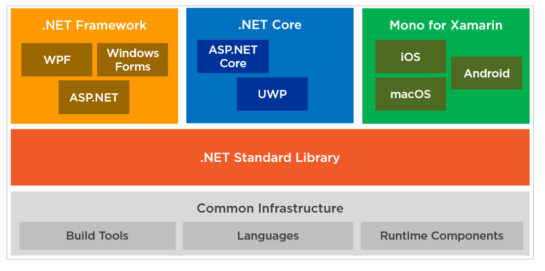
# Inleiding

**C#** is gebaseerd op de .NET technologie van Microsoft en is deel van de .NET familie samen met F# en Visual Basic. C# kan je gebruiken om zowat alle moderne applicaties te ontwikkelen in tegenstelling tot Visual Basic dat voornamelijk office applications maakt. C#7 is de nieuwste versie sinds 2017.

## Het .NET ecosysteem

Het .NET development platform onderging een grote herstructurering en is nog steeds in volle evolutie. Voor de ontwikkeleing van een .NET applicatie zijn er meerdere zogenaamde **runtimes** beschikbaar. Deze runtimes – **het .NET Framework, de .NET Core en Mono for Xamarin** – hebben elk hun specifiek toepassingsgebied.

Het .NET framework is **sinds 2002** het hart van de .NET technologie. In eerste instantie werd het .NET framework/C# gebruikt om applicaties te ontwikkelen die enkel op een Windows platform draaien (Windows, Windows Server, Windows Mobile, Windows Azure ...).

**ASP.NET** staat voor Active Server Pages en is een manier om met .NET talen webpagina’s te maken. Dit gaat dan om gegevens uitwisselen tussen de user en de server, niet om interactieve animaties.

Microsoft besliste om de ondersteuning van het .NET Framework voor andere platformen als **Linux**, **macOS** uit te breiden en om de broncode van het .NET framework **gedeeltelijk** beschikbaar te stellen als ***open source* software.** Dit werd de **.NET Core**.

**Mono** is de oorspronkelijke *open source* software implementatie van het .NET framework. Met **Xamarin**, dat gebaseerd is op Mono, is het mogelijk om met C# native applicaties voor **Android, iOS, MacOS** en Windows te ontwikkelen.

## .NET Framework 4.7

Het .NET Framework wordt **samen** met Windows geïnstalleerd en is dan beschikbaar voor alle applicaties gebaseerd op dit framework. Dit framework is platform-afhankelijk: het draait enkel op een Windows besturingssysteem. De reden hiervoor is dat deze applicaties soms gebruik maken van Windows specifieke API’s. Het is ook *monolithic*, wat wilt zeggen dat je alle components moet installeren of het werkt helemaal niet.

De .NET Framework runtime bestaat uit twee hoofdcomponenten:

* De **.NET Framework Class Library = FCL.** Dit is een grote **verzameling** van herbruikbare componenten waarvan de ontwikkelaar gebruik kan maken bij het ontwikkelen van zijn eigen applicaties. Deze **Class Library** kan in elke .NET programmeertaal gebruikt worden.
* De **Common Language Runtime = CLR**. Deze is verantwoordelijk voor het **uitvoeren** en **beheren** van .NET applicaties. Het framework zorgt er o.a. voor dat onderdelen die in verschillende .NET compatibele talen geschreven zijn, gecompileerd worden tot één werkende toepassing. De CLR staat ook in voor het geheugenbeheer, de beveiliging en de foutafhandeling tijdens de uitvoering van een .NET applicatie.

**Projecttypes** (of **workloads**) die met het .NET framework kunnen ontwikkeld worden:

* Console application :

Een commandline-application als .exe bestand.

* **Windows Client of Desktop Application :**

Een applicatie lokaal geïnstalleerd met een gebruikersinterface. Voorbeelden hiervan zijn Windows Office-applicaties, Photoshop ... . Voor de ontwikkeling hiervan heb je de keuze tussen twee technologieën:

* + **Windows Forms**
  + **WPF**: het **Windows Presentation Framework** dat op basis van **XML** de structuur en lay-out vastlegt.
* **ASP.NET Web application:**

Een webapplicatie die draait op een webserver en websites toont.

* + ASP.NET Web Forms: Het oudere framework dat moeilijker kan samenwerken met jQuery en CSS.
  + **ASP.NET MVC**: De nieuwere versie die webpagina’s opdeelt in Model, View en Controller. Hierdoor kunnen meerdere mensen makkelijker aan 1 webpagina werken en **jQuery** en **CSS** zijn ook gemakkelijker te gebruiken.
* Components and Controls:

Een library van herbruikbare componenten (met eventueel een grafische interface). Interessant om dezelfde code in verschillende WPF applications (of Windows Forms) te gebruiken. Dit wordt steeds opgeslagen als een **.dll** bestand.

* Windows Service:

Een verzameling classes zonder grafische gebruikersinterface. Het is een .exe bestand dat in de achtergrond taken zal uitvoeren (bijvoorbeeld na het starten van Windows).

* **Web Service:**

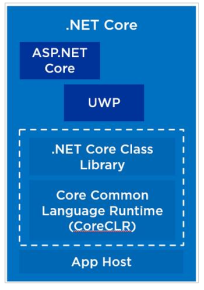
Een web service maakt het mogelijk gegevens op te vragen van een server. Hiermee kan men krachtige service-georiënteerde applicaties ontwerpen. De methods van deze web service kunnen ook vanuit andere programmeertalen worden opgeroepen zoals Java, C++, COBOL ... via:

* + **WCF (Windows Communication Framework)**
  + **Web API**, de communicatie met dit type web service verloopt via het **HTTP** protocol.
* **Cloud based Application:**

Met C# kan je applicaties ontwikkelen voor **Windows Azure**, het Microsoft besturingssysteem voor de **Cloud**. C# kan hierbij gebruik maken van **SQL Azure**, de cloud versie van Microsoft’s **SQL Server** **database** server.

## .NET Core

(Niet in deze cursus)

.NET Core is een modulair opgebouwde runtime die volledig open-source, platformonafhankelijk en geoptimaliseerd is voor de cloud en mobiele applicaties.

De .NET Core runtime bestaat uit:

* Een verzameling van class libraries: .NET Core Class Library of CoreFX
* Een runtime: de .NET Core Common Language Runtime of CoreCLR

Deze applicaties hebben een kleine voetafdruk en zijn zeer performant. De .NET Core is een toevoeging op het .NET Framework, geen vervanging. Het heeft zijn eigen specifieke toepassingen:

* **Console Application:**

Deze commandline-application draait zowel op Windows als op Linux als op macOS.

* **ASP.NET Core Web Application:**

MVC Web applicaties, Web API (HTTP) services ...

* **Universal Windows Platform Application (UWP):**

Ipv meerdere versies van dezelfde applicatie te ontwikkelen voor verschillende Windows devices kan er met UWP één applicatie ontwikkeld worden die op ieder Windows 10 Device werkt.

* **.NET Core Class Library:**

Een class library specifiek voor .NET Core.

! Bij .NET Core is er de keuze tussen:

* Framework dependant applicaties:

De .NET Core moet geïnstalleerd zijn op de computer waarop de applicatie zal draaien.

* Self-contained applicaties:

De applicatie en de .NET Core worden in 1 applicatie voorzien.

## Mono for Xamarin

(Niet in deze cursus)

Xamarin kan gebruikt worden voor alle native **mobile** **applicaties**. Xamarin helpt om een groot deel van de code steeds te hergebruiken. De ontwikkelde apps zien er hetzelfde uit, werken op dezelfde manier en zijn even performant als de apps in de specifieke programmeertaal. Het zijn **self-contained applicaties**.

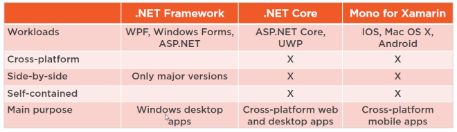
## .NET Standard

.NET Standard specificieert een verzameling van API’s die door de 3 runtimes geïmplementeerd worden. Zo kunnen libraries ontwikkeld worden die door alle .NET applicaties kunnen aangesproken worden, ongeacht op welke .NET runtime of op welk besturingssysteem deze applicaties draaien. In tegenstelling tot .NET Core.

## .NET Common Infrastructure

Alle runtimes gebruiken tools om code te schrijven, te compileren en uit te voeren. Zoals de CLR, CORECLR, de Garbage Collector ...

## Samengevat

De evolutie van het .NET Ecosystem zorgt ervoor dat men bij de ontwikkeling van een nieuwe applicatie zorgvuldig de vereisten van de applicatie moet nagaan.

# Visual Studio 2017 starten

Een ontwikkelomgeving om een C# applicatie te maken is Visual Studio (VS). Dit is een geïntegreerde ontwikkelomgeving (**Integrated Development Environment** of **IDE**), die alle hulpmiddelen bevat waarmee de programmeur op relatief snelle, efficiënte en visuele manier applicaties kan schrijven, uitvoeren, testen en publiceren. In Visual Studio kan je meerdere programmeertalen, waaronder C#, gebruiken. Er zijn ondertussen verschillende versies van VS die elk hun eigen versie van het .NET Framework bevatten.

Wil je een .NET applicatie ontwikkelen op een ander platform zoals Linux of macOS, installeer dan Visual Studio Code.

Visual Studio organiseert applicaties in **projects** en **solutions**.

Een **project** is een applicatie die je met VS ontwikkelt en alle gerelateerde bestanden. Een project is op zich een container met de nodige source-bestanden voor de applicatie.

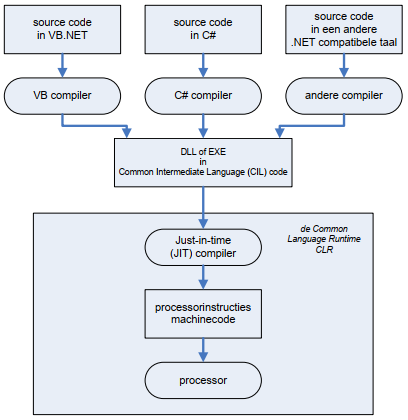
Een **solution** bevat één of meerdere projecten en is een totaaloplossing voor een (grote) applicatie. Een solution kan meerdere projecttypes bevatten, waarbij elk project met een andere programmeertaal kan ontwikkeld worden. Hoewel de **projecten in verschillende talen** geschreven zijn, kan je vanuit het ene project code oproepen van het andere project. Dit komt omdat alle programmeertalen in .NET voldoen aan de **CLS –** de **Common language Specification**.

Iedere **solution** krijgt een **eigen directory**. VS maakt per solution ook een inhoudsopgave aan, die de onderdelen van deze solution bevat. Het is een **bestand** met **extensie** ***sln***dat je met een gewone teksteditor als Notepad kan inkijken en eventueel kan wijzigen. Ieder **project** krijgt binnen de directory van de solution een eigen **subdirectory**. VS maakt per project een inhoudsopgave van de onderdelen van het project in een **bestand** met de **extensie *csproj***.

## Assembly

De broncode wordt vertaald door een compiler naar een **assembly** = de gecompileerde versie van een project met extensie .exe of .dll . In **.NET** wordt de broncode eerst vertaald naar een platformonafhankelijke ***tussentaal***, de zogenaamde **Common Intermediate Language** of CIL of IL. Om deze CIL-code te laten uitvoeren door een processor moet de **Common Language Compiler** (onderdeel van het .NET Framework) of CLR geïnstalleerd zijn. Pas wanneer het programma wordt uitgevoerd wordt de CIL-code door de **JIT** **Compiler** (**Just In Time Compiler**, ook een deel van het .NET Framework) omgezet naar processorinstructies. Vandaar dat de onderdelen van een .NET-applicatie in verschillende talen mogen geschreven worden. Deze worden namelijk allemaal vertaald naar **dezelfde tussentaal**. N**adeel** van deze werkwijze is dat bij iedere uitvoering van de applicatie de tussentaal nog vertaald moet worden. Het **voordeel** is dat de resulterende machinecode wel is aangepast aan het *device* en het programma pas wordt vertaald wanneer het nodig is. De JIT Compiler houdt bovendien de vertaalde code nog even bij in de cache.

De Compiler voor C# en VB.NET is bovendien *open source* en kan dus volledig geanalyseerd en geoptimaliseerd worden.



## Namespaces

**Namespaces** zijn een verzameling van classes die bij elkaar horen. Zo kan je meerdere classes met dezelfde naam gebruiken. In het begin van de *source* kan men het woord ***using*** gebruiken om namespaces te refereren. Standaard zijn er een paar *system* namespaces die al gelinkt zijn. In het eerste document starten we met een *namespace CSharpPFCursus* waarin dan classes omvat kunnen zitten. Zoals in dit geval de class *Program* met daarin een method *Main,* wanneer de applicatie start, start ze met deze method.

! Een namespace laat je steeds starten met een hoofdletter !

! Een class laat je ook steeds starten met een hoofdletter !

! Een method laat je ook steeds starten met een hoofdletter !

Een namespace definieer je door te starten met ***namespace***

Een class definieer je door te starten met ***class***

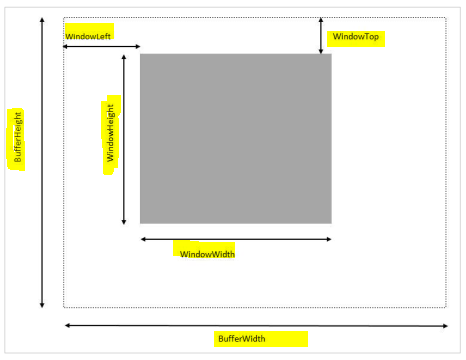
Een method definieer je door optioneel het sleutelwoord ***static.*** Ditgeeft aan dat dit een method met class bereik is. dwz dat je de method kan oproepen zonder eerst een object van de bijbehorende class te creëren. Bij de opstartmethod van een applicatie is *static* verplicht. Het sleutelwoord ***void*** geeft aan dat de method geen resultaatwaarde teruggeeft bij oproep. Hierna komt de naam van de method met de eventuele parameters tussen haakjes. Om een array te beschrijven, tik je eerst het type van de array (string), gevolgd door vierkante haakjes, gevolgd door een spatie, gevolgd door de naam van de array (args). Deze argumenten (en door de array kunnen dit er meerdere zijn) heten *command line arguments*.

## Tekst op het scherm

Het **scherm** is in .NET gekend als de class ***Console***.  
Je gebruikt de method ***WriteLine()***van de class *Console* om tekst op het scherm af te beelden. Dit is  
een static method (een method met class bereik). Dit wil zeggen dat je van de class ***Console*****niet**  
eerst een **object** moet creëren waarop je de method uitvoert, maar dat je de method via de class zelf kan uitvoeren. Bij deze method vermeld je tussen ronde haakjes wat je wil afbeelden. Vaste tekst vermeld je in C# tussen **dubbele** **aanhalingstekens**. Met Write() komt de cursor **NIET** op de volgende regel.

Andere properties en methods van **Console**:

|  |  |
| --- | --- |
| BufferHeight en BufferWidth | De Console is als het ware een venster op een onderliggend “document”. Dit document bestaat uit rijen of regels waarvan het aantal is vastgelegd in de property BufferHeight, en uit kolommen of tekens horizontaal, waarvan het aantal is vastgelegd in de property BufferWidth. Schrijf je meer regels dan er in de buffer kunnen, dan verdwijnt de eerste lijn uit de buffer (FIFO). |
| WindowHeight, WindowWidth en SetWindowSize() | De hoogte en breedte van de Console uitgedrukt in rijen (of regels) en kolommen (tekens naast elkaar). WindowHeight kan niet groter zijn dan BufferHeight en WindowWidth niet groter dan BufferWidth. Je kan de grootte van het venster op de buffer ook instellen met de method SetWindowSize(). |
| WindowLeft, WindowTop en SetWindowPosition() | Hiermee positioneer je het virtuele venster op de buffer en bepaal je welk deel van de buffer zichtbaar is in de Console. WindowLeft plus WindowWidth mag niet groter zijn dan BufferWidth, anders probeer je kolommen te tonen die er niet zijn. Op analoge wijze mag WindowTop plus WindowHeight niet groter zijn dan BufferHeight. Je kan de positie van het venster op de buffer ook instellen met de method SetWindowPosition(). |



|  |  |
| --- | --- |
| Clear() | Met deze method maak je de buffer helemaal leeg en dus ook de Console. |
| Title | Met deze property bepaal je welke tekst er in de titelbalk van het consolevenster verschijnt. |
| Read() en ReadLine() | Met deze methods lees je tekst in vanaf de console. Meer hierover in het hoofdstuk INVOER VANAF HET TOETSENBORD. |
| BackgroundColor, ForegroundColor en ResetColor() | Met de properties BackgroundColor en ForegroundColor bepaal je de achtergrond- en voorgrondkleur van de tekst die je in de Console afbeeldt. De method ResetColor() zet de kleuren terug op hun standaardwaarden. |
| CursorLeft, CursorTop en SetCursorPosition() | Via de properties CursorLeft en CursorTop of via de method SetCursorPosition() kan je de cursor verplaatsen in de buffer. Zo kan je dus bepalen waar tekst, die bijvoorbeeld met de method Write() naar de Console geschreven wordt, terecht komt. |

# De opbouw van een applicatie in VS

# Een eerste programma

# Lokale variabelen

De naam van een variabele moet aan de volgende regels voldoen:

* Hij moet met een **letter** **beginnen**
* Hij mag enkel **letters**, **cijfers** en **underscores** bevatten
* Hij mag maximaal **255** **tekens** bevatten
* Microsoft raadt aan variabelenamen te laten beginnen met **een kleine** **letter**

C# heeft twee grote categorieën van types variabelen:

* **Value types**: De variabele bevat zijn waarde
* **Reference types**: De variabele bevat een verwijzing naar een object. Dat object bevat zelf één of meerdere waardes en/of verwijzingen naar andere objecten.
* Variables declared within a method are "local variables"
* Variables declared within the class not within any methods are "member variables"(global variables).
* Variables declared within the class not within any methods and defined as static are "class variables".

C# kent de volgende Value types:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Type in CLR | Mogelijke inhoud variabele | Aantal bytes |
| byte | Byte | Geheel getal tussen 0 en 255 | 1 |
| sbyte | SByte | Geheel getal tussen -128 en 127 | 1 |
| short | Int16 | Geheel getal tussen-32.768 en 32.767 | 2 |
| ushort | UInt16 | Geheel getal tussen 0 en 65.535 | 2 |
| int | Int32 | Geheel getal tussen –2.147.483.648 en 2.147.483.647 | 4 |
| uint | UInt32 | Geheel getal tussen 0 en 4.294.967.295 | 4 |
| long | Int64 | Geheel getal tussen -9.223.372.036.854.775.808 en 9.223.372.036.854.775.807 | 8 |
| ulong | UInt64 | Geheel getal tussen 0 en 18.446.744.073.709.551.615 | 8 |
| float | Single | Decimaal getal tussen 1.5\*10-45 tot 3.4\*1038 (precisie ± 7 decimalen) | 4 |
| double | Double | Decimaal getal tussen 5.0\*10-324 tot 1.7\*10308 (precisie ± 15 decimalen) -> default bij komma | 8 |
| decimal | Decimal | Decimaal getal tussen 1.0\*10-28 tot 7.9\*1028 (precisie ± 28- decimalen) -> Voor GELD | 16 |
| char | Char | Eén teken (2 bytes voor Unicode ipv ASCII) | 2 |
| bool | Boolean | true of false | 1 |

***MaxValue()*** *om de maximumwaarde van een datatype te tonen (behalve boolean).*

## Declaratie

Je moet een variabele declareren vóór je ze gebruikt. Je declareert een variabele door eerst het type van de variabele te typen, gevolgd door een spatie, gevolgd door de naam van de variabele.

vb. *byte statenUSA; float alcoholDuvel, alcoholPalm;*

Variabelen een waarde geven via *literals*:

**bool** kan enkel de waarde *true* or *false* bevatten

**char** kan één teken bevatten omsloten met **enkele aanhalingstekens**

Als je een literal hebt dat een **geheel** **getal** is en **geen** suffix heeft dan zal VS een datatype kiezen (tussen int, uint, long en ulong) met zo weinig mogelijk bytes. Hetzelfde geldt voor byte, sbyte, short en ushort. Zelfs al ken je het zelf toe aan bijvoorbeeld een short.

