gedrag met een eigen implementatie. In onderstaand voorbeeld van de `MyAwesomeType`-class, is een override op de `ToString()` methode.  
Normaal als je `MyAwesomeType.ToString()` zou aanroepen, zou je de namespace en de naam van de class terug krijgen. Da’s niet handig. Daarom hebben we een override toegevoegd om nuttige informatie terug te geven.

 We kunnen dus een property van een complex type doorgeven, en die waarde wegschrijven in een bestand:

Of we kunnen gebruik maken van de `ToString` implementatie die wij hebben ‘overridden’.

# LINQ

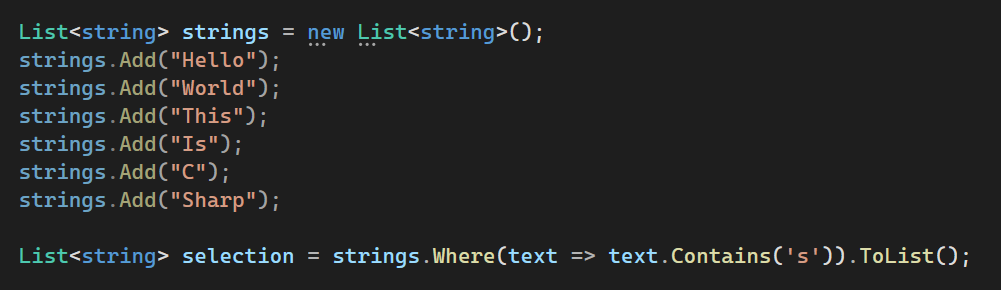
Linq is een manier van data opvragen uit een collectie waarbij je niet direct de index weet, ook wel bekend als een iteratieve actie. Er zijn een aantal verschillende methodes waarmee je zoek-acties uitvoert om zo 1 of meer stukken data op te vragen.

Alle collecties kunnen gebruik maken van de LINQ mechanieken. Hiervoor dien je wel een extra `using directive` te gebruiken, namelijk “System.Linq”.  
  
De aard van LINQ methodes zijn iets anders dan de methods die je wellicht al gewend bent. LINQ methodes werken op basis van een ‘predicate’. Een predicate bouw je op vanuit het punt dat elk item apart wordt gecontroleerd. Dat klinkt vaag, maar in de voorbeelden word het duidelijker.

We gaan niet alle LINQ methodes behandelen, maar wel een aantal die veel gebruikt worden.

## Where

De `Where()` methode gaat door elk item in een collectie heen, en onthoud welke items voldoen aan een bepaalde conditie. Uiteindelijk komt hier dan een selectie uit van items die aan de conditie voldeden.



**Text:** de naam die als referentie gebruikt zal worden om te controleren in de conditie.

**‘=>’:** Dit is een short-hand predicate assignment.

**Conditie:** Hier staat waar een item aan moet voldoen om onthouden te worden tijdens de iteratieve actie.

## Select

Select wordt gebruikt om items die voldoen aan een conditie simpel in een andere vorm te vouwen. Dit is handig als je een sub-collectie wilt maken van een andere collectie met aangepaste inhoud.

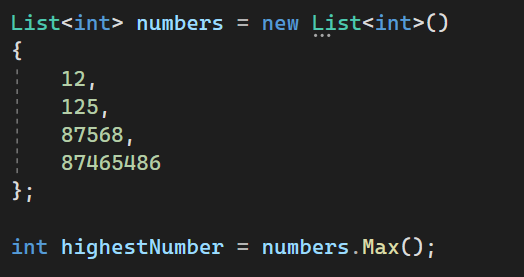


Eerst hebben we een standaard array met wat fruit. Vervolgens selecteren we elk item in de array, en zetten we deze om in een class instantie ‘Fruit’. De inhoud van de class is in dit geval niet van belang, en gaan we later verder op in.