Daarom, om conversies te vermijden en aan performantie te winnen, kan je gebruik maken van suffixen. Geheel getal literals met een **suffix** **u** of **U** worden door C# beschouwd als een ***uint*** of ***ulong***, l of L voor ***long***of ***ulong*** *en UL of LU en alle andere varianten* voor **ulong**.

Wil je een **decimal** of **float**, gebruikt dan m of M en f of F.

Decimalen worden aangeduid met een punt.

Je kan in één toekenningsopdracht eenzelfde waarde in meerdere variabelen invullen:

*alcoholDuvel = alcoholHapkin = 8.5f;*

Je kan variabelen ook meteen initialiseren door:

*byte statenUSA = 50;*

## Typeconversies

*double nieuwGetal = 1500;*

*int oudGetal = (int)nieuwGetal;*

Een *double* is groter dan een *int*. Bij de typeconversie gaan cijfers na de komma verloren als de variabele links een geheel getal type is. De waarde van het cijfer rechts moet ook binnen het bereik vallen van het datatype links, anders krijg je onverwachte resultaten.

*int* geeft ook de Unicode van een char of char leest de Unicode en zet deze om naar de overeenkomstige letter. *if (teken >= 'A' && teken <= 'Z')*

Je kan ook het woord *var* gebruiken om een variabele te declareren en dan bepaalt de compiler zelf welk type de variabele wordt op basis van de initiële waarde of expressie.

*var aantalKinderen = 3; --> int*

**! *var* kan enkel gebruikt worden om lokale variabelen te declareren. Dit kan dus NIET gebruikt worden bij de declaratie van CLASS VARIABELEN en CLASS PROPERTIES. Ook niet als returntype van een METHOD of als PARAMETER TYPE. !**

# Constanten

Je gebruikt **constanten** om **onveranderlijke** waarden te onthouden in het intern geheugen.

* Microsoft raadt aan een constante met een **hoofdletter** te laten beginnen

Als de constante uit een formule bestaat gebruikt men een gewone declaratie + het woord *readonly*. Dit zorgt ervoor dat de waarde van de constante maar één keer wordt berekend; bij de uitvoering van het programma .

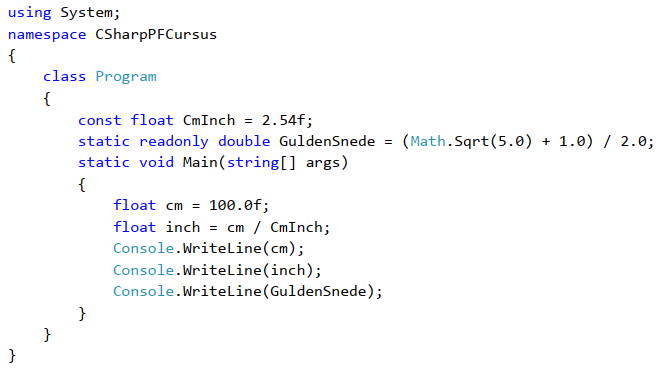
*(static)* ***readonly*** *double GuldenSnede=(Math.Sqrt(5.0) + 1.0) / 2.0;*

Als je deze waarde in een class definieert zal deze alsnog meerdere keren worden opgeslagen in het geheugen, wat zinloos is. Daarom gebruikt men in deze gevallen ook nog het woord *static.*

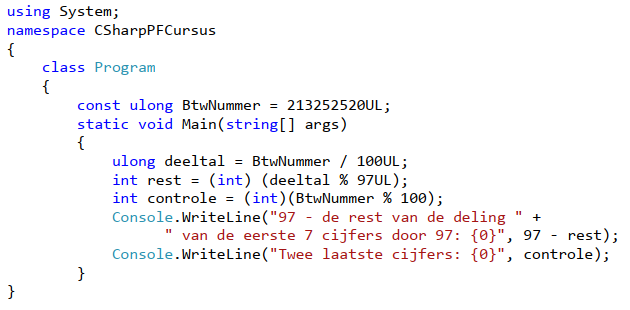
Constanten met een simpele waarde kan men gewoon definiëren met ***const***. Deze krijgen dan ook hun inhoud bij de compilatie van het programma.

# Bewerkingen

/ Als de deler én het deelgetal gehele getallen zijn, kapt C# het resultaat af tot een **geheel getal**.

% Restbepaling bij delingen

*vb. centimeters -> inches:*

**

*<-- vb. BTW-nummer controle*

wedde **+=** 2 ; wedde **-=** 3 ; wedde **\*=** 2 ; wedde **/=** 2 ; getal **%=** 5

**++** **vóór** de variabele verhoogt de variabele bij het begin van de opdracht, **++** er **ná** bij het einde.

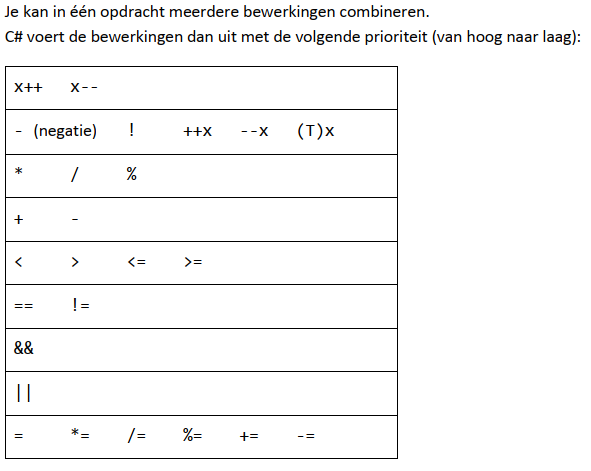
Dit valt ook toe te passen op een **char** variabele. Deze toont dan het vorige of volgende teken. Net zoals – en + om verschillen te weten te komen. en > of < om een bool terug te krijgen.

**Boolean** bewerkingen:

! waarde waarde1 && waarde2 waarde1 || waarde2

waarde1 ^ waarde2: **exclusive or**: *false* als beide waarden gelijk zijn

**Prioriteiten van bewerkingen:**



(T)x is een typeconversie zoals (int)getal.

De class *Math* van de .NET library bevat wiskundige constanten en methods die je in een programma kan gebruiken. De methods zijn *static* methods (methods met class bereik). Je kan deze dus via de class *Math* oproepen zonder eerst een object hiervan te maken.

|  |  |
| --- | --- |
| Constante of Method | Betekenis |
| PI | Het getal PI (3,14159265358979323846). |
| E | Het getal E (2,7182818284590452354). |
| Abs(getal) | Geeft de absolute waarde van getal. |
| Ceiling(getal) | Afronding naar boven, naar het kleinste geheel getal groter of gelijk aan getal. |
| Cos(getal) | Geeft de cosinus van getal. |
| Floor(getal) | Afronding naar beneden, naar het grootste geheel getal kleiner of gelijk aan getal. |
| Max(getal1,getal2) | Geeft getal1 als getal1 > getal2, anders getal2. |
| Min(getal1,getal2) | Geeft getal1 als getal1 < getal2, anders getal2. |
| Pow(getal1,getal2) | Geeft getal1 tot de macht getal2. |
| Round(getal) | Geeft de afgeronde waarde van getal = dichtstbijzijnde geheel getal: Console.WriteLine(Math.Round(1.44)); ◊ 1 Console.WriteLine(Math.Round(1.65)); ◊ 2 Let op: Console.WriteLine(Math.Round(1.5)); ◊ 2 Console.WriteLine(Math.Round(2.5)); ◊ ook 2 Er wordt afgerond naar het dichtsbijzijnde even getal. Met een extra parameter kan je de manier van afronden beïnvloeden: Console.WriteLine( Math.Round(1.5,MidpointRounding.AwayFromZero)); ◊2 Console.WriteLine( Math.Round(2.5,MidpointRounding.AwayFromZero)); ◊ 3 Een uitgebreide documentatie vind je in de MSDN. |
| Round(getal,decim) | Geeft de afgeronde waarde van getal op *decim* decimalen. |
| Sin(getal) | Geeft de sinus van getal. |
| Sqrt(getal) | Geeft de vierkantswortel van getal. |
| Tan(getal) | Geeft de tangens van getal. |

# Het type DateTime

DateTime is een **value type**, het bevat zelf de waarde en is geen verwijzing naar een andere plaats in het intern geheugen. Het kan een datum of een combinatie van datum én tijd zijn.

*DateTime beginTijdRekening = new DateTime(); (1)*

*DateTime beginEuro = new DateTime(2000, 1, 1); (2)*

*DateTime mijnGeboortedatum = new DateTime(1966, 8, 1, 10, 15, 0);*

*Console.WriteLine("Aankoopdatum: {0:dd-MM-yyyy}", AankoopDatum);*

Als je **geen waarde** meegeeft tussen de ronde haakjes, krijg je 1 januari van het jaar 1, middernacht.

Bij het afbeelden van datums houdt .NET rekening met de instellingen op de computer waarop het programma **draait.** Je kan ook zelf het datumformaat kiezen:

*Console.WriteLine(eenDatum.ToShortDateString());*  --> 1/03/2018 (afhankelijk van de landinstellingen)

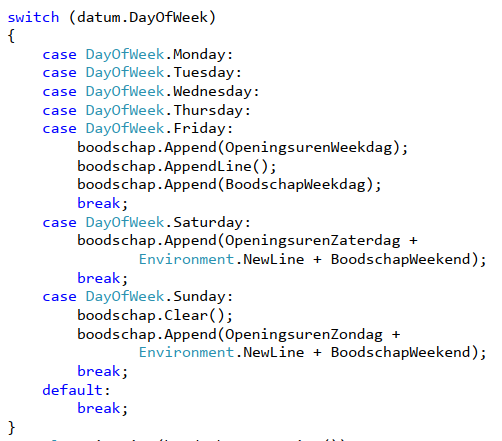
*Console.WriteLine(eenDatum.ToString("yyyy-MM-dd"));*  --> 2018-03-01



**! Sommige properties zijn *static* properties en sommige methods zijn *static* methods.** Dit zijn properties en methods die je niet op een DateTime variabele kan toepassen, maar enkel op het type DateTime zelf. Een voorbeeld hiervan is de property *Today* !

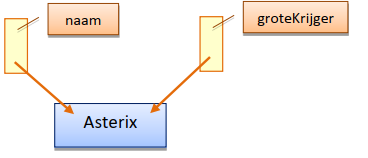
* We gebruiken een variabele geboortedatum met de waarde 1 augustus 1966, 4:30:15.

|  |  |
| --- | --- |
| Method of Property | Resultaat van deze method of property |
| AddDays(dagen) | Geeft een nieuwe datum die dagen verwijderd is van de datum. geboortedatum.AddDays(3) geeft 4/8/1966 4:30:15. |
| AddYears(jaren) | Geeft een nieuwe datum die jaren verwijderd is van de datum. geboortedatum.AddYears(-5) geeft 1/8/1961 4:30:15. Soortgelijke methods: AddMonths(maanden), AddHours(uren), AddMinutes(minuten), AddSeconds(seconden), AddMilliseconds(milliseconden). |
| Year | Het jaartal van de datum. geboortedatum.Year geeft 1966. Soortgelijke properties: Month, Day, Hour, Minute, Second, Millisecond. |
| DayOfWeek | De dag in de week van de datum. 0 = zondag, 1 = maandag, ... 6 = zaterdag. geboortedatum.DayOfWeek geeft Monday. (int) geboortedatum.DayOfWeek geeft 1. |
| DayOfYear | De hoeveelste dag in het jaar de datum is. geboortedatum.DayOfYear geeft 213. |
| DaysInMonth(jaar, maand) | Het aantal dagen in de maand van jaar (static property). DateTime.DaysInMonth(2000,2)geeft 29. |
| Today | De systeemdatum zonder tijd (static property). DateTime.Today geeft de systeemdatum zonder tijd. |
| Now | De systeemdatum met tijd (static property). DateTime.Now geeft de systeemdatum met tijd. |

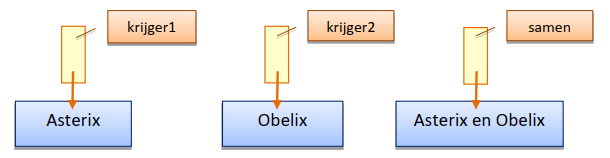


# Het type string

Variabelen van het type **string** zijn **reference** variabelen, geen value variabelen.



*string groteKrijger; groteKrijger = naam;*

*string samen = krijger1 + “ en “ + krijger2*

Dit komt omdat strings ***immutable*** zijn. dwz dat éénmaal de string een waarde gekregen heeft, deze inhoud niet kan gewijzigd worden. Om dit te vermijden kan je de class *StringBuilder* gebruiken uit de namespace *System.Text* wanneer de string vaak moet aangepast worden.

***System.Text.StringBuilder*** *bericht = new System.Text.StringBuilder(“hallo”); bericht.Append(“ iedereen”);*

## Strings vergelijken en bewerken

Met == en != controleer je een string op zijn inhoud. Met de *CompareTo()* method van een string variabele controleer je of die string voor of na een andere string in het alfabet komt.

***CompareTo(andereString)*** geeft:

0 als de string gelijk is

-1 als de string alfabetisch vóór de andereString komt

1 als de string ná de andereString komt

! Hierbij maakt C# een onderscheid tussen hoofd- en kleine letters. !

**Veelgebruikte methods en properties bij string variabelen:**

We gebruiken een string variabele naam met de waarde Asterix en Obelix.

|  |  |
| --- | --- |
| Method of property | Resultaat van deze method of property |
| ToUpper() | De voorstelling in hoofdletters. naam.ToUpper() geeft ASTERIX EN OBELIX. |
| ToLower() | De voorstelling in kleine letters. naam.ToLower() geeft asterix en obelix. |
| Length | Het aantal tekens. naam.Length geeft 17. |
| StartsWith(tekenreeks) | true als de string met de tekenreeks begint, anders false. naam.StartsWith("Aste") geeft true. Deze method is hoofdlettergevoelig. naam.StartsWith("aste") geeft false. |
| EndsWith(tekenreeks) | true als de string op de tekenreeks eindigt, anders false. naam.EndsWith("x") geeft true. Deze method is eveneens hoofdlettergevoelig. |
| IndexOf(tekenreeks) | De positie van de tekenreeks in de string. Als de tekenreeks helemaal vooraan in de string voorkomt, krijg je nul. Als de tekenreeks niet in de string voorkomt, krijg je -1. Als de tekenreeks meerdere keren in de string voorkomt, krijg je de positie van de eerste vindplaats. naam.IndexOf("st") geeft 1. Deze method is eveneens hoofdlettergevoelig. |
| LastIndexOf(tekenreeks) | De *laatste* positie van de tekenreeks in de string. naam.LastIndexOf("x") geeft 16. Deze method is eveneens hoofdlettergevoelig. |
| Substring(positie) | Het stuk tekst uit de string vanaf een *positie*. Het eerste teken van de string heeft het volgnummer 0. naam.Substring(2) geeft terix en Obelix. |
| Substring(positie, aantal) | Het stuk uit de string vanaf een *positie* over *aantal* posities. naam.Substring(2, 5) geeft terix. |
| Insert(positie, invoegtekst) | De tekst waarin op *positie* de *invoegtekst* is ingevoegd. naam.Insert(0, "De vrienden ") geeft De vrienden Asterix en Obelix.  ibanRekeningNr = ibanRekeningNr.Insert(12, " ") .Insert(8, " ").Insert(4, " "); |
| Remove(positie, aantal) | De tekst waarin op *positie aantal* tekens verwijderd zijn. naam.Remove(8,3) geeft Asterix Obelix. |
| Replace(oudetekst, nieuwetekst) | De tekst waarbij *oudetekst* vervangen is door *nieuwetekst*. naam.Replace("x", "s") geeft Asteris en Obelis. |
| Trim() | De tekst waaruit de spaties vooraan én achteraan verwijderd zijn. |
| TrimStart() | De tekst waaruit spaties vooraan verwijderd zijn. |
| TrimEnd() | De tekst waaruit de spaties achteraan verwijderd zijn. |

**Het individuele teken van een string krijg je door deze tussen vierkante haakjes te plaatsen:**

***string[0]***

## Converteren

Ieder **value** type heeft een ***ToString()*** method.

Je converteert een **string** naar **value** type via één van de volgende *static* methods (methods met class bereik) van de class *Convert*.

|  |  |
| --- | --- |
| Method | Betekenis |
| ToByte(string) | string converteren naar byte |
| ToSByte(string) | string converteren naar sbyte |
| ToInt16(string) | string converteren naar short |
| ToUInt16(string) | string converteren naar ushort |
| ToInt32(string) | string converteren naar int |
| ToUInt32(string) | string converteren naar uint |
| ToInt64(string) | string converteren naar long |
| ToUInt64(string) | string converteren naar ulong |
| ToSingle(waarde) | string converteren naar float |
| ToDouble(waarde) | string converteren naar double |
| ToDecimal(waarde) | string converteren naar decimal |
| ToBoolean(waarde) | string converteren naar bool |
| ToDateTime(waarde) | string converteren naar DateTime |

Een **betere** method is ***Parse()***:

*byte kinderen = byte.Parse(kinderenWoord);*

## Verbatim strings

Om speciale tekens te gebruiken, gebruikt men een ***escape sequence***:

|  |  |
| --- | --- |
| * Escape sequence | Betekenis |
| \n | nieuwe lijn |
| \t | tab |
| \" | dubbel aanhalingsteken (") |
| \' | enkel aanhalingsteken (') |
| \\ | backslash (\) |
| \a | het belsignaal laten horen |

Je gebruikt dezelfde escape sequences om deze tekens in te vullen in een variabele van het type  
char.

Als je een string laat voorafgaan door een @, dan wordt dit een ***verbatim string***. Hierin hebben alle tekens terug hun letterlijke betekenis. Om een dubbel aanhalingsteken te vermelden moet je deze nog wel laten voorafgaan door een dubbel aanhalingsteken.

Verbatim strings kan je splitsen over meerdere regels. De tekst wordt letterlijk getoond zoals weergegeven, inclusief enters en spaties.

## Een lege string

op 2 manieren:

* *“”*
* *string.Empty* --> deze constante kan rechtstreeks via de class *string* opgeroepen worden

een handige *static* method is ***string*.*IsNullOrEmpty(tekst)****.*

of wanneer deze ook gevuld kan zijn met enkel spaties: ***string.IsNullOrWhiteSpace(tekst)***.

## Placeholders

*Console.WriteLine(“”); of string.Format(“{0} en {1} zijn goede vrienden “, krijger1, krijger2);*

Deze werkwijze verdient de voorkeur boven het *+* teken owv de leesbaarheid. Je kan de placeholder ook optioneel een opmaakcode geven. Bijvoorbeeld *{0:d MMMM yyyy}* geeft 1 januari 2018.

## Interpolated strings

Een string voorafgegaan door een **dollar-teken** (**$**). Een dergelijke string kan **accolades met een expressie** bevatten.

*double straal = 5.0;  
Console.WriteLine($"De omtrek van een cirkel met straal {straal} cm" +  
$" bedraagt {2 \* Math.PI \* straal} cm.");  
//omtrek met 2 decimalen  
Console.WriteLine($"De omtrek van een cirkel met straal {straal} cm" +  
$" bedraagt {2 \* Math.PI \* straal:0.00} cm.");*

Bij het afbeelden wordt de expressie {straal} vervangen door de waarde van de variabele *straal*, en de expressie {2 \* Math.PI \* straal} door het resultaat van de berekening. In een interpolated string kan je de expressie eveneens van een **opmaakcode** voorzien. Je plaatst deze code binnen de { }; ná de expressie en gescheiden van de expressie door een dubbel punt. Als een interpolated string over verschillende regels splitst, gebruik je het + teken om de verschillende delen samen te voegen en schrijf je voor elk deel het $ teken.

of bijvoorbeeld: *Console.WriteLine($"Polishouder: {Polishouder}");*

# Invoer vanaf het toetsenbord

## Console.ReadLine()

Met de *static* method ReadLine van de class *Console* kan je een tekst, die de gebruiker met het toetsenbord intikt, inlezen. Deze wacht tot men op de ENTER-toets drukt. De method geeft deze **altijd** als een **string** terug.

*float cm = float.Parse(Console.ReadLine());*

# Selecties

## if

*if (voorwaarde)*

*{*

*opdracht*

*opdracht*

*}*

## if ... else ...

*if (voorwaarde1)*

*{*

*opdracht*

*}*

*else if (voorwaarde2)*

*{*

*opdracht*

*}*

*else*

*{*

*opdracht*

*}*

## De ? operator

**ternary operator** of **conditional operator**

*leeftijd < 18 ? “Niet toegelaten” : “Kom maar binnen”;*

## switch

*switch(expressie)*

*{*

*case vergelijkwaarde1:*

*opdracht*

*break;*

*case vergelijkwaarde2:*

*opdracht*

*break;*

*default: (niet verplicht)*

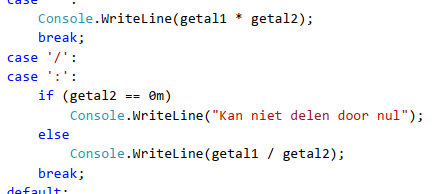
*opdracht*

*break;*

*}*

Iedere case dient afgesloten te worden met een ***break***, behalve als er een ***return*** of ***throw*** staat.

Vóór C#7 moest de expressie als type een *geheel getal, char, string, bool of* ***enum*** zijn. Ondertussen werd deze uitgebreid met ***Pattern Matching*** mogelijkheden en kan je op eender welk type testen.



! Als je 2 of meerdere keren dezelfde opdrachten wil laten uitvoeren zonder ze meerdere keren te meoten schrijven, kan je cases op elkaar laten volgen zonder ze af te sluiten. Belangrijk hierbij is dat de eerste cases **geen** code bevatten.

**OPMERKING:**

Je kan ook **when** gebruiken in een switch-statement.

# Iteraties

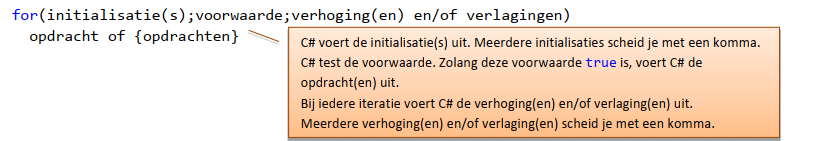
## while

C# kan de iteratie minstens één keer uitvoeren of niet door

* ***while*** *(voorwaarde) {opdrachten}*
* ***do*** *{opdrachten}* ***while*** *(voorwaarde)*.

Je kan deze iteratie voortijdig verlaten met de opdrach ***break***.

## for



Gezien je de teller in de *for* declareert is ze enkel binnen die *for* gekend.

De 3 onderdelen van de *for* zijn elk **optioneel**. De volgende opdracht geeft een eindeloze iteratie die je met Ctrl+C kan afbreken.

for (; ; )

Console.WriteLine(“En ze draaiden maar door ...”);

Je kan een *for* loop ook voortijdig verlaten met de opdracht ***break***.

# Declaratieplaats van variabelen in een method

De volgende regels gelden:  
• Als je een variabele vooraan in een method declareert, is deze variabele gekend in de gehele method.  
• Als je een variabele halverwege een method declareert, maar niet binnen een if, else,  
switch, while of for, is deze variabele vanaf die plaats gekend tot het einde van de method.  
• Als je een variabele binnenin een if, else, switch, while of for declareert, is deze variabele enkel gekend tot het einde van die if, else, switch, while of for.

# Arrays

## Één-dimensionele arrays

Een rij variabelen **van hetzelfde type**.\*

***type[] arrayNaam=new type[aantalElementen];***

Een voorbeeld: een array declareren met 12 elementen van het type double:

*double[] temperaturen=new double[12];*

De variabele *temperaturen* is enkel een **reference variabele**. De variabele bevat zelf niet de array maar een verwijzing naar de array in het intern geheugen.

Men kan ook eerst de variabele maken en hierna deze laten verwijzen naar een nieuwe array:

double[] temperaturen;

temperaturen=new double[12];

C# initialiseert alle elementen van de array volgens hun type:

* Elementen van een array met als type een getal (int, double, decimal,...) krijgen een **beginwaarde nul**.
* Elementen van een array met als type char krijgen als beginwaarde het teken met **Unicodewaarde 0**.
* Elementen van een array met als type bool krijgen een **beginwaarde false**.
* Elementen van een array met als type DateTime krijgen een **beginwaarde 1/1/1 0:0:0**.

Als je naar een niet-bestaand element uit een array verwijst, stopt het programma met een fout (een exception).

## Elementen initialiseren bij declaratie

**type[] arrayNaam={waarde1,waarde2, ...};**

De array bevat evenveel elementen als dat je beginwaarden opgeeft.

**Implicitly Typed Arrays**

Hierbij wordt het type van de array bepaald op basis van de elementen die opgegeven worden bij de initialisatie van de array.

**var getallen = new[] {10,20,30};**

## foreach

Met foreach loop je met een iteratie door alle elementen van een array.

**foreach (type** variabeleNaam **in** array) {opdracht(en)}

De variabele die je bij de foreach declareert, moet hetzelfde type hebben als de elementen van de array. Deze variabele is enkel binnen de foreach gekend. Deze variabele bevat bij de 1e iteratie de waarde van het 1e element ... . De iteratie stopt automatisch als alle elementen van de array doorlopen zijn. Je kan met foreach de elementen **enkel lezen**, **niet wijzigen.**

Handiger hierbij is als type het **sleutelwoord** **var** te gebruiken.

## Twee-dimensionele arrays

**double[,] voorbeeld = new double[4,7];**

**string[,] hoofdstedenLanden={{“Cuba”,”Havana”,”Spaans”},{“Brazilië”,”Brasilia”,”Portugees”}};**

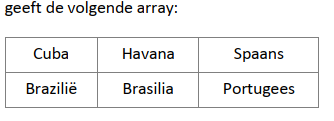
of

**var hoofdstedenLanden = new [,]**

**{**

**{“Cuba”, “Havana”,”Spaans”},{“Brazilië”,”Brasilia”,”Portugees”}**

**};**



hoofdstedenLanden[rij, kolom]

## De class Array

De class Array bevat static methods (methods met class bereik)

|  |  |
| --- | --- |
| Sort(array) | sorteert de één-dimensionele array array oplopend. |
| Reverse(array) | wisselt de elementen in de één-dimensionele array array om:  eerste met laatste, tweede met voorlaatste, ... |
| Copy(vanArray,naarArray,aantal) | Kopieert elementen van de array vanArray naar de array naarArray.  Het kopiëren begint vanaf het 1° element.  o parameter aantal:  het aantal te kopiëren elementen. |
| Copy(vanArray,vanafElement, naarArray,naarElement,aantal) | Kopieert elementen van de array vanArray naar de array naarArray.  o parameter vanafElement:  vanaf welk element uit de array vanArray begint het kopiëren.  o parameter naarElement:  vanaf welk element uit de array naarArray begint het kopiëren.  o parameter aantal:  het aantal te kopiëren elementen. |
| IndexOf(array,zoekWaarde) | Geeft het volgnummer van het eerste element uit de één-dimensionele array array, dat gelijk is aan zoekWaarde.  Geeft -1 als geen enkel element gelijk is aan zoekWaarde. |
| LastIndexOf(array,zoekWaarde) | Geeft het volgnummer van het laatste element uit de één-dimensionele array array, dat gelijk is aan zoekWaarde.  Geeft -1 als geen enkel element gelijk is aan zoekWaarde. |
| Clear(array,vanaf,aantal) | Plaatst elementen in de array array terug op hun beginwaarde  (elementen van het type int op 0, van het type bool op false, ...)  o parameter vanaf:  het volgnummer van het element waar het initialiseren begint.  o parameter aantal:  aantal te initialiseren elementen vanaf het element met volgnummer vanaf. |

# Enum

Met een **enum** beschrijf je een nieuw type variabele als een **opsomming** van **constanten**. Een variabele die je op dit nieuwe type baseert, kan als inhoud **enkel** **één** van de gedefinieerde **constanten** krijgen.

Microsoft stelt voor de naam van een **enum** en **iedere waarde** binnen een enum met een **hoofdletter** te beginnen, en elk apart woord binnen de naam ook met een hoofdletter te beginnen.

Voorbeelden van enum:

* De enum **Seizoen** met als mogelijke waarden Lente, Zomer, Herfst, Winter.
* De enum **Geslacht** met als mogelijke waarden Man, Vrouw.
* De enum **BladLigging** met als mogelijke waarden Staand of Liggend.

Je kan een enum laten voorafgaan door een **sleutelwoord** dat het toegangsniveau van deze enum aangeeft:

|  |  |
| --- | --- |
| Sleutelwoord | Betekenis |
| public | Je kan de enum aanspreken vanuit alle code van de applicatie. |
| private | Je kan de enum enkel aanspreken vanuit de class waarin je de enum declareert. |
| protected | Je kan de enum enkel aanspreken vanuit de class waarin je de enum declareert en vanuit afgeleide classes (zie verder). |
| internal | Je kan de enum enkel aanspreken in de assembly waarin je de enum declareert. Een assembly is de gecompileerde versie van het project (een exe of dll). |
| protected internal | Je kan de enum enkel aanspreken in de assembly waarin je de enum declareert en in afgeleide classes (zelfs als deze niet tot de huidige assembly behoren). |

* Voor een enum gedeclareerd **binnen** een **class**, kan je het **toegangsniveau** **vrij** kiezen uit de bovenstaande lijst.
* Voor een enum gedeclareerd **buiten** een **class**, kan je enkel het sleutelwoord **public** of **internal** gebruiken.
* Als je **geen** enkel van deze sleutelwoorden bij een enum vermeldt, krijgt de enum het toegangsniveau **public**.

Voorbeeld:

using System;

namespace CSharpPFCursus

{

public enum Seizoen

{

Lente, Zomer, Herfst, Winter

}

class Program

{

static void Main(string[] args)

{

Seizoen plukseizoen = Seizoen.Herfst;

Console.WriteLine(plukseizoen);

Console.WriteLine((int)plukseizoen);

}

}

}

Je declareert een variabele van het type Seizoen. Je vult in de variabele één van de mogelijke Seizoen waarden in. Bemerk dat je de waarde laat voorafgaan door de naam van de enum en een punt. Door deze volledige verwijzing is er geen conflict als een andere enum ook de waarde Herfst bevat.

Je ziet het getal 2. C# kent intern aan iedere enum waarde een getal toe, beginnend vanaf 0:

Lente=0, Zomer=1, ...

Je kan aan de waarden van een enum ook zelf een getal toekennen:

public enum Seizoen

{

Lente = 1, Zomer = 2, Herfst = 3, Winter = 4

}

Als je een waarde of meerdere waarden geen nummer geeft, nummert C# verder vanaf de waarde

die je wel een nummer gegeven hebt. Je kan het laatste voorbeeld dus ook schrijven als:

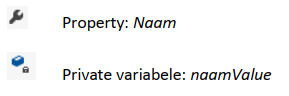
public enum Seizoen

{

Lente = 1, Zomer, Herfst, Winter

}

# Classes, Objects en object variables

De meeste onderwerpen die tot nu aan bod kwamen (variabelen, selecties, iteraties ...) komen terug in alle klassieke programmeertalen. Vanaf dit onderwerp pas je objectgeoriënteerd programmeren toe in C#.

## Class

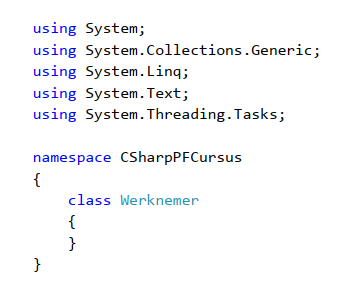
Met een *class* beschrijf je zelf een gegevenstype. Een class beschrijft een begrip of voorwerp uit de werkelijkheid. In de analyse van de toepassing die je programmeert, bepaal je welke classes je nodig hebt. Bijvoorbeeld in een banktoepassing beschrijf je de classes *Klant, Rekening, Overschrijving ...* Deze klasses hebben properties en methodes.

Een class is enkel een beschrijving (of **prototype**) van een gegevenstype. Op basis van deze classes maak je *objecten* of *instanties*. Deze krijgen dan een plaats in het intern geheugen. In je code verwijs je naar een object via een ***reference variable*** (of **object variabele**). Je kan via de reference variabele de properties **invullen** en **opvragen** en de **methods** van het object **uitvoeren**.

## Class creëren

Microsoft raadt aan de naam van een class met een **hoofdletter** te **beginnen**. Je kan in één source file één of meerdere classes definiëren. Als je per class een source file maakt, is het gemakkelijker om de source van die ene class op een andere PC te gebruiken.

* Kies in het menu *Project* de opdracht *Add* *Class*… of klik in de *Solution* *Explorer* met de rechtermuisknop op de naam van het project en kies *Add* – *Class*…
* Kies in het midden van het dialoogvenster voor *Class*.
* Tik bij *Name*: de naam van de nieuwe class: *Werknemer*.
* Klik op de knop *Add*.
* C# maakt een nieuwe source file aan (**Werknemer.cs**)

met de *namespaces System*... kan men classes zoals Console aanspreken zonder hun volledige namespacenaam te schrijven. De class *Werknemer* is ingekapseld in de namespace *CSharpPFCursus*. Zo kan je in dezelfde applicatie nog een andere class Werknemer creëren (met een andere betekenis), zolang ze maar in een andere namespace is ingekapseld.

**Sleutelwoorden** zijn dezelfde als in het hoofdstuk ENUM. Alleen is de class ***internal*** wanneer er geen sleutelwoord wordt gegeven. Dit betekent dat je de class enkel kan gebruiken in de **assembly** waarin je de class **declareert**. Wil je deze ook buiten de assembly gebruiken moet je *public* gebruiken.

## Property toevoegen

Per property voorzie je in de class **3 onderdelen**:

* **private** variabelen

Een **private** variabele in een property zorgt er onder andere voor dat een programmeur wel de waarde van deze variabele kan oproepen maar **niet** kan **veranderen**, wat bij een **public** variabele wel zou zijn en waarbij we de invoer niet zouden kunnen controleren. Voor de property InDienst maak je bv. een **private** DateTime variabele *inDienstValue*.

* **setter**

Een **setter** is een **beveiligde toegangsweg** waarmee een programmeur de waarde van een property kan **invullen**. Deze code controleert onder andere of de nieuwe waarde een geldige waarde is. C# roept de ***set*** **automatisch** op als een programmeur de property van een werknemer object invult:

*ik.InDienst = new DateTime(2018,1,1);*

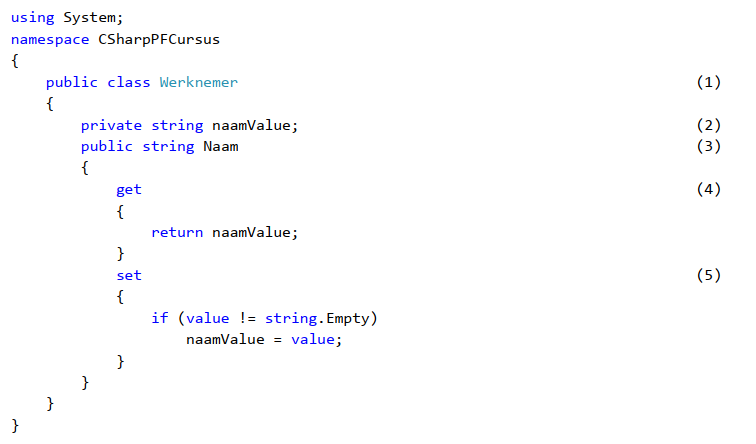
De set krijgt de in te vullen waarde voor de property binnen via het sleutelwoord **value**. In de set kan je deze waarde controleren voor je ze invult in de **private** variabele *inDienstValue*.

* **getter**

Een **getter** is een toegansweg waarmee een programmeur de waarde van de property kan opvragen. In de get haal je de waarde op uit de private variabele en geef je deze waarde aan de buitenwereld met de return opdracht. C# roept deze routine automatisch op als je de property van een *Werknemer* object opvraagt.

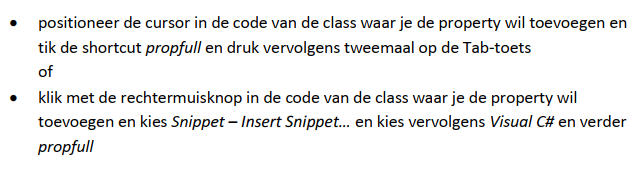
*Console.WriteLine(ik.InDienst);*

Op zich zijn de **set** en **get** in de source van je class genest in een **andere** structuur. Deze structuur noemt men ***property***. Deze begint met het **sleutelwoord** van het toegangsniveau, gevolgd door het **type** van de property en dan de **naam** van property.



1. Het toegangsniveau van de class declareer je als **public.**
2. Ieder object van het type *Werknemer* heeft in zijn intern geheugen zijn eigen variabele *naamValue*.
3. De **beschrijving** van de property *Naam*. Deze property krijgt als **type** string. Normaal kies je hetzelfde type als de private varibele die de waarde van de property onthoudt (*naamValue*).
4. Een **get** routine. C# roept dan de **routine *get*** op als een programmeur de property *Naam* opvraagt van een bepaald object. bv *ik.Naam*
5. Een **set** routine**.** Zo kan een programmeur de property *Naam* van een werknemer object een waarde geven. bv *ik.Naam=”Asterix”*

Er zijn 2 **shortcuts** om op een snelle manier een nieuwe property toe te voegen:



Een **nieuwe** *source file* (bv. een nieuwe enum voor Geslacht) wordt toegevoegd via *Add new item*, *Code File*. De naam van het bronbestand wordt met een **hoofdletter** geschreven. Daar typ je:

namespace CSharpPFCursus

{

public enum Geslacht

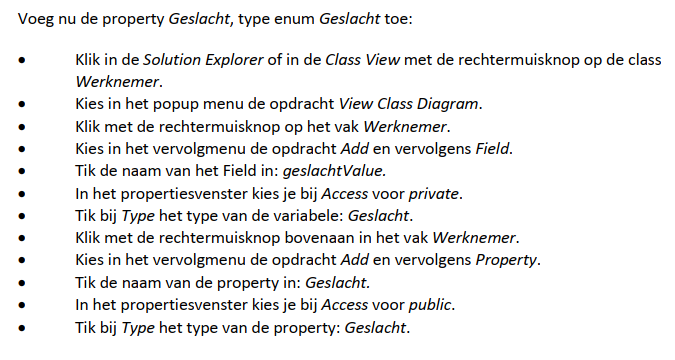
{

Man, Vrouw

}

}

Nu gaan we de code van de property niet zelf typen maar gebruiken we een **Class Diagram**, een grafische voorstelling van de class. Dit moet wel eerst geïnstalleerd worden.



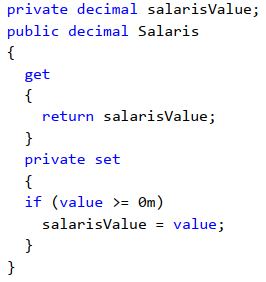
## Auto properties

Wanneer de get/set routines geen extra code moeten bevatten kan je de auto properties zoals voorgesteld gebruiken.

! De *private variabele* die bij deze properties horen, **zie je niet** !