## Max

De Max functie is het handigste als je met een collectie met getallen werkt. Er is geen conditie die je hier hoeft op te geven. De functie gaat door elk item heen en geeft het item terug met de hoogste waarde.

In theorie werkt deze methode met alle types. Maar dan is het resultaat afhankelijk van hoe deze zijn gebouwd. Maar dat is niet in scope van dit document.



# Programming architectuur

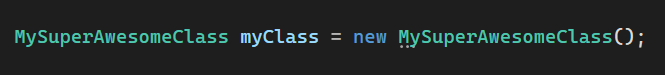
Je kan ervoor kiezen om alles in een console applicatie onder elkaar te schrijven. Dat werkt voor hele kleine apps, maar is op lange termijn niet vol te houden. Daarom kunnen we elementen onderbrengen in methods, classes, structs en interfaces.

## Classes

Classes zijn containers waar methodes, types, en andere elementen in staan. Classes zijn complex types en kunnen gebruikt worden als type-aanwijzing bij het aanmaken van variables.

Afbeelding met tekst

Automatisch gegenereerde beschrijving



Een class kan properties, methods, fields en andere classes bevatten. Een class kan ook van andere elementen erven door middel van een interface of base-class. Erving is een complex onderwerp waar we het een andere keer over gaan hebben.

## Constructors

Een Constructor is een eigenaardig wezen, en is afkomstig uit C++. In C++, prepareert de Constructor een Complex Type (een Class of Struct) om mee gewerkt te kunnen worden, properties worden toegewezen, werkgeheugen gereserveerd, fields krijgen een standaard waarden (als dit mogelijk is).

Het toevoegen van een constructor is technisch gezien optioneel, maar er zitten voordelen aan om deze wel toe te voegen. Je kan dan zelf bepalen welke standaard waarden of operaties er moeten worden uitgevoerd terwijl een nieuw object word aangemaakt.

Je kan variables maken van complex types. Dit doe je vaak door een toewijzing te starten met het ‘new’ keyword en de type. Zie [Variables, fields en Properties](#_Variables,_fields_en).

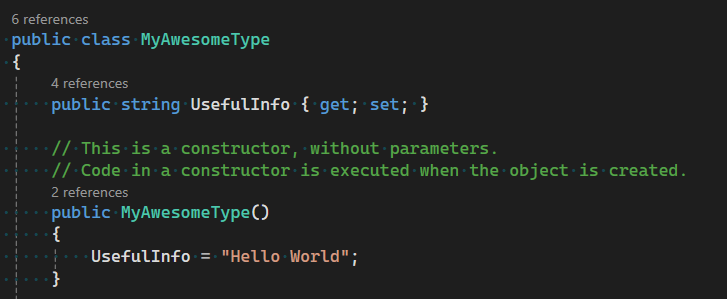
Een Constructor bestaat uit de volgende onderdelen:

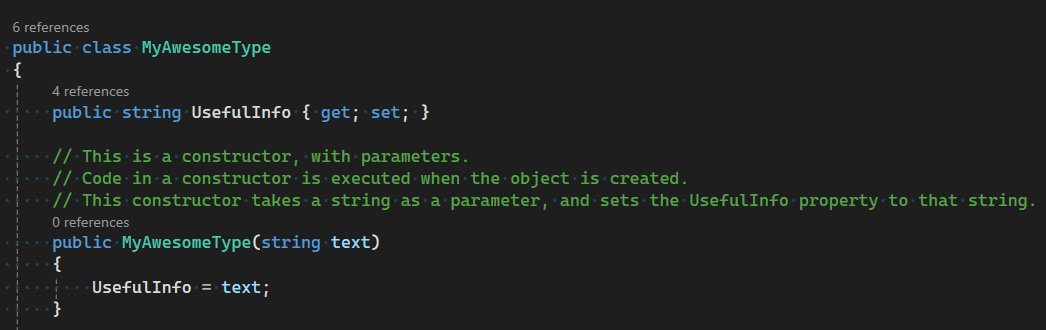
**Accessor:** Geeft aan of de constructor *public* of *private* is. Let op, als je een *private* gebruikt, dan kan je vaak NIET de Complex Type instantiëren.