## Mixed access levels

Tot hiertoe heb je properties altijd het access level *public* gegeven. Dit access level geldt dan voor zowel de set- al de get-method van de property. Je kan echter deze 2 methods een verschillend access level geven. Je laat bijvoorbeeld toe iedereen een **property** *salaris* te lezen (get) maar de set-routine mag enkel door een **method uit de class zelf** uitgevoerd worden. Dan geef je deze routine een ***private***access level als volgt:

Wens je deze set-method ook in een **afgeleide class** te gebruiken, dan kan je ***protected*** gebruiken.

Overriden: wil je properties in een afgeleide class overriden, dan moet je deze in de afgeleide class **dezelfde** access levels geven als in de moederklasse.

## Reference variabelen

Je kan variabelen declareren met als **type** classes. Deze zijn **reference variabelen** of **object variabelen**.

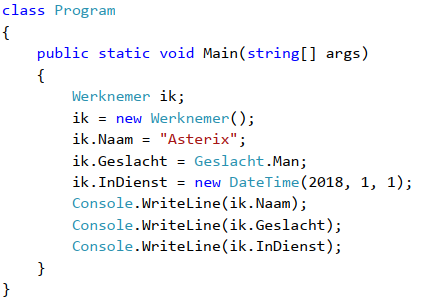
*Werknemer ik;*

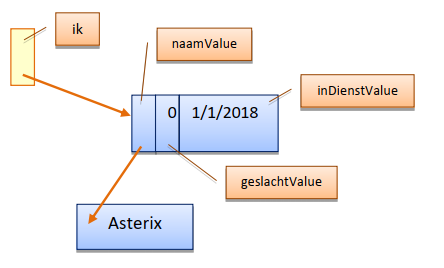
*ik = new Werknemer();*

Eerst declareer je een reference variabele *ik* die als **type** de class Werknemer heeft. Pas daarna maak je een nieuw Werknemer object met het sleutelwoord **new**. (dit is een **constructor**)

Nu kan je via de **object variabele** *ik* de properties van het object invullen (setter):

*ik.Naam = “Asterix”;*



De variabele *naamValue* is van het type string en is dus op zich ook een *reference variabele* naar de tekst Asterix in het geheugen.

## null

Je kan een reference variabele die naar een object verwijst, terug “nergens naar” laten verwijzen, door de waarde **null** in de reference variabele in te vullen. Een reference variabele die enkel een type heeft meegekregen en geen waarde, verwijst nog nergens naar. (bijvoorbeeld *Werknemer ik;*)

Dit kan je oplossen door:

*Werknemer ik = new Werknemer();*

2 object variabelen kunnen ook verwijzen naar hetzelfde object. (*Werknemer* *mezelf = ik*).

## Een array van objecten (references)

Het type van een array kan ook een class zijn, om een verzameling gelijkaardige objecten voor te stellen.

*Werknemer[] onzeWerknemers = new Werknemer[20];*

! MAAR ieder element van de array is slechts een reference, ze bevatten zelf geen Werknemer object. De inhoud van ieder element is *null*, ze verwijzen nog niet naar Werknemer objecten. Je moet ze ieder apart ook nog een verwijzing geven:

*for (int teller = 0; teller < onzeWerknemers.Length; teller++) onzeWerknemers[teller] = new Werknemer();*

## Garbage collection

Garbage collection is een techniek om het intern geheugen terug vrij te maken van objecten die niet meer in gebruik zijn. C# voert automatisch regelmatig garbage collection uit. Een object leeft totdat geen enkele reference variabele er nog naar verwijst. C# zal deze verwijderen wanneer er te weinig intern geheugen vrij is.

## ReadOnly en WriteOnly properties

Een ReadOnly property is een property die een programmeur enkel kan lezen maar niet kan invullen of wijzigen. Je maakt een ReadOnly property door de property enkel te voorzien van een **get** gedeelte, **geen set** gedeelte.

Een WriteOnly property is een property die een programmeur enkel kan invullen of wijzigen, maar niet kan lezen. Je maakt een WriteOnly property door de property enkel te voorzien van een **set** gedeelte, **geen get** gedeelte.

Je kan ook een *auto property* ***readonly*** maken door de *set* te voorzien van het sleutelwoord *private:*

*public int PersoneelsNummer { get; private set; }*

**Deze method kan enkel door een method *binnen* de class of via een *constructor* ingevuld en gewijzigd worden.**

public int *PersoneelsNummer* { get; }

**Deze method kan enkel via een *constructor* ingevuld worden, maar *niet* door een method binnen of buiten de class.**

## Constructors

Behalve wanneer een class *static* is, krijgt iedere class automatisch een **default constructor** (een constructor zonder argumenten). Zo creëert men een leeg object. Men kan zelf ook een **constructor** aanmaken door een *public method* te maken met dezelfde naam als de *class* zelf:

*public Rekening(string nummer){ Rekening = nummer; }*

Het sleutelwoord ***this*** kan gebruikt worden om een andere constructor op te roepen als volgt:

*public Rekening() { : this(“0”) }*

## Auto-ID

static int lastId = 0;

static int generateId()

{

return Interlocked.Increment(ref lastId);

}

## Default value

public int X { get; set; } = x;

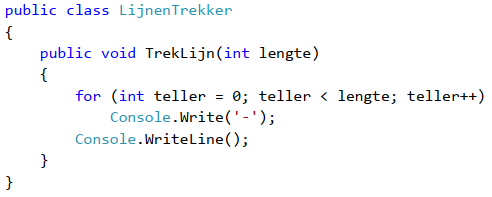
# Methods

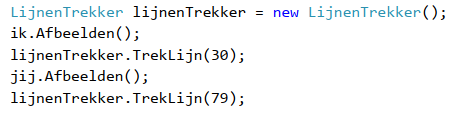
Je begint de declaratie van een method met een sleutelwoord dat het toegangsniveau van de method aangeeft (**public, private, protected, internal, internal protected)**. Wanneer je geen toegangsniveau opgeeft past C# **private** toe. Als de method **geen waarde teruggeeft** gebruikt men het woord **void**. Daarna volgt de naam van de method.

Microsoft stelt voor een method met een **hoofdletter** te starten.

In tegenstelling tot properties van verschillende objecten van 1 class, worden *methods* maar 1 keer en op maar 1 plaats opgeslagen in het geheugen.

## Parameters





Je kan ook parameters invullen adhv **named parameters**, deze kunnen handig zijn wanneer de method veel optionele parameters heeft. Dan geef je gewoon de naam met een dubbelpunt mee aan de functie:

*lijnenTrekker.TrekLijn(****teken:*** *‘\*’,* ***lengte:*** *10);*

Je maakt een **optionele parameter** aan door de parameter een defaultwaarde te geven:

*public void TrekLijn(int lengte, char teken = ‘-‘) { }*

* Bij de **declaratie** van de method moeten de niet-optionele parameters altijd als eerste vermeld worden.
* Bij optionele parameters kunnen de sleutelwoorden *ref* of *out* niet voorkomen.

## Overloading

*Overloading* betekent dat je van een **method meerdere versies** maakt.

C# aanvaardt overloading als een method een verschillend aantal parameters heeft of minstens één van de parameters een verschillend type heeft.

## Sleutelwoord *ref*

Standaard roept C# methods op ***by value***. De waarden van de variabelen zullen dus niet veranderen wanneer deze worden aangepast in de method. Bij ***ref*** daarentegen bevat de parameter de **verwijzing** naar de originele variabele. Hiervoor moet je zowel bij het maken van de method als bij het oproepen ervan de parameters laten voorafgaan door *ref*.

*public void Verwissel(****ref*** *int getal1,* ***ref*** *int getal2){ } verwisselaar.Verwissel(ref eerste, ref, tweede);*

## Sleutelwoord *out*

Het sleutelwoord *out* werkt op dezelfde manier als *ref*, alleen hoeft bij *out* de parameter nog geen waarde te hebben. Je kan dus in dezelfde lijn nog variabelen declareren:

*int eerste = 10, tweede = 20; Verwisselaar.VerwisselNaarAndereVariabelen(eerste, tweede,* ***out int resultaat1****,* ***out int resultaat2****);*

De scope van *out* parameters is niet beperkt tot het codeblok. Deze variabelen kunnen in het omringende codeblok aangesproken worden. de *TryParse()* method maakt gebruik van *out* parameters.

## Return

Wanneer een method een resultaatwaarde teruggeeft gedraagt deze zich als een functie. Een method kan maximaal één resultaatwaarde geven. Dit kan een *Value* type zijn of een *reference* (zoals een string of werknemer).

## TryParse()

Iedere **value** type heeft een method **Parse()** waarmee je een string kan omzetten naar een value type. TryParse() is een meer voorzichtige methode, indien het lukt is het resultaat *true* en wordt de omgezette waarde in de tweede parameter geplaatst. Je hebt hierbij *out* of *ref* nodig omdat anders een value type wordt doorgegeven en geen reference dus wordt de waarde niet aangepast.

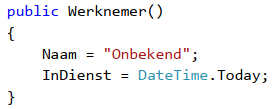
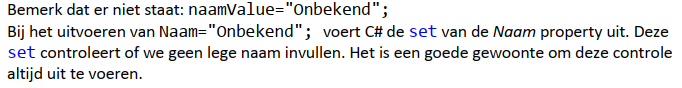
# Constructors

Een constructor is een method in een class, die bij de **creatie van een object** van die class uitgevoerd wordt en eventueel de properties van dat object op een beginwaarde plaatst. Bij uitgebreide classes kan een constructor ook extra initialisatiewerk doen:

* Een bestand openen waaruit een object gegevens leest
* Een tijdelijk bestand aanmaken waarin het object tijdelijks gegevens bewaart

Maar een class kan **meerdere constructors** bevatten. Je maakt een constructor door in de class een method aan te maken met **dezelfde** **naam** als de class. Standaard maakt Visual Studio geen constructor aan voor een class. De C# compiler maakt zelf een constructor aan, die geen code bevat.

|  |  |
| --- | --- |
| type | beginwaarde |
| getal (int, double, decimal,...) | nul |
| char | teken met Unicode waarde 0 |
| enum | eerst beschreven constante in de enum (bij de enum *Geslacht* is dit *Man*) |
| bool | false |
| DateTime | 1/1/1 0:0:0 |
| class (bvb. string, Werknemer, ...) | null |

Een **default constructor** is een constructor die **geen parameters** heeft.

Als je nu een Werknemer object aanmaakt, bevat dit Werknemer object reeds meer logische beginwaarden. Zoals bijvoorbeeld naam als “onbekend” en datum als *DateTime.Today*.

Je kan zelf ook constructors aanmaken **met parameters** als volgt:

*public Werknemer(string naam, DateTime inDienst, Geslacht geslacht) { Naam = naam; InDienst = inDienst; Geslacht = geslacht; }*

Vanaf dat je één constructor aanmaakt maakt de C# compiler geen default compiler meer aan. Zo kan je iemand verplichten andere constructor te gebruiken en kan de programmeur geen ‘anoniem’ object’ meer aanmaken.

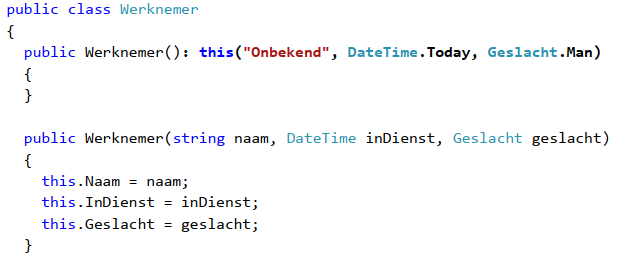
## this

Het sleutelwoord *this* staat voor een **reference** naar het **huidige object**. Als je binnen class code *this* gebruikt, verwijs je naar het object waarvoor de class code op dat moment van toepassing is. Via *this* kan je dus de **properties** en **methods** van het huidige object **aanspreken**.

Als een parameter of lokale variabele van een method dezelfde naam heeft als een variabele of property van de class waartoe deze method behoort, kan je het sleutelwoord ***this*** ook gebruiken om de parameter of lokale variabele te **onderscheiden** van de **class** variabele of class property. *this* verwijst dan nog steeds naar het object.

## Constructor chaining

Bij constructor chaining roep je vanuit een constructor een andere constructor op, om van zijn diensten gebruik te maken. Je roept de andere constructor op met ***this(parameters)***.



Constructor chaining is een techniek die ook bij *Inheritance* toegepast wordt.

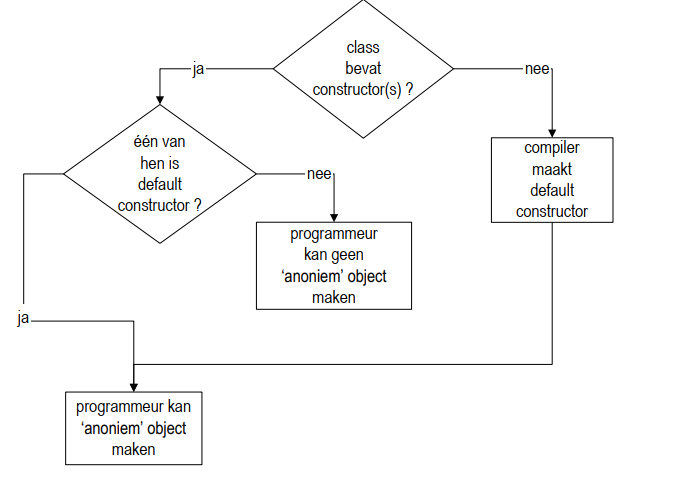
## Destructor

Een destructor is een method van de class die C# automatisch oproept juist vóór de garbage collector het geheugen van het object opkuist. De garbage collector verwijdert een object uit het geheugen als er geen enkele reference variabele meer naar dit object verwijst.

Je maakt een destructor door een method aan de class toe te voegen met als naam het ~ teken, gevolgd door de naam van de class (in de class Werknemer maak je bvb. een method met als naam

**~Werknemer()**).

Je moet enkel een destructor in je class voorzien als objecten van deze class externe bronnen (bestanden, internetverbindingen, ...) gebruiken, die je terug wil vrijgeven als het object niet meer gebruikt wordt. Als je bvb. een class BeursAandeel maakt, kan je in de constructor van deze class een internetverbinding openen om de huidige koers van een aandeel op het internet op te zoeken. In de destructor sluit je de internetverbinding. Zo ben je zeker dat C# de internetverbinding automatisch sluit voor het object opgekuist wordt.



# Static members

## Static variabelen

**Static** variabelen zijn class variabelen die door alle objecten van deze class gedeeld worden (variabelen met class bereik).

**Gewone** **class** variabelen krijgen voor elk object van die class een eigen versie in het intern geheugen: ieder object heeft zijn eigen waarde voor deze variabelen. **Static** **class** variabelen komen slechts **één** keer in het **geheugen** voor. Alle objecten van de class delen deze variabelen en hebben dezelfde waarde voor deze variabelen.

Je definieert static class variabelen met het sleutelwoord static. Ook bij properties en class methods kan je het woord static tikken. Deze properties en methods kan je dan **enkel** **toepassen** via de **class** zelf, niet op de individuele objecten van de class (properties en methods met class bereik). In de code van static properties en static methods kan je **enkel** **de** **static** **class** **variabelen** **aanspreken**, niet de gewone class variabelen (*naamValue*, *inDienstValue*, *geslachtValue*). Je kan ook niet verwijzen naar één individueel object van de class, dus ook **niet** naar **this** (het huidige object).

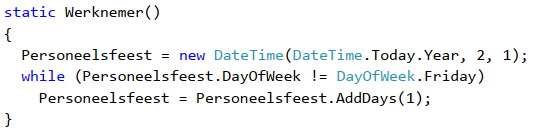
Voorbeeld: je voegt aan de class *Werknemer* een property *Personeelsfeest* toe. Deze property bevat de datum van het personeelsfeest dat jaarlijks doorgaat voor alle werknemers. Gezien deze datum voor alle werknemers dezelfde is, definieer je de variabele die deze datum bijhoudt static. C# houdt deze variabele slechts één keer in het intern geheugen bij en niet per Werknemer object.

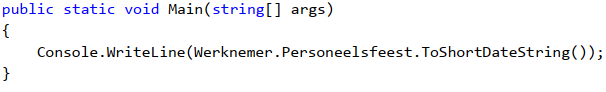
*public static DateTime Personeelsfeest { get; set; }*

*Werknemer.Personeelsfeest = new DateTime(2018, 12, 12);*

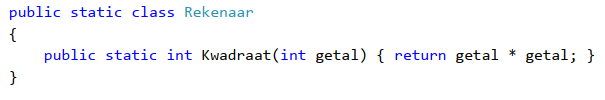
*Console.WriteLine(Werknemer.Personeelsfeest);*

## Static constructors

C# voert de static constructor uit **vóór** je de class gebruikt!

De property *Personeelsfeest* krijgt nu via de **static constructor** een beginwaarde:

## Static classes

Je kan ook een **class** static maken. Je doet dit door het sleutelwoord *static* voor het sleutelwoord *class* te plaatsen. Een static class kan enkel static methods bevatten. Je kan dan deze static methods oproepen zonder eerst een instance te maken.

# Inheritance

**Inheritance** betekent dat je een class maakt op basis van een bestaande class. De nieuwe class erft de **properties** en **methods** van de basis klasse. Op die manier maak je vlot nieuwe classes die op bestaande classes lijken. Je kan:

1. aan de nieuwe class properties en methods toevoegen
2. properties en methods uit de basis klasse in de nieuwe class overschrijven (**overriding**)

De class waarop je de nieuwe class baseert heet de **base class**.

De nieuwe class heet de **derived class**.

In C# kan een derived class slechts **één base class** hebben. Dit heet **single inheritance**. Die base class kan op zich terug één base class hebben (In C++ kan een derived class meerdere base classes hebben, multiple inheritance).

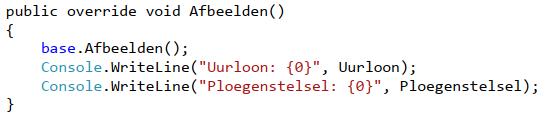
Op dit moment kunnen de default constructors nog voor compilerfouten zorgen:

De class *Arbeider* erft van de class *Werknemer*.

Wat in C# wel mag is dat je met een **variabele** van de **base class** verwijst naar een object van een **derived class**.

## Overriding

Bij overriding overschrijf je properties en/of methods van de base class in de derived class.

**Eerst** moet je in de base class **toelaten** dat een derived class de method mag overschrijven. Je voegt daartoe het sleutelwoord ***virtual*** toe aan deze method. Nu kan je in de derived class een nieuwe method *Afbeelden()* toevoegen door het sleutelwoord ***override*** te schrijven.

Het sleutelwoord ***base.*** laat toe in de nieuwe method de oude method nog op te roepen. Als je wilt verhinderen dat een afgeleide class van *Arbeider* op zijn beurt de method *Afbeelden()* kan overschrijven, kan je het sleutelwoord ***sealed*** toevoegen aan de method *Afbeelden()* in de class *Arbeider*.

## Constructors en inheritance

**De volgende regels** bepalen het samenspel tussen inheritance en constructors:

* Een derived class **erft** de **constructors** van de base class **niet**.
* Als de **derived** class **geen** **constructors** heeft, maakt de **compiler** zelf een weliswaar onzichtbare **default** **constructor** aan, die op zijn beurt de **default** **constructor** **van** **de** **base** **class** oproept.
* Als de derived class een constructor heeft waarin je **niet** expliciet één van de **constructors** van de base class **oproept**, roept de compiler **eerst** **de** **default** **constructor** **van de base class**

op, en voert daarna de code uit van de constructor van de derived class.

De class *Arbeider* heeft geen default constructor. De compiler maakt een onzichtbare default constructor aan, die de default constructor van de class Werknemer oproept. Als de base class geen default constructor heeft, krijg je een fout.