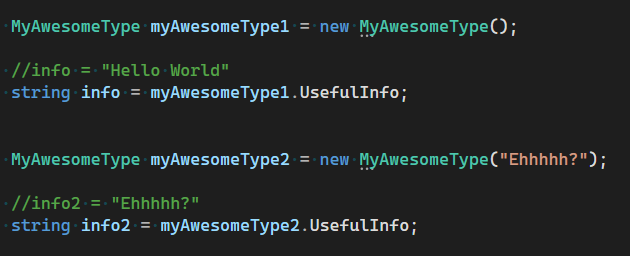
**Type-name:** Dit is de naam van het Complex Type (de naam die je hebt opgegeven tijdens het aanmaken van de Class of Struct).

**Parameters:** Optioneel. Parameters van zowel primaire datatypes als Complex types met een bijbehorende naam.

Hieronder nu twee voorbeelden van Constructors.

Dit is een Constructor zonder parameters, maar we geven nu een standaard waarde aan de property ‘UsefulInfo’.

Hier zie je een constructor met een parameter, namelijk een string ‘text’. Tijdens het uitvoeren van de constructor, krijgt de property ‘UsefulInfo’ de waarde van de ‘text’ parameter.

Het effect van deze constructors zien we pas als we onze type proberen te instantiëren, zoals in het voorbeeld hieronder.

## Methods

Methods, in de volksmond ook wel bekend als functions, zijn blokken uitvoerbare code. De code wordt uitgevoerd door de methode aan te roepen. Een method bestaat uit een aantal onderdelen.

**Accessor:** Geeft aan of de methode *public* of *private* is. Bij public mogen andere classes deze methode aanroepen. Bij private mogen alleen elementen binnen dezelfde class de methode aanroepen.

**Return Type:** Geeft aan wat voor type er wordt teruggegeven aan het einde van de uitvoering. Dit kan een primair type zijn, maar ook een Complex Type. Hier kan je ook *void* opgeven, dat geeft aan dat een methode geen waarde teruggeeft.

**Naam:** De naam van de methode. Dit schrijf je altijd als een werkwoord. Bedenk een goede naam zodat het duidelijk is wat de methode doet.

**Parameters:** De stukken extra die je mee geeft aan de methode om werk te doen. Parameters schrijf je binnen haken, ‘(’ en ‘)’. Het opgeven van parameters is optioneel, maar de haakjes zijn wel verplicht.

Afbeelding met tekst, overdekt, schermopname

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

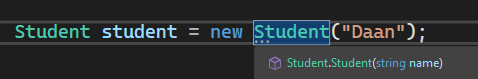
Onderstaand zie je een voorbeeld van een Class en hoe die eruit kan zien:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Bovenstaand is de class `Student`. Deze heeft een aantal public properties, die door andere kan worden opgevraagd en aangepast.

Kijk goed naar de Constructor: `public Student(string name)`.  
Dit houd in dat wanneer je een nieuwe Student initialiseert dat je altijd een string met de naam van de student moet opgeven.



# Afsluiting

Nu zijn we op het einde van het document. Dat houd niet in dat je nu alles weet over C#, eigenlijk weet je nu alleen de basis van het programmeren in C#. Experimenteer veel met verschillende projecten soorten, Console, Windows Forms, Blazor WebApp, MAUI en API zijn maar een klein aantal project soorten.

Hoe wordt je goed in C# en programmeren in het algemeen? Door veel te herhalen en veel te experimenteren. Er zijn altijd docenten die je ook kunnen helpen of kunnen inspireren als je een project zoekt om te proberen!

Vragen, klachten of opmerkingen?  
swind@novacollege.nl