**Constructor chaining:**

Je kan in een derived class vanaf nu op twee manieren constructor chaining (een constructor oproepen vanuit een constructor) toepassen:

* Je kan een constructor uit de eigen class oproepen **this(***parameters****)***
* Je kan een constructor uit de base class oproepen **base(**parameters**)**

*public Bediende(string naam, DateTime indienst, Geslacht geslacht, decimal wedde): base(naam, indienst, geslacht)*

*{*

*Wedde = wedde;*

*}*

## De base class Object

Een class die niet erft van een andere class krijgt toch een base class, namelijk de class *Object*, die in .NET ingebakken is. Het sleutelwoord ***object*** is een synoniem voor de **class *Object*.** De class *Object*bevat enkele methods die je in je eigen classes beter overschrijft.

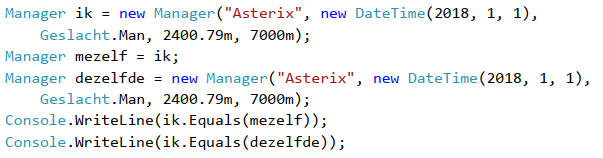
### ToString() method

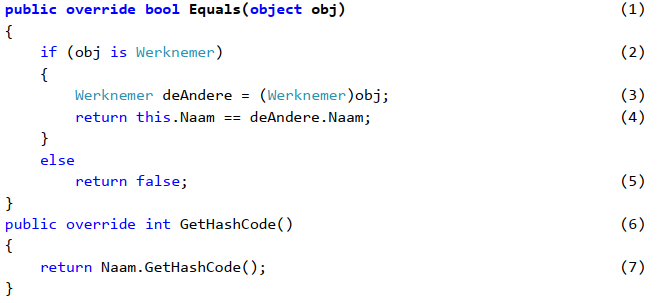
Deze method heeft als doel het object in de vorm van een string terug te geven. Deze method geeft in de **class** ***Object***enkel de naam van de **namespace** waaring de class gedefinieerd is, terug, gevolgd door de **naam.** *--> CSharpPFCursus.Manager*

Je kan de method *ToString()* in je eigen classes **overschrijven** zodat deze bijvoorbeeld de belangrijkste properties van het object als string teruggeeft.

De ***WriteLine()*** method van de class *Console* aanvaardt ook een **object** als **parameter** en voert dan zelf de method *ToString()* uit. Console.WriteLine(ik);

### Equals

De default method van de class *Object* heeft een beperkte betekenis. Deze returnt *True* als de 2 variabelen naar hetzelfde **verwijzen**.

Je kan deze method dus *overriden* zodat deze bepaalde waarden van de 2 objecten gaat vergelijken.

1. De method *equals()* aanvaardt een parameter van het type *Object*.
2. Om zeker te zijn dat het wel een object van de class *Werknemer* is wordt hier gecontroleerd met het sleutelwoord *is*. Een ander soort object zou zinloos zijn.
3. Je zet het object van de class *Object* nu om in één van de class *Werknemer* door deze te casten.
4. Je vergelijkt nu de namen van de 2 objecten (of id-nummer)
5. Wanneer je de method *Equals()* *override* moet je ook de method *GetHashCode()* overschrijven. Anders geeft dit een compiler warning. Deze method geeft het getal van de objecten in de hashtables terug. Gelukkig is deze al overschreven in de class *String*, Dus kunnen we deze gebruiken via de string *Naam.*

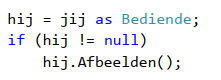
### *is*

Met het sleutelwoord *is* kan je controleren of een **object** waar een **reference variabele** naar verwijst, tot de opgegeven **class** of **afgeleide class** behoort.

*object is class*

### *as*

Met het sleutelwoord *as* kan je eveneens een **typeconversie** of **typecasting** doen. We hebben dit al eerder gedaan met ronde haakjes (), MAAR het grote verschil is dat bij de conversie met *as* **geen fout** optreedt wanneer de conversie niet mogelijk is. Als de conversie mislukt wordt **null** teruggegeven, wat je als programmeur kan opvangen.

! Je mag in C# een **variabele** van een **base class** gebruiken om een object van een **derived class** op te slagen. Hierna moet je deze nog wel typecasten mocht je deze willen vergelijken met een derived class.

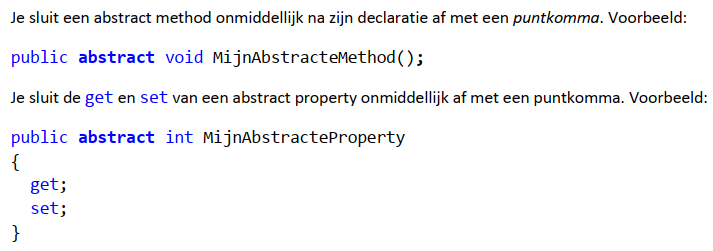
Als deze conversie **niet** lukt, krijgt de variabele *hij* de waarde *null*.

# Abstract classes en sealed classes, abstract methods

Een **abstract class** is een class waarvan je **geen object** kan maken. Je kan **wel** van zijn **derived classes** objecten maken. Je kan echter wel reference variabelen aanmaken met als type een abstract class die dan verwijzen naar een derived class. Dit is handig bij *polymophism*. Dit ziet er als volgt uit:

Een **sealed class** is een class waarvan je geen andere classes meer kan afleiden. Dit doe je door het sleutelwoord ***sealed*** te vermelden bij de class.

**Abstract members** zijn properties en/of methods die je in een base class declareert (de naam, het type en eventuele parameters), maar **niet** in de base class **uitwerkt**. Je **verplicht afgeleide classes** de methods uit te werken. Als je in een class een abstract method maakt, moet de class zelf ook abstract zijn. Je moet bij die class dus ook het sleutelwoord ***abstract*** gebruiken!

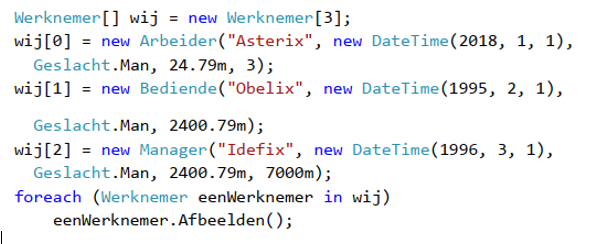


Een method met het sleutelwoord ***abstract*** is **automatisch** ook ***virtual***. (niet ***readonly***)

# Inheritance polymorphism

Bij **inheritance polymorphism** reageert ieder object in een verzameling gelijkaardige objecten op zijn eigen manier als je object per object van de verzameling vraagt eenzelfde method uit te voeren of eenzelfde property in te lezen of te tonen. De methods en properties die je voor ieder element kan gebruiken, zijn de methods en properties van **de gemeenschappelijke base class**.

Als je voor een element van de array echter een method uitvoert of een property toont of inleest doet **C# het volgende**:

* Als je de method of property in de derived class **overschreven** hebt, roept C# de method of property van de **derived class** op
* Als je de method of property in de derived class **niet overschreven** hebt, roept C# de method of property uit de **base class** op

# Aggregation

Bij **aggregation** gebruik je binnen een class variabelen die als **type** een **class** hebben. Deze variabelen zijn dus reference variabelen naar objecten.

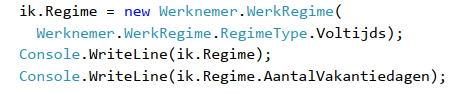
Als voorbeeld heb je een Afdeling als property bij Werknemers. Het betreft hier een **reference variabele**. Je zal in het hoofdprogramma *Afdeling* objecten aanmaken en *Werknemer* objecten naar die Afdeling objecten laten verwijzen. Dit is een voorstelling uit de echte wereld.

# Nested en partial classes

Een ***nested class*** – ook wel ***inner class*** genoemd – is een class die gedefinieerd is binnen een andere class. Je gebruikt een nested class om een class te definiëren die verband houdt met de class waarbinnen ze gedefinieerd is, en die zonder de class waarbinnen ze gedefinieerd is, geen betekenis heeft.

Bij nested classes gebruikt men het woord ***top-level class***. Dit is een class die een nested class bevat maar zelf geen nested class is.

|  |  |
| --- | --- |
| public | overal bruikbaar. |
| private | enkel bruikbaar in de class waarin de class genest is. |
| protected | enkel bruikbaar in de class waarin de class genest is en in afgeleide classes. |
| internal | enkel bruikbaar in de assembly (het project) waarin de class gedeclareerd is. |
| protected internal | enkel bruikbaar in de class waarin de class genest is, in afgeleide classes en in de assembly waarin de class gedeclareerd is. |

Een hele code , zie *Werknemer.cs*. Een object van de class Arbeider krijgt op de volgende manier een regime toegeëigend:

Als je de source van de class *Werknemer* uit de vorige paragraaf bekijkt, dan merk je dat deze source snel vrij onoverzichtelijk wordt. De definitie van de class *Werknemer* kan je in meerdere stukken splitsen (letterlijk). Die deelbeschrijvingen van de class kan je desgewenst over meerdere source files spreiden:

**Partial classes**:

*public* ***partial*** *class Werknemer { ..eerste stuk code van class Werknemer..}*

*public* ***partial*** *class Werknemer { ..tweede stuk code van class Werknemer..}*

Zo zou je de code voor de inner class Werkregime uit de classbeschrijving van *Werknemer* kunnen halen. Je zou deze zelfs in een aparte source file kunnen onderbrengen. De naam van de **class**Werknemer bij het woord **partial** geeft aan dat het source files van dezelfde class *Werknemer* zijn.

# Interfaces

In een **interface** definieer je een ***contract*.** Dit contract is een **verzameling** van property en/of method **declaraties.** In de interface schrijf je **enkel** **de declaratie** (naam, parameters, return type) van de properties en methods. Je schrijft geen code. Wanneer een class de code voor een interface bevat spreekt men over *interface implementatie*.

* een class kan meerdere interfaces implementeren
* meerdere classes kunnen dezelfde interface implementeren
* Interfaces brengen classes met elkaar in verband die geen gemeenschappelijke base class hebben, maar wel gemeenschappelijk gedrag.
* Interfaces kunnen een beperkt beeld geven van een uitgebreide class. Een klant van een bank bekijkt de class Bank anders dan een bediende van een bank.

Een class die een interface implementeert, moet deze **volledig** implementeren: alle methods en properties van de interface moeten ook terug te vinden zijn in de class.

Een interface kan ook erven van één of meerdere base interfaces. Deze erft dan alle methods en properties.

Microsoft raadt aan de naam van een interface met een hoofdletter i (I) te starten.

## Een interface ontwerpen

Een **interface** plaats je standaard ook in een ***namespace***. Deze is default ***internal***. Bij de properties en methods van een interface kan je **geen** **toegangsniveau** vermelden. De interface zelf kan je **wel** een **toegangsniveau** geven. In de *derived classes* moet je de interface niet meer implementeren.

Je moet in de class code steeds alle members van de interface vermelden. Wanneer je deze toch niet nodig hebt maar wel in een *derived class*, maak je hier een ***abstract*** method van.

*public abstract class Werknemer: IKost*

Nadat je de interface implementeert voegt VS de 2 classes toe, maar volgens een ***expression-bodied syntax***. ipv de *get* van een *readonly property* uit te schrijven, gebruiken ze een lambda notatie: =>

*public decimal Bedrag => throw new NotImplementedException();*

*public bool Menselijk => throw new NotImplementedException();*

Verander deze (later hierover meer) en maak *Bedrag* abstract.

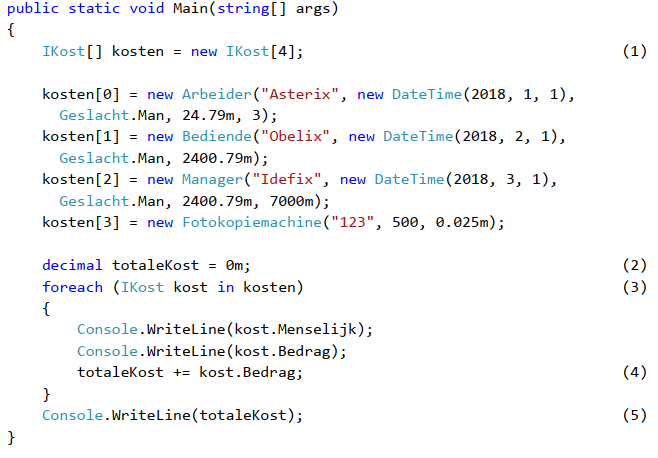
**Een base class en meerdere interfaces**:

Je plaatst steeds de **base class eerst**.

## Interface polymorphism

Zoals bij **inheritance polymorphism** reageert bij interface polymorphism **ieder object** in een verzameling gelijkaardige objecten op zijn **eigen manier** als je object per object van de verzameling vraagt eenzelfde **method** uit te voeren of eenzelfde **property** in te lezen of te tonen.

* De array met references naar de gelijkaardige objecten moet als **type** de **gemeenschappelijke interface** hebben. In ons voorbeeld IKost.
* Je kan enkel de methods en properties van de gemeenschappelijke interface gebruiken.
* Dit kan handig zijn om bv. een property van alle gelijkaardige objecten op te tellen.

1. Je maakt een **array** van het type *IKost*. Daarna vul je deze op met **verwijzingen** naar objecten van classes die deze interface implementeren.

Je kan **geen** objecten **instantiëren** van het type van een interface. *Ikost kost = new IKost()*

Je kan wel **reference variabelen** declareren met als type het type van een interface. Deze reference variabelen (bv. verzameld in een array) kan je dan laten verwijzen naar objecten met als type een class die deze interface implementeert. *Ikost kost = new Fotokopiemachine(“123”, 1000, 10m);*

1. Je doorloopt alle elementen van de array en toont de properties *Menselijk* en *Bedrag* van elk element. Ieder object heeft zijn eigen code voor de property *Bedrag*.

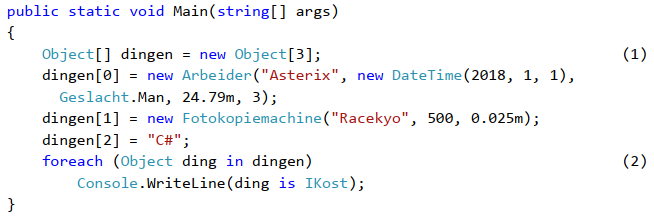
**Interfaces spelen een belangrijke rol bij het ontwerpen en het ontwikkelen van objectgeoriënteerde applicaties.** Programmeren tov interfaces zorgt er voor dat grote applicaties eenvoudiger te onderhouden, uit te breiden en te testen zijn. Een goede guideline zijn de **SOLID design principles**:

* **S**ingle-responsibility principle: A class should have one and only one responsibility, meaning one job.
* **O**pen-closed principle: A class should be easily extendable without modifying the class itself.
* **L**iskov substitution principle: Every derived class should be substitutable for their base class. Meaning they should be used in the same way.
* **I**nterface segregation principle: A client should never be forced to implement an interface or (abstract) method they do not need.
* **D**ependency Inversion principle: A higher class must never depend on the lower class

## *is*

Met het sleutelwoord ***is*** controleer je of een object tot een class (of een afgeleide class) behoort die een interface (of een afgeleide interface) implementeert. Het resultaat van deze expressie is *tru*e of *false.*

*object is interface*



Met een array van het type *Object*kan je dus verwijzen naar alle afgeleide classes van *Object* (de moeder der alle classes). Het laatste object zal *False* returnen omdat deze de interface *IKost* niet implementeert.

# Tuples

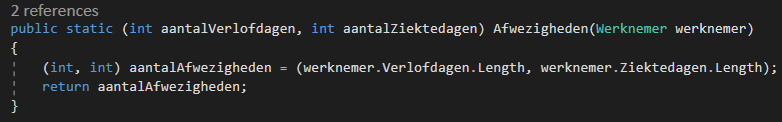
## Meerdere resultaatwaarden

Een method kan normaal één resultaatwaarde hebben. De resultaatwaarde is dan een *Value type* of *Reference type*. Wanneer de resultaatwaarde van een method **meerdere** **gegevens** bevat, dan definieer je hiervoor meestal een **class** die als **resultaatwaarde** wordt gebruikt. **Of** je maakt gebruik van ***out*****parameters**. Maar C# voorziet hiervoor ook ***tuples***.

Een **tuple** is een **datastructuur** met een bepaald **aantal** **elementen** van een **bepaald** **type** in een bepaalde volgorde. Tuples die binnen een method gedeclareerd worden, kunnen ook buiten deze method aangesproken worden.

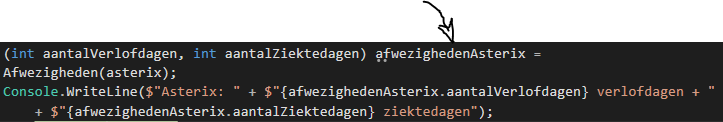
* Tuples zijn ***Value******types***, afgeleid van *System.ValueType*.
* Het **type** **én** de **volgorde** van de elementen van een tuple **bepalen** het **type**.

Een gewone method in Program.cs:



Deze resultaatwaarden vallen te ontvangen met:

array van datums



*afwezighedenAsterix* is hier een variabele gedeclareerd als een *tuple*-type met 2 benoemde elementen van het type *int*. Je kan ook gewoon *var afwezighedenAsterix* gebruiken. Je verwijst naar de elementen van de tuple via *variabele.naamtuple-element*. *afwezighedenAsterix* en *aantalAfwezigheden* zijn Tuples.

## Intermezzo: class vs struct

Je definieert een **struct** op dezelfde manier als een class. Net zoals een class is een struct een **template** voor een gegevenstype. Net zoals bij een class kan je in een struct eigenschappen en handelingen/acties definiëren dmv respectievelijk **properties** en **methods**. Net zoals bij een class kan je op basis van een struct **objecten** of **instanties** creëren. **Inheritance** is bij een struct echter **niet** mogelijk.

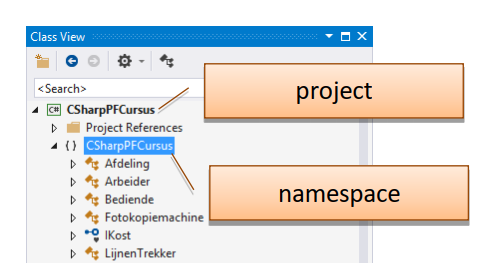
Het grote verschil zit men in het soort variabele. Een **class** is een ***Reference******type***. Wat wil zeggen dat wanneer 2 variabelen naar hetzelfde object verwijzen, het object kan aangepast worden door de beide. Een **struct** is een ***Value******type***. Voorbeelden hiervan zijn ***DateTime****,* ***Byte****,* ***Int32****,* ***Double****,* ***Decimal*** *...*

# Namespaces

Een **namespace** is de naam voor een verzameling van bij elkaar horende types (*classes, interfaces, enums*). Namespaces zijn dus interessant om naamconflicten te vermijden wanneer classes of interfaces dezelfde naam hebben.

Namespaces hebben geen verband met de toegankelijkheid van classes (*public, private ...*)

**Geneste** **namespaces** kunnen helpen om verschillende afdelingen te onderscheiden.

Zo’n volledige naam heet een *fully qualified name*. De conventie is dat alle woorden starten met een **hoofdletter**.

Om niet steeds de volledige naam te moeten gebruiken om te verwijzen naar een class, kan men best het sleutelwoord ***using*** gebruiken. Als je een **geneste namespace** wil importeren, moet je die expliciet **apart** **importeren**, zelfs als je zijn omsluitende namespace reeds geïmporteerd hebt.

Met ***using static*** kan je specifieke classes importeren. Zo hoef je niet de volledige namespace te importeren; *using static System.Console;*

Je kan ook een alias geven aan een namespace om typwerk te beperken. Dit doe je door de namespace te laten voorafgaan door een alias en een =. *using MateriaalStatus = Firma.Materiaal.Status;*

p360: Om sommige namespaces te kunnen gebruiken, moet je eerst een reference leggen naar de DLL-file. *Project – Add Reference*

# Delegates en events

We hebben al variabelen gemaakt die een *reference variabele* van het type *Arbeider* zijn. We hebben ook al variabelen gemaakt die als type een *interface* hebben en *reference variabelen* zijn naar classes met die interface. Maar een *reference variabele* kan nu ook van het type ***delegate*** zijn.

Een **delegate** is een type waarmee je kan verwijzen naar een **method**. Dit kan een method van een object zijn, of een static method van een class, of een anonymous method of een lambda expressie. Delegates helpen om

* Via een reference variabele van het type delegate kan een gekoppelde method uitgevoerd worden.
* Delegates worden ook gebruikt om methods als parameters door te geven aan andere methods. De parameters die de methods ontvangen, hebben dan als type een delegate. De delegate geeft in dit geval enkel aan hoe de method die wordt meegegeven er zou moeten uitzien.
* Delegates vormen ook de basis voor **events**. Een event heeft als type een delegate. Je kan aan een event enkel methods koppelen die voldoen aan de signatuur van de delegate op basis waarvan het event gedeclareerd is.

## Werkwijze

Bij de **definitie** van de delegate leg je de **signatuur** van de delegate vast: Je bepaalt zowel het *returntype* als de *parameters* *(aantal, volgorde en type)* van de methods, waar je met deze delegate naar wil verwijzen.

***delegate******void*** *DelegateMethod(parameters);*

***delegate******resultType*** *DelegateFunction(parameters);*

***Een simpel voorbeeld dat werkt:***

***delegate******void*** *WerknemersLijst(Werknemer[] werknemers);*

*...*

*WerknemersLijst lijst = Werknemer.UitgebreideWerknemersLijst; lijst(arrayVanWerknemers);*

*...*

* Het **type** van de **parameters** van de method **mag** **verschillend** **zijn** van het type van de parameters gedefinieerd door de delegate, op voorwaarde dat de conversie tussen de gegevenstypes mogelijk is.
* Het **type** van de **returnwaarde** van de method **mag** **verschillend** **zijn** van het type van de returnwaarde gedefinieerd door de delegate, op voorwaarde dat conversie tussen de gegevenstypes mogelijk is.

## Events

Een **event** is een **gebeurtenis** die zich kan voordoen **in** **een** **object**: een werknemer neemt ontslag. Andere objecten kunnen in deze gebeurtenis geïnteresseerd zijn om op die gebeurtenis te reageren: een manager roept een werknemer die ontslag neemt bij zich voor een gesprek.

Het is mogelijk dat er **meerdere** **objecten** op **één** **gebeurtenis** **willen** **reageren**, elk op hun eigen manier. Deze reactie op een event wordt als een method geschreven in de class van een object dat op het event wil reageren. De **signatuur** van deze **method** wordt bepaald door het type van het event: **een** **delegate**. Het event type bepaalt dus de signatuur van de methods die kunnen uitgevoerd worden wanneer het event plaatsvindt. In deze module leer je hoe je zelf events kan voorzien in eigen geschreven classes, hoe je deze events kan laten optreden en hoe je op deze events kan reageren door er code aan te koppelen.

**Voorbeeld:**

eerst een **delegate** maken (buiten de class, maar normaal wel binnen hetzelfde bestand):

een **event** aanmaken volgens de signatuur van de delegate:

ergens het **event** **oproepen** (het event kan *null* bevatten als geen enkele method met het event zou zijn verbonden) bijvoorbeeld in een method Fotokopieer(int aantalBlz):

*delegate void Onderhoudsbeurt(Fotokopiemachine machine);*

*public class Fotokopiemachine : IKost {*

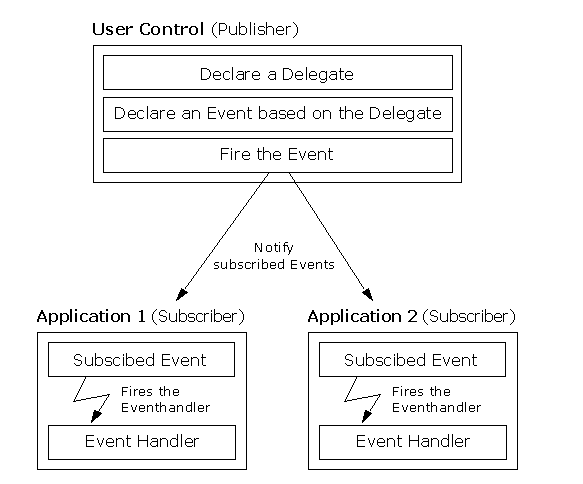
*public event Onderhoudsbeurt* **OnderhoudNodig***;*

public void Fotokopieer(int aantalBlz){ ... ***OnderhoudNodig?.Invoke(this); ...***

**OF** *if (OnderhoudNodig != null) OnderhoudNodig(this); ...**}*

}

Nu kan je in andere classes methods aanmaken volgens dezelfde signatuur als de delegate en deze in het hoofdprogramma aan elkaar koppelen.

machine.OnderhoudNodig += eenBediende.DoeOnderhoud;

## Anonymous methods

Een anonymous method gebruiken heeft enkel zin als je er zeker van bent dat je de code slechts één keer moet uitvoeren. Je kan een delegate *reference variabele* er naartoe laten verwijzen of de delegate ook koppelen aan een event.

***delegate******void*** *WerknemersLijst(Werknemer[] werknemers);*

*...*

*WerknemersLijst rapport;*

*rapport =* ***delegate****(Werknemer[] werknemers) { ... };*

*rapport(wij);*

*...*

OF de anonymous method **koppelen** **aan een event**:

machine.OnderhoudNodig += **delegate**(Fotokopiemachine apparaat) { ... };

# Exceptions

Een **exception** is een signaal dat aangeeft dat er in het programma een uitzonderlijke fout is opgetreden.

De exception zelf is een **object** dat behoort tot de class *Exception* of tot een afgeleide class van *Exception.* Een *Exception* object heeft een:

* ReadOnly property ***Message*** van het type string met een omschrijving van de fout.
* ReadOnly property ***StrackTrace*** van het type string. Met deze property zie je welke methods in uitvoering waren op het moment dat de fout zich voordeed.

***if you can not solve the problem locally, then throw an exception, and handle it where you know what to do about it.*** *Typically use exceptions for exceptional cases – the instance where nothing should go wrong, and if it does go wrong, that’s a big problem. Like a missing hardware driver – they are more expensive than well formed if/else logic.*

## Exceptions opvangen met try – catch

Als je geen maatregelen neemt om exceptions op te vangen, stopt het programma zodra zich een fout voordoet.

Als C# de ingetikte tekst niet naar een getal kan converteren, werpt C# een exception van het type ***FormatException*** (een afgeleide classe van *Exception*).

Als er wordt gedeeld door 0, werpt C# een exception van het type ***DivideByZeroException***.

! Als je fouten kan opvangen met een *if* is dit de te prefereren manier: het is het meest performante.

Je kan echter niet alle mogelijke fouten voorkomen met een *if*. Dus moet je code die mogelijke fouten kan veroorzaken omsluiten met de opdracht ***try*** en accolades. Na de sluitaccolade van try vang je de fouten op met één of meerdere *catch* opdrachten. Iedere *catch* opdracht heeft op zich een open en sluit accolade. Als er tijdens de uitvoering van het programma een fout optreedt in de code binnen de try, gaat het programma verder naar de **eerste** ***catch*** **opdracht** waarvan het type exception (vermeld tussen de ronde haakjes van de catch opdracht) overeenkomt met het type van de fout die opgetreden is. Dit is de *catch* opdracht waarvan het type exception **gelijk** is aan **of** een **base** **class** is van het **type** **exception** dat opgetreden is in de code. De code van deze *catch* opdracht wordt dan uitgevoerd. In deze code kan je een gebruiksvriendelijke boodschap tonen en/of het programma op een ‘nette’ manier afsluiten. Indien geen enkele *catch* de fout kan opvangen, stopt het programma. Als je na het sleutelwoord *catch* **geen** **type** tussen ronde haakjes vermeldt, vangt deze ***catch*** opdracht **alle** **exceptions** op: exceptions van het type *Exception* én exceptions waarvan het type een afgeleide class van *Exception* is.

Om technische informatie over de fout te weten te komen kan je het type een variabele geven en deze dan verder uitwerken:

***catch*** *(FormatException* ***ex****) { Console.WriteLine(****ex****.****StackTrace****); }*

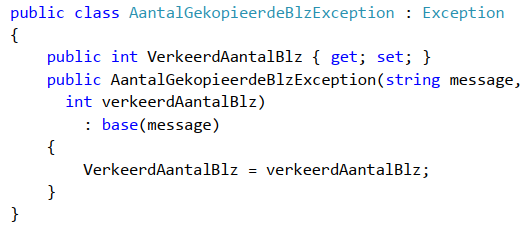
Om te weten bij **welke** **invoer** het fout is gelopen. Kan je de ***try opdrachten nesten***.

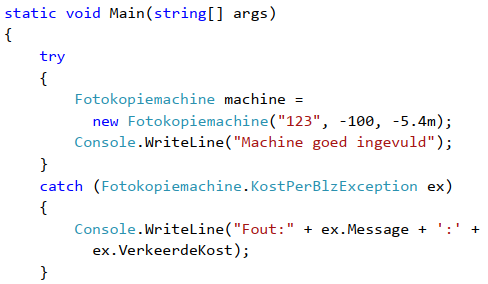
## Een fout melden met throw

Je kan ook zelf een exception geven door een type *Exception* te *throwen* en deze dan verderop (of buiten de method) op te vangen met een *catch*. Zodra je een fout werpt, springt C# naar de **dichtsbijzijnde** **catch** die de fout kan opvangen, of stopt het programma als geen enkele *catch* de fout kan opvangen.

*throw new Exception(“Aantal gekopieerde blz. < 0!”);*

## Een eigen exception class ontwerpen

Je kan geen onderscheid maken tussen een foutieve KostPerBlz en een foutief AantalBlz, tenzij met een aangepaste message. Als een fout optreedt, kan je enkel een omschrijving van de fout meegeven, maar geen extra informatie, bv het verkeerde bedrag in KostperBlz. Een eerste probleem los je op door per fout een aparte **INNER** class te maken: bijvoorbeeld een class KostPerBlzException en een class *AantalGekopieerdeBlzException.* Je werkt de *Exception* classes uit als een *inner class* van uw object. Je geeft deze 2 properties**: Een omschrijving van de fout** en de **verkeerde waarde** van de nieuwe property.



## Gegarandeerde tijdige opkuis van resources

Sommige bestanden worden niet zoals een variabele verwijderd door ***.NET garbage collector*** zodra deze niet meer (kunnen) worden gebruikt. Dit heeft de volgende nadelen:

* Je kan het bestand in die tijd **niet** **verwijderen**
* Je kan het bestand in die tijd **hernoemen**
* Als je het bestand exclusief opende, kunnen anderen het bestand in die tussentijd **niet** **openen**

Deze bestanden moeten zo snel mogelijk gesloten worden zodat dit niet kan gebeuren. Bij de classes die een resource voorstellen zijn er één van de twee methodes: ***Close()***of ***Dispose()*.**

Als je zeker wil zijn dat het bestand tijdig wordt gesloten kan je gebruikmaken van een ***finally*** blok. Deze code zal ongeacht een ***exception*** of ***return*** worden uitgevoerd:

*public int ProvincieGrootte(string provincieNaam)  
{*

*StreamReader lezer = new StreamReader(@"C:\VS2017\provincies.txt");****try*** *{*

*string regel;  
while ((regel = lezer.ReadLine()) != null)  
{*

*int dubbelPuntPos = regel.IndexOf(':');  
string provincie = regel.Substring(0, dubbelPuntPos);  
if (provincie == provincieNaam)*

*return int.Parse(regel.Substring(dubbelPuntPos + 1)); (1)*

*}*

*}****finally*** *{*

*lezer.Close(); (2)  
}  
throw new* ***Exception****("Onbestaande provincie: " + provincieNaam);*

*}*

Maar de code kan nog **korter** met een ***using*** opdracht. Je past deze opdracht toe bij het aanmaken van een object dat je achteraf wil sluiten. Zelfs bij een ***exception*** of ***return***.

public int ProvincieGrootte(string provincieNaam)  
{

**using** (StreamReader lezer = new StreamReader(@"C:\VS2017\provincies.txt"))  
**{**

string regel;  
...

**}**  
throw new Exception("Onbestaande provincie:" + provincieNaam);

}

**OPMERKINGEN**:

***using*** is een verkorte versie van *try – finally*. Je kan dus met using **geen *catch*** verbinden. Als je foutopvang met *catch* en *finally* nodig hebt, zal je dus geen *using* kunnen gebruiken.

Je kan *using* enkel gebruiken voor objecten die de interface ***IDisposable*** implementeren. Dit kan je opzoeken in het programma of online.

# Null operatoren

## De null-conditional operator ?

***afdelingsnaam = asterix.Afdeling?.Naam;***

De operator ?. test eerst of het object waarvan je de property of method wil gebruiken *null* is. Zo ja, dan wordt de expressie niet verder uitgevoerd en geeft ze als resultaat *null* terug.

## De operator ??

***afdelingsnaam = asterix.Afdeling?.Naam??”onbekend”;***

Met de operator ?? kan je een defaultwaarde opgeven voor het geval een expressie een *null* waarde teruggeeft. Een NullReferenceException:

*Console.WriteLine(sb?.ToString().ToUpper()??string.Empty);*

# Indexers

Indexers in een class laten toe data van objecten van deze class aan te spreken zoals data van een array: deze data gedragen zich als een verzameling waarvan je één element kan opvragen. **Indexers** zijn **krachtiger** **dan** **gewone** **arrays**. Een class kan meerdere indexers bevatten. Bijvoorbeeld de class *String* bevat een indexer om individuele tekens aan te spreken via een *index*.

Indexers worden gebruikt bij classesdieeen **interne verzameling** bevatten:

* Hetzij op basis van een volgnummer of index van het element, zoals bij een array.
* Hetzij op basis van een sleutelwaarde (bijvoorbeeld een string)

private int[] overurenValue = new int[12];

*public int* ***this[int index]*** *{get; set; }*

*public int* ***this[string index]*** *{get; set; }*

Een indexer zorgt ervoor dat men een array als property kan hebben en deze adhv deze *constructor*-achtige properties kan aanspreken via de index.

Om deze via een **string** te kunnen aanspreken wordt een **2e array** gemaakt en een **methode** die de overkomstige index zoekt:

private static readonly string[] maanden =

{"jan","feb","maa","apr","mei","jun",

"jul","aug","sep","okt","nov","dec"};

public int this[string maand]

{

get

{

return overurenValue[WelkeMaand(maand)];

}

set

{

overurenValue[WelkeMaand(maand)] = value;

}

}

private int WelkeMaand(string maand)

{

int maandNr = Array.IndexOf(maanden, maand);

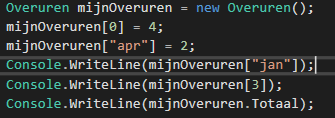
if (maandNr == -1)

throw new IndexOutOfRangeException("Ongeldige maand:"

+ maand);

return maandNr;

}

Aan te spreken en in te vullen via:

# Collections en Generics

Arrays hebben hun beperkingen; het vergt namelijk wat werk om een array te vergroten, elementen in te voegen of elementen te schrappen. Daarom zijn er ***Collections***, het is verzameling voor verschillende soorten **rekbare lijsten**. *Collections* vinden we terug in de namespace ***System.Collections***;

|  |  |
| --- | --- |
| ArrayList | Een rekbare array. |
| BitArray | Een array van booleans, opgeslagen als bitwaarden. |
| Hashtable | Een lijst van waarden met elk een key (key/value pairs). De sortering gebeurt a.d.h.v. de hashcode van de key. |
| Queue | Dit is een first-in-first-out lijst. |
| SortedList | Een verzameling waarden met bijhorende sleutel (key/value pairs). De sortering gebeurt op basis van de sleutel die bij elke waarde hoort. De waarden kunnen opgevraagd worden via een index of via de sleutel. |
| Stack | Dit is een last-in-first-out lijst. |

## ArrayList

Bij wijze van voorbeeld demonstreren we in deze paragraaf het gebruik van een ArrayList. De voornaamste methods en properties zijn:

|  |  |
| --- | --- |
| Count | Levert het aantal elementen van de lijst. |
| Add() | Voegt een element achteraan de lijst toe. |
| Clear() | Verwijdert alle elementen uit de lijst. |
| Contains() | Controleert of een welbepaald element in de lijst zit. |
| IndexOf() | Zoekt een element in de lijst en geeft de index ervan. |
| Insert() | Voegt een element toe aan de lijst op een welbepaalde plaats. |
| LastIndexOf() | Zoekt het laatste voorkomen van een element in de lijst. |
| Remove() | Verwijdert een welbepaald element uit de lijst. |
| RemoveAt() | Verwijdert het element op de opgegeven positie uit de lijst. |
| Reverse() | Draait de volgorde van de elementen in de lijst om. |
| Sort() | Sorteert de elementen van de lijst. |

**OPMERKING:**

Elementen van een *ArrayList* worden altijd als **gewone objecten** opgeslagen. Daarom moeten ze achteraf nog altijd terug naar een andere class gecast worden.

## Generic lists

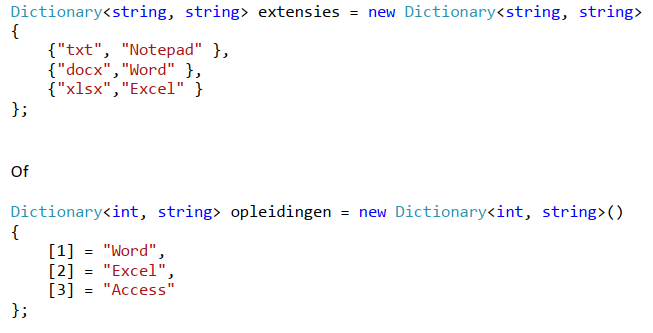
Om het vorige probleem van steeds te moeten casten op te lossen heeft het .NET Framework sinds 2.0 ***generic lists***. De namespace *System.Collections.Generic* bevat een aantal classes zoals *List, Queue, SortedList, Stack, Dictionary*. In tegenstelling tot de classes in de namespace *System.Collections* bevatten deze collections elementen van een welbepaald type en niet van het algemene type *Object*.

in de plaats van *ArrayList personeel = new ArrayList();* doen we nu ***List<Werknemer>*** *personeel = new* ***List<Werknemer>()****;*

Nu hoef je niet langer een cast uit te voeren maar kan er ook maar één type worden opgeslagen in de list.

## Dictionary

De generic ***Dictionary<TKey, TValue>*** is nog zo’n veel gebruikte collectie. Bij iedere value hoort een key.

 *Dictionary<int, string> opleidingen; Dictionary<string, string> extensies;*

**TOEVOEGEN**

opleidingen.Add(4, “Powerpoint”); OF opleidingen[5] = “Outlook”;

als de key niet aanwezig is wordt het element toegevoegd.

**OPVRAGEN**

extensies[“docx”] of **opleidingen.TryGetValue(2, out string opleiding)** --> geeft de waarde in *opleiding* of geeft *false* terug.

# Nullable types

Value types krijgen bij declaratie meteen een default waarde. Voor getallen is dit 0, voor een bool *false* en voor een DateTime is dit 1/1/0001 00:00. Als je een variabele zoals *aantalKinderen* wil gebruiken weet je niet of de Werknemer 0 kinderen heeft of dat dit een default waarde is. In de databasewereld gaat dit wel en met een zogenaamde *nullable type* gaat dit nu ook in C#. Je definieert ze als volgt:

***byte? aantalKinderen;***

***bool? gehuwd;***

Een local variabele (gedeclareerd in een method) van een *nullable* type en geen beginwaarde toegekend, zal een compilerfout geven wanneer je deze wil gebruiken.

Een member variabele (gedeclareerd in een class en niet *static*) zal als default waarde *null* krijgen.

**aantalKinderen.HasValue;**

**aantalKinderen.Value;**

**OPMERKING:**

Als je een *nullable* type waarde wil doorgeven aan een *Value* type moet je eerst controleren op een *null* waarde:

**aantalKamers = aantalKinderen ?? 0;**

# Extra taalelementen

## Pattern matching

Je kan niet alleen testen op waardes in *if* en *switch-*statements maar ook op ***type*** dmv de ***is***-operator en het ***switch***-statement. Wat nieuw is in C#7, is dat je niet alleen de controle kan uitvoeren op een waarde maar deze ook meteen een variabele van het betreffende type aan toekennen.

Voorbeelden:

**if (obj is Werknemer w) {Console.WriteLine(“... w.Naam ...”);**

Bij **switch**-statements werkt het hetzelfde. Je kan bij controle meteen een variabele declareren en initialiseren.

**OPMERKING:**

Je kan een switch-statement uitbreiden met ***when***.

***switch*** *(item) {*

***case*** *Werknemer w* ***when*** *w.Geslacht == Geslacht.Man:*

*...*

***break****;*

*}*

## Extension methods

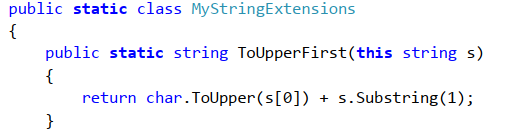
Zogenaamde extension methods kunnen gedefinieerd worden om extra functionaliteit toe te voegen aan **bestaande** types (classes, interfaces, Value types), zonder echter de source code van deze types te wijzigen of een nieuwe afgeleide class of interface te creëren.

Je kan extension methods gebruiken als je bijvoorbeeld beter niets wil veranderen aan de source code, je geen extra derived class wil maken of het om een *sealed class* gaat en je geen derived class kan maken.

**Extension methods zijn altijd static methods die in een static class gedefinieerd worden.** Wat opvallend is, normaal zou je deze methods niet kunnen toepassen op objecten van deze class.

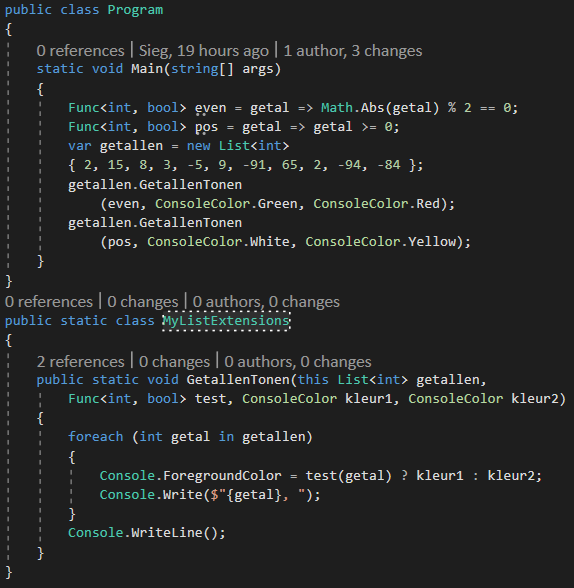
Voorbeeld:

twee extension methods voor *String: ToUpperFirst()* en *Right(n)*. We definiëren deze in een nieuwe class *MyStringExtensions.cs*.



De **this string s** als parameter geeft aan dat dit een extension method is voor de class String. Voor de rest is deze parameter niet nodig.

* Het is verstandig om alle extension methods logisch te groeperen in een aparte namespace.



## Object initializers

**In de plaats van (geparametriseerde) constructors te gebruiken kan je gewoon OBJECT INITIALIZERS gebruiken.**

**var eersteVrouw = new Persoon { Naam = “Adam”, aantalKinderen = 2 };**

## Collection initializers

Collections meteen initialiseren bij de declaratie. Een voorbeeld met de combinatie van object initializers:

*var personen = new List<Persoon> {*

*new Persoon {Naam=”Adam”, AantalKinderen=2},*

*new Persoon {Naam=”Eva”, AantalKinderen=2}*

*};*

## Anonymous types

Anonymous types spelen een rol bij LINQ. Daar deze geen methods bevatten en je ook nergens kan verwijzen naar de naam van een anoniem type, kan je het type ook niet gebruiken om objecten van dit type door te geven tussen methods.

Anonymous types worden on-the-fly gecreëerd en gedifinieerd dmv object initializers.

*var persoon = new { Nr = 1, Naam = “Adam”, AantalKinderen = 2};*

* Anonieme types erven rechtstreeks van de class *Object*.
* In C# zijn de properties van een object van een anoniem type ***readonly***.
* Twee objecten van een anonymous type zijn gelijk als de waarden van al hun properties hetzelfde zijn.
* Twee anonymous types zijn hetzelfde als hun properties dezelfde naam en hetzelfde type hebben, én als de properties in dezelfde volgorde staan.

## Partial methods

Partial methods kunnen alleen tot stand komen in partial classes. Een partial method is een **lege private method** van het type **void**. Deze heeft twee delen: de definitie en de implementatie.

Private methods kunnen handig zijn wanneer een class/code door VS wordt gegenereerd. Als je de method in de door VS gegenereerde code helemaal zou uitschrijven, loop je het gevaar deze te verliezen wanneer VS deze code opnieuw genereert.

De method wordt als volgt gedefinieerd:

**partial void NaamChanged(string naam);**

In het andere deel van de partial class kan nu de method verder worden uitgeschreven (mits dezelfde signatuur). Mocht de partial method **nergens** **geïmplementeerd** zijn door de programmeur, **verwijdert**/negeert de compuler **automatisch** de method-oproepen naar deze partial method, zodat er geen compiletime of runtime fouten optreden.

**partial void NaamChanged(string naam) { };**

## Lambda λ expressions

Een lambda expressie is een andere schrijfwijze voor een anonymous method. Een lambda expressie is een **beknopte**/functie zonder naam met **dezelfde functionaliteit** als een anonymous method. Het is een expressie die **nul, één of meerdere parameters** binnenkrijgt en al dan niet een **returnwaarde** heeft. Een lambda expressie kan **één of meerdere statements** bevatten. Lambda expressies worden **in** **combinatie** met **delegates** gebruikt.

**Een predicate method**: een method die op basis van een parameter *true* of *false* teruggeeft.

**Een anonymous method wordt in een delegate variabele gestopt:**

*Filter getalDeelbaarDoorVijf = delegate(int getal)  
{*

*return (getal % 5 == 0);*

*};*

De syntax van een lambda expressie is **beknopt** en **eenvoudig** en daardoor beter leesbaar. Een lambda expressie beschrijft ***wat*** moet gebeuren, **niet** ***hoe***iets moet gebeuren. Dit heet ***declaratief*** **programmeren**, in tegenstelling met het ***imperatief*** **programmeren** waarbij de code álle logische opdrachten bevat die moeten uitgevoerd worden. Bijvoorbeeld een verzameling gegevens sorteren kan met één lambda expressie.

Een lambda expressie bestaat uit **3 onderdelen**:

1. Een parameterlijst met nul, één of meerdere parameters die als argument(en) voor de lambda expressie gebruikt worden.
2. Het symbool =>
3. Eén expressie (*expression lambda*) of meerdere statements tussen accolades (*statement lambda*): de code hiervan wordt uitgevoerd voor de meegegeven parameter(s) en geeft al dan niet een resultaatwaarde terug.

* ***Expression lambda’s***

**getal => getal \* getal**

Deze lambda expressie geeft het kwadraat van de opgegeven parameter getal als resultaat terug. Bij een lambda expressie met slechts één parameter, mogen de haakjes () rond de parameter weggelaten worden.

**(getal1,getal2) => getal1 + getal2**

Deze lambda expressie geeft de som van de meegegeven parameters getal1 en getal2 als resultaat terug.

**() => new Random().Next(100)**

Deze lambda expressie geeft een positief willekeurig getal, kleiner dan 100 terug. Merk op dat als de lambda expressie geen parameter(s) bevat, de haakjes toch vermeld  
moeten worden.

**getal => getal % 2 == 0**

Deze lambda expressie geeft true wanneer de meegegeven parameter getal even is en false wanneer getal oneven is.

* **Statement lambda’s**

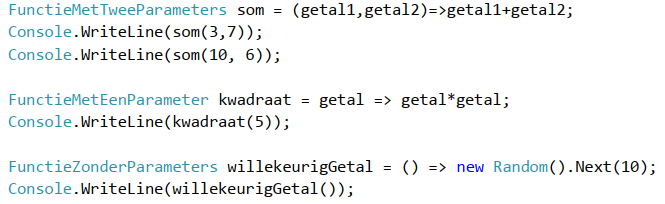
Bij een statement lambda worden de statements tussen accolades geplaatst. Dergelijke  
statement lambda’s kunnen al dan niet een returnwaarde hebben. Als de lambda expressie  
een returnwaarde heeft, vermeld je dit als laatste opdracht met het sleutelwoord return.  
Het aantal statements is in de praktijk meestal beperkt tot twee of drie.

**getal => { return getal % 2 == 0;}**

**tekst => {**

**var letters = tekst.ToCharArray();  
Array.Reverse(letters);  
Console.WriteLine(new String(letters));**

**};**

**OPMERKING:** om de lambda expressie’s te kunnen uitvoeren, moet je de expressies toewijzen aan een *delegate reference variabele* met dezelfde signatuur.

*Lambda expressions* kunnen ook worden doorgegeven **als** **parameter**. Ze moeten overeenkomen met het delegate type dat wordt verwacht als parameter:

*Console.WriteLine("Getallen deelbaar door 5:");  
ToonGetallen(getal => getal % 5 == 0);*

*Lambda expressions* kunnen ook gebruikt worden als **returnwaarde** van een method:

*static Filter MaakLambda()*

*{ Console.Write("Geef een getal: ");  
var deelbaarDoor = int.Parse(Console.ReadLine());  
return getal => getal % deelbaarDoor == 0;}*

private static void ToonGetallen(Filter filter)

{var getallen = new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
foreach (var getal in getallen)  
if (filter(getal))  
Console.WriteLine(getal);}

*public static void Main(string[] args)*

*ToonGetallen(MaakLambda());}*

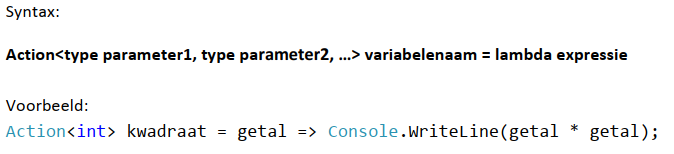
**Opvallend** is dat de *lokale variabele deelbaarDoor* nog steeds gekend is in de method ToonGetallen. De functie MaakLambda() wordt één keer uitgevoerd bij ToonGetallen(MaakLambda()); en geeft op dat moment de **lambda functie** *getal => getal % deelbaarDoor == 0* terug. **Vanaf dan** wordt alleen nog deze functie uitgevoerd maar blijft de lokale variabele ***deelbaarDoor*** wel gekend en bestaan. Dit noemt met **VARIABLE LIFTING**: de lokale variabele leeft even lang als de lambda expressie.

## Action en Func delegates

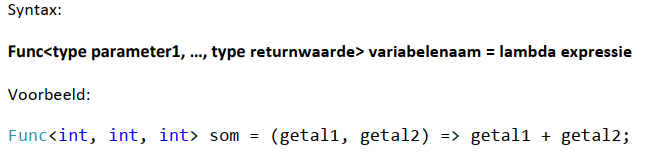
Ipv zelf in de code expliciet een delegate type te definiëren met het sleutelwoord *delegate*, kan je gebruik maken van de ingebouwde delegate types ***Func*** en ***Action*** van het .NET Framework. De signatuur van de method of lambda expressie moet nog steeds voldoen aan de signatuur van de Action/Func delegate.

**Action**

Met een action delegate type kan je verwijzen naar een method of lambda expressie **zonder returnwaarde**. Deze sub of lambda expressie kan **0 tot 16 parameters** bevatten. *Action* werkt met generics: de parameters kunnen van eender welk type zijn.



**Func**

Met een Func delegate type kan je verwijzen naar een method of lambda expressie **met een returnwaarde**. Deze functie of lambda expressie kan **0 tot 16 parameters**. Func werkt eveneens met generics: zowel de returnwaarde als de parameters kunnen van eender welk type zijn. Het type van de **returnwaarde** vermeld je als **laatste parameter** tussen de <> van *Func*.

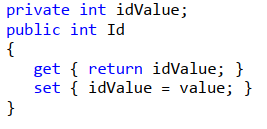
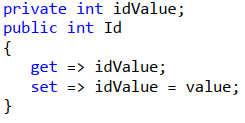
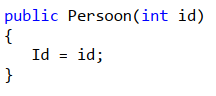
* Met de sleutelwoorden *Func* en *Action* kan je ook verwijzen naar een method die elders gedefinieerd is.
* *Action* en *Func* delegates kunnen ook als type gebruikt worden voor de returnwaarde van een functie of voor de parameters van een functie/sub.

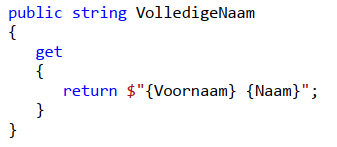
## Expression-bodied methods en properties

ipv de body van een method die één expressie bevat of de get/set van een property uit te schrijven tussen accolades, kan je de lambda notatie (pijl =>) gebruiken. Hierdoor schrijf je minder code en kan de code beter leesbaar zijn. Het betreft hier geen lambda expressies, enkel de lambda notatie (=>) wordt gebruikt.

**methods:**

**properties:**





# LINQ – een introductie

LINQ of Language Integrated Query is een *SQL-achtige* technologie die je toelaat met dezelfde syntax met gegevens te werken, ongeacht de bron van deze gegevens (een array, een collection, een relationele database, een XML bestand, een tekstbestand, een web service ...). Tot nu toe moest je , om met gegevens van een specifieke gegevensbron te werken, hiervoor telkens een specifieke taal gebruiken: Transact-SQL voor MS SQL Server, XPatch of XQuery voor XML gegevens, geneste for/if statements voor arrays en collections ... Met LINQ wordt het allemaal eenvoudiger: je kan dezelfde C# syntax gebruiken voor elk type gegevensbron. Bovendien krijg je bij het schrijven van een LINQ query uitgebreide hulp (IntelliSense) en wordt de syntax van de code reeds gecontroleerd at compile-time. LINQ voorziet een set van standaard query operatoren om gegevens te selecteren, te filteren, te sorteren, te groeperen, te totaliseren, te koppelen met andere gegevens, ... .

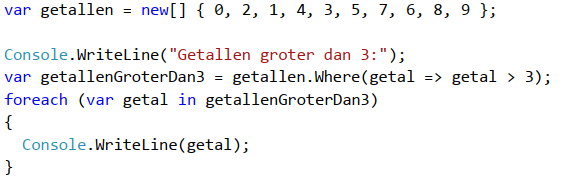
Je kan zelfs gebruikmaken van alle nieuwe taalelementen zoals *local type inference, object initializers* ... .

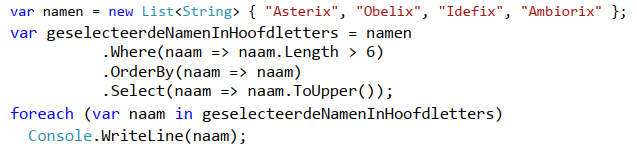
In één query kunnen gegevens uit meerdere verzamelingen en uit verschillende bronnen. De **namespace** ***System.Linq*** moet geïmporteerd worden.

## *IEnumerable*

Elke gegevensverzameling of collection die de **interface *IEnumerable*** implementeert kan *verwijderen, toevoegen ...* en met behulp van een ***Enumerator* object** ook itereren. Het is als een soort **verplaatsbare pointer**. Bijvoorbeeld een *foreach* constructie is gebaseerd op een *Enumerator*. In de static class *Enumerable* zijn een aantal **extension methods** gedefinieerd voor de interface *IEnumerable*. Dmv deze extension methods kan je query’s schrijven waarmee je gegevens kan selecteren, filteren, sorteren, berekeningen op gegevens kan uitvoeren ... vergelijkbaar met de SQL query’s. ***Select(), Where(), OrderBy() ...***

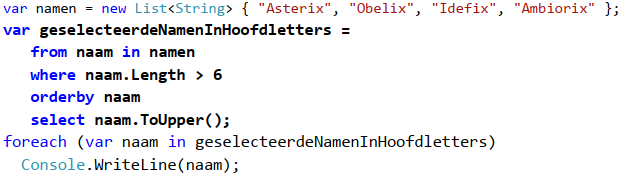
De resultaatwaarde van deze extension methods kan opnieuw van het type *IEnumerable* zijn, zodat ze op elkaars resultaat kunnen opgeroepen worden. Deze extension methods verwachten een **delegate als parameter**. Deze delegate (of lambda expressie) wordt dan uitgevoerd **voor elk element** van de verzameling.





## Query Comprehension Syntax/Query Expression Syntax

C# voorziet met Query Expressions een nog meer **beknopte** **syntax** dan extension methods en lambda expressies. Maar uiteindelijk blijf je een mix gebruiken, NIET combineren (query expression syntax bevat bijvoorbeeld **geen** **Distinct** **operator**).

De volgende code doet hetzelfde als de laatste code. Deze wordt door de C# compiler vertaald naar een equivalente query die gebruik maakt van extension methods en lambda expressies. Het resultaat wordt toegekend aan een variabele via ***local type inference*** van het type ***IEnumerable<string>***.

**OPMERKING**: Toch verschilt de LINQ syntax van de SQL syntax en is de gedachtengang bij een LINQ query helemaal anders dan bij een SQL query. Een LINQ query begint met een **from** clausule en eindigt met een **select-** of **group by** clausule. De ***from*** introduceert een iteratie variabele en ***select*** bepaalt welke gegevens uit de verzameling in het resultaat opgenomen worden.

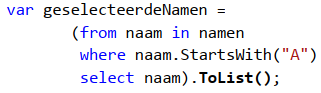
***deferred execution*** of ***lazy evaluation***:

De LINQ query wordt pas uitgevoerd wanneer deze in de *foreach* doorlopen wordt, **NIET** op het moment deze geïnitialiseerd wordt.

***ToList()***

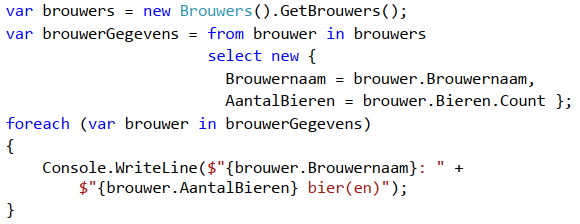
**je kan dit oplossen** door gebruik te maken van de **ToList()** method. Deze geeft een List<T> terug.

***FirstOrDefault()***

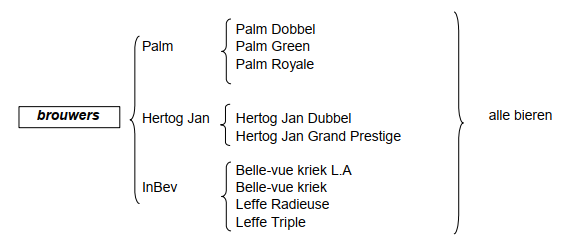
Deze geeft meteen het eerste element van het resultaat van de LINQ query of de *default* waarde van het type.

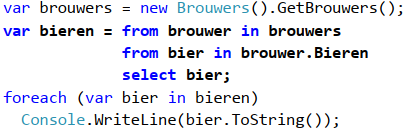
## Voorbeelden

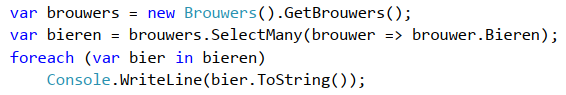
### ANONYMOUS TYPE met een naam

Je kan in de **select**nieuwe objecten creëren op basis van een *anonymous type*. Hier wordt voor elke brouwer een object gecreëerd met een property ***Brouwernaam*** (type string), die de naam van de brouwer bevat, en een property ***Count*** (type int), die het aantal bieren van de brouwer bevat. In de *foreach* constructie kunnen deze properties van het *anonymous type* ook met een naam aangesproken worden.

### Een DUBBELE ITERATIE of een list van een list selecteren:



Om **alle bieren** te tonen volstaat het niet om “var bieren = from **brouwer** in **Brouwers** select **brouwer.Bieren**;” uit te voeren en deze dan via **foreach** te overlopen. Hierdoor krijg je enkel een lijst van elementen van het type ***List<Bier>***. Om deze te selecteren heb je een dubbele iteratie nodig:

***SelectMany()*** geeft hetzelfde resultaat:

### Het GEMIDDELDE berekenen + een SELECTIE:

### GROUP BY

Het resultaat van een group by is een verzameling van IGrouping objecten die op basis van een Key waarde verdeeld zijn. Tussen *group* en *by* vermeld je uit welke verzameling en na *by* op welke waarde.

var brouwers = new Brouwers().GetBrouwers();  
var opBelgisch = from brouwer in brouwers

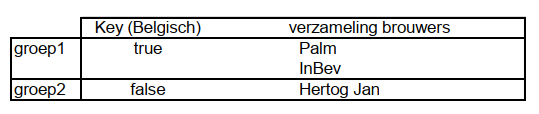
group brouwer by brouwer.Belgisch;

foreach (var welNietBelgisch in opBelgisch)  
{

Console.WriteLine(welNietBelgisch.Key ? "Belgisch" : "Niet-Belgisch");

foreach (var brouwer in welNietBelgisch)

Console.WriteLine(brouwer.Brouwernaam);

}

De LINQ query kan je ook schrijven met de extension method ***GroupBy()*** waaraan je een lambda expressie als parameter meegeeft:

var brouwers = new Brouwers().GetBrouwers();  
var opBelgisch = brouwers.GroupBy(brouwer => brouwer.Belgisch);

### ORDER BY - INTO

Indien je de groepen wil sorteren moet je deze sortering zelf in de LINQ query voorzien. Met het sleutelwoord ***into*** creëer je een tijdelijke variabele waarin je het resultaat van een **select, group by**of **join**bijhoudt. Via deze variabele kan je verder in query naar de groepen verwijzen.

**! EEN LINQ QUERY MOET STEEDS EINDIGEN OP group OF select. !**

var brouwers = new Brouwers().GetBrouwers();

var opAantalBieren = from brouwer in brouwers

group brouwer by brouwer.Bieren.Count **into mijnGroep  
orderby mijnGroep.Key *OF where mijnGroep.Key == 3***

**select mijnGroep;**

foreach (var aantalBieren in opAantalBieren)

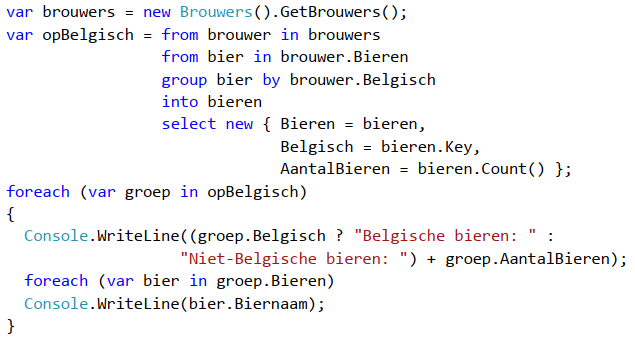
{

Console.WriteLine(aantalBieren.Key);

foreach (var brouwer in aantalBieren)

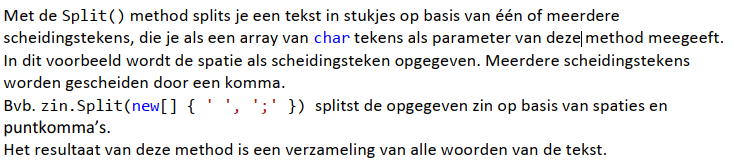
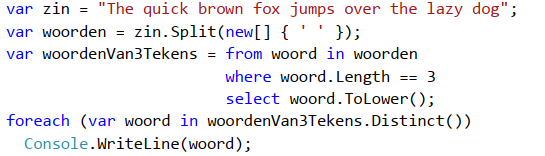
Console.WriteLine(brouwer.Brouwernaam);

}

Belgisch en niet-Belgisch bier tonen + het aantal per groep + de verzameling bieren afprinten per groep:

### JOIN

### bewerkingen op een STRING



## Oefeningen

### List<> met objecten

var planten = new List<Plant> {

new Plant {PlantID=1, Plantennaam="Tulp", Kleur="Rood", Prijs=0.5m, Soort="bol" },

new Plant {PlantID=2, Plantennaam="Krokus", Kleur="wit", Prijs=0.2m, Soort="bol"},

new Plant {PlantID=3, Plantennaam="Narcis", Kleur="geel", Prijs=0.3m, Soort="bol"},

};

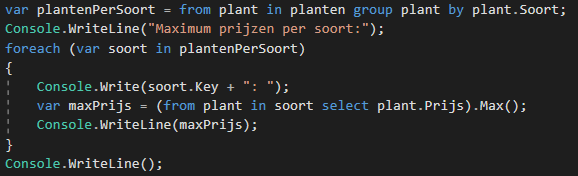
### Count()

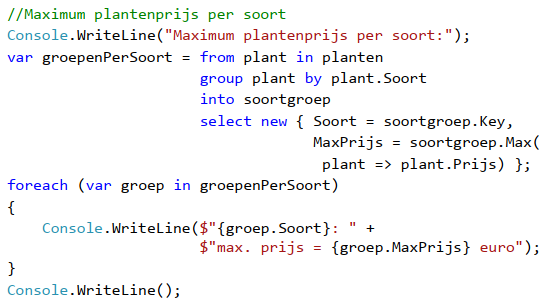
Een count pas je toe op de hele LINQ query:

(from plant in planten where plant.Kleur == "wit" select plant.PlantID).Count();

### Max()

Maximum prijs per soort op 2 manieren:





### Soorten sorteren op alfabetische volgorde met het aantal planten per soort, en de namen per soort

Console.WriteLine("Soorten alfabetisch:");

var soortenAlfabetisch = from plant in planten orderby plant.Soort

group plant by plant.Soort

into soortgroep

select new {Soortnaam = soortgroep.Key,

AantalPlanten = soortgroep.Count(),

Planten = soortgroep};

foreach (var soort in soortenAlfabetisch)

{

Console.WriteLine($"{soort.Soortnaam} ({soort.AantalPlanten})");

foreach (var plant in soort.Planten)

{

Console.WriteLine($"\t {plant.Plantennaam}");

}

}

### Groeperen op 2 kenmerken

Console.WriteLine("PLANTEN per SOORT per KLEUR");

var groepenPerSoortKleur = from plant in planten

group plant by plant.Soort

into soortgroep

select new

{

Soort = soortgroep.Key,

GroepKleur = from plant in soortgroep

group plant by plant.Kleur

into kleurgroep

select new

{

Kleur = kleurgroep.Key,

Planten = kleurgroep

}

};

foreach (var soort in groepenPerSoortKleur)

{

Console.WriteLine($"Soort: {soort.Soort}");

foreach (var kleur in soort.GroepKleur)

{

Console.WriteLine($" Kleur: {kleur.Kleur}");

foreach (var plant in kleur.Planten)

Console.WriteLine($" {plant.Plantennaam}");

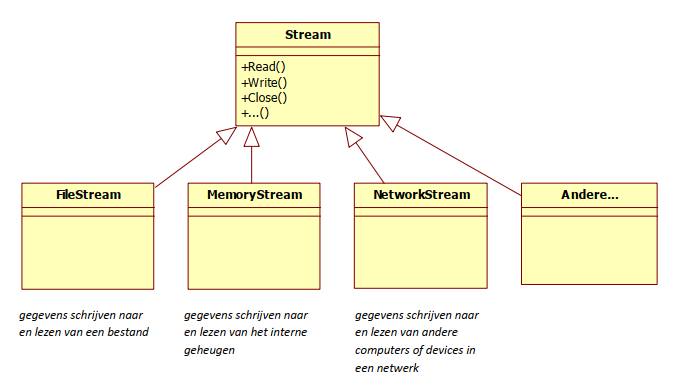
}

}

**OF** GROUP plant BY NEW {plant.Soort, plant.Kleur}

INTO test SELECT NEW classname() {.. = test.Key.Soort, ... = test.Key.Kleur }

# Files and Streams – I/O

.NET maakt gebruik van zogenaamde **streams** om gegevens te transporteren. Een stream is een gegevensstroom, een verzameling van bytes. Om gegevens naar een andere locatie te versturen of van een andere locatie te ontvangen, maakt een programma altijd gebruik van een **Stream object**.

Er zijn meerdere ingebouwde stream classes. Deze zijn allemaal afgeleid van de abstract base class **Stream, uit de namespace System.IO**.

In deze cursus beperken we ons tot het wegschrijven naar en het inlezen van gegevens van een bestand op de harde schijf. Hiervoor gebruik je de class **FileStream**.

* Eerst wordt een ***FileStream*** object gecreëerd
* Dit *FileStream* object wordt **gekoppeld** aan een *bestand*. **LET OP**: **één** **FileStream** object kan tegelijkertijd maar aan **één** **bestand** gekoppeld worden.
* Stream schrijven “**bytes**” naar een bestand. De tekst die je wil wegschrijven, moet dus geconverteerd worden naar een array van bytes. Dit wordt ***encoding*** genoemd. Het Stream object neemt dit voor zijn rekening. Met de ***Write()*** method worden deze bytes dan naar het bestand weggeschreven.
* Hierna moeten het **FileStream** object **gesloten** worden. Als je dit vergeet, blijft het bestand gelocked en kan het niet door andere programma’s gebruikt worden.

Om deze gegevens terug in te lezen wordt op dezelfde manier gewerkt.

## Bestanden en directories

### De classes Directory en DirectoryInfo:

* nagaan of een directory bestaat
* een nieuwe directory creëren
* een directory verplaatsen, hernoemen, verwijderen
* een lijst van bestanden van een opgegeven directory opvragen
* de subdirectories van een opgegeven directory opvragen

string directorynaam = **@"C:\Data"**;

if (**Directory.Exists**(directorynaam))  
{

Console.WriteLine($"De directory {directorynaam} bestaat.");  
if (**Directory.GetDirectories**(directorynaam).Count() != 0  
|| **Directory.GetFiles**(directorynaam).Count() != 0)  
{

Console.WriteLine("De directory met inhoud wordt verwijderd");  
**Directory.Delete(directorynaam, true);**

}  
else {

Console.WriteLine("De directory wordt verwijderd");  
Directory.Delete(directorynaam);

}

} else  
{

Console.WriteLine($"De directory {directorynaam} bestaat niet.");  
Console.WriteLine("De directory wordt gecreëerd");  
**Directory.CreateDirectory**(directorynaam);

}

**LET OP:** De class ***Directory***bevat *static methods*. De functionaliteit overlapt met deze van de class ***DirectoryInfo*** maar van deze moet je wel eerst een object creëren.

### De classes File en FileInfo:

* streams creëren
* lezen van of schrijven naar een bestand
* tekst toevoegen aan een bestand
* nagaan of een bestand bestaat
* een bestand verwijderen, kopiëren, verplaatsen
* informatie over een bestand opvragen

**string directorynaam = @"C:\Data";**  
string bestandsnaam = "Pizzas.txt";  
string bestand = directorynaam + @"\" + bestandsnaam;  
**Directory.CreateDirectory(directorynaam);**  
if (!**File.Exists**(bestand))  
{

Console.WriteLine($"Het bestand {bestand} bestaat niet");  
Console.WriteLine("Het wordt gecreëerd");  
string tekst;  
tekst = "Pizza Margherita (tomatensaus - mozzarella): 8 EUR";  
**File.WriteAllText(bestand, tekst);  
tekst = Environment.NewLine +**"Pizza Vegetariana (tomatensaus-mozzarella-groenten): 9.5 EUR";  
**File.AppendAllText(bestand, tekst);**

}  
else  
{

Console.WriteLine($"Het bestand {bestand} bestaat");  
string bestandsinformatie =  
"Datum creatie: " + **File.GetCreationTime**(bestand) +  
System.Environment.NewLine +  
"Datum laatst gebruikt: " + **File.GetLastAccessTime**(bestand);  
Console.WriteLine(bestandsinformatie);  
Console.WriteLine("De inhoud van het bestand:");

string tekst = **File.ReadAllText**(bestand);  
Console.WriteLine(tekst);  
**File.Delete**(bestand);

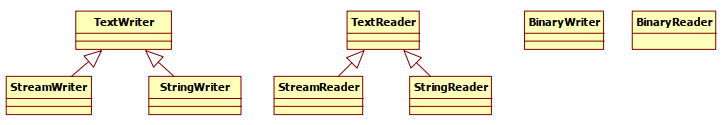
}

**LET OP:** Weeral, de class *File* bevat *static methods,* terwijl de functionaliteit met *FileInfo* gedeeltelijk overlapt maar je van deze clas eerst een object moet creëren.

Deze methods kunnen ***Exceptions*** veroorzaken dus gebruik ***try-catch***blokken.

## Schrijvers en lezers

Het .NET Framework bevat naast de verschillende *Stream* classes andere classes – schrijvers en lezers – waarmee je gegevens naar streams kan schrijven en van streams kan lezen. Zij behoren eveneens tot de namespace *System.IO*. Deze classes **vereenvoudigen** het werk. Zij **creëren zelf automatisch *FileStream* object** en **koppelen** deze aan het **bestand** waarnaar je wil schrijven of waaruit je wil lezen.



|  |  |
| --- | --- |
| *TextWriter* (abstract) | een schrijver waarmee je een sequentiële reeks karakters (tekst) kan wegschrijven. |
| StreamWriter | een schrijver om karakters naar een *stream* te schrijven met een specifieke encoding (standaard UTF-8). |
| StringWriter | een schrijver om gegevens naar een *string* te schrijven. |
| *TextReader* (abstract) | een lezer waarmee je een sequentiële reeks karakters (tekst) kan inlezen. |
| StreamReader | een lezer waarmee je karakters uit een *stream* kan inlezen met een specifieke encoding. |
| StringReader | een lezer om gegevens uit een *string* te lezen. |
| BinaryWriter | een schrijver om primitieve types en strings binair weg te schrijven met een specifieke encoding. |
| BinaryReader | een lezer om primitieve gegevenstypes binair in te lezen met een specifieke encoding. |

### StreamWriter – StreamReader:

Met een **StreamWriter** schrijf je vanuit je programma gegevens naar een bestand.

* Creëer een *StreamWriter* object waarbij je via de constructor het bestand waarnaar je wil schrijven, meegeeft. Dit *StreamWriter* object creëert zelf automatisch een *FileStream* object en koppelt dit aan het bestand waarnaar je wil schrijven. Het bestand wordt automatisch geopend.
* Gebruik de *Write()* of *WriteLine()* method van de *StreamWriter* om de gegevens weg te schrijven.
* Sluit het *StreamWriter* object waardoor de onderliggende stream en dus het bestand losgekoppeld wordt en door andere programma’s kan gebruikt worden.

string locatie = @"C:\Data\";  
try  
{

**using (var schrijver = new StreamWriter(locatie + "Pizzas.txt"))**{

foreach (var pizza in pizzas)  
{

**pizzaRegel = new StringBuilder();**

pizzaRegel.Append(pizza.Naam + ':') ;  
pizzaRegel.Append(pizza.Onderdelen.Count.ToString() + ':');  
foreach (string onderdeel in pizza.Onderdelen)  
{

pizzaRegel.Append(onderdeel + ':');

}  
pizzaRegel.Append(pizza.Prijs);  
**schrijver.WriteLine(pizzaRegel);**

}

}  
//gegevens toevoegen  
**using (var schrijver = new StreamWriter(locatie + "Pizzas.txt",true))**{

schrijver.WriteLine("Pizza Lardiera (tomatensaus-mozzarella-spek): 9.5 EUR”);  
}

}  
**catch (IOException)**  
{

Console.WriteLine("Fout bij het schrijven naar het bestand!");  
}  
catch (Exception ex)  
{

Console.WriteLine(ex.Message);  
}

**OPMERKINGEN:**

* Je omsluit de creatie en het gebruik van het *StreamWriter* object met een ***using***structuur. Op het einde van de wordt het bestand en de gekoppelde stream **automatisch** **gesloten**.
* De class *StreamWriter* bevat meerdere overloaded constructors. Met *True* of *False* kan je aangeven of er toch een **nieuw** **bestand** moet gecreëerd worden (**False**) of dat het **bestaande** **bestand** moet geopend worden en nieuwe gegevens hieraan toegevoegd worden (**True**).
* Wanneer je werkt met een bestand zijn alle **exceptions** afgeleid van ***System.IO.IOException****.* Met deze *catch* opdracht vang je dus alle mogelijke I/O fouten op.

Met een **StreamReader** lees je vanuit je programma gegevens uit een tekstbestand.

* Creëer een *StreamReader* object waarbij je via de constructor het bestand waaruit je wil lezen, meegeeft. Dit *StreamReader* object creëert zelf automatisch een *FileStream* object en koppelt dit aan het bestand waaruit je wil lezen. Het bestand wordt automatisch geopend.
* Gebruik de *Read()* of *ReadLine()* method van de *StreamReader* om de gegevens weg te schrijven.
* Sluit het *StreamReader* object waardoor de onderliggende stream en dus het bestand losgekoppeld wordt en door andere programma’s kan gebruikt worden.

string locatie = @"C:\Data\";  
try  
{

string regel;

**using (var lezer = new StreamReader(locatie + "Pizzas.txt"))**{

**while ((regel = lezer.ReadLine()) != null)**

{

Console.WriteLine(regel);

}

}

}

**catch (IOException)**

{

Console.WriteLine("Fout bij het lezen van het bestand!");  
}  
catch (Exception ex)  
{

Console.WriteLine(ex.Message);

}

**OPMERKING:**

Je kan ook de *Read()* method van het *StremReader* object gebruiken. Hiermee lees je de tekst **teken per teken**, waarbij elk teken als een **Int32** object teruggegeven wordt. De method geeft **-1** terug als je na de het laatste teken nog een teken probeert te lezen:

int teken;  
using (var lezer = new StreamReader(locatie + "Pizzas.txt"))  
{

while ((teken = **lezer.Read()**) != -1)  
{

Console.Write((**char**)teken);

}

}

Makkelijke manier om met een array een bestand te lezen en List met objecten aan te maken:

while ((pizzaRegel = lezer.ReadLine()) != null)  
{

**string[] gegevens = pizzaRegel.Split(new Char[] { ':' });**pizzaNaam = **gegevens[0];**  
aantalOnderdelen = int.Parse(**gegevens[1]**);  
pizzaOnderdelen = new List<string>();  
for (var teller = 0; teller < aantalOnderdelen; teller++)  
{

pizzaOnderdelen.Add(**gegevens[teller + 2]);**

}  
prijs = decimal.Parse(**gegevens[gegevens.Length - 1]**);  
pizzas.Add(

new Pizza  
{  
Naam = pizzaNaam,  
Onderdelen = pizzaOnderdelen,  
Prijs = prijs  
});

}

### BinaryWriter – BinaryReader

Met een **BinaryWriter** object kan je gegevens van het Value type **char, byte, int, long, float, double,** **decimal**, **bool**, … en **arrays** van bytes of chars en **strings** binair wegschrijven naar een bestand.

Hierbij wordt ieder gegeven weggeschreven als een aantal bytes, afhankelijk van het gegevenstype:

* **int**: 4 bytes
* **double**: 8 bytes
* **string** VDAB: 5 bytes (UTF-8 encoding): naast het aantal tekens wordt ook de lengte van  
  de string weggeschreven om deze string achteraf terug te kunnen inlezen
* …

***using (BinaryWriter schrijver = new BinaryWriter(File.Open(@"C:\Data\Lottogetallen.bin", FileMode.Create)))***

*{*

*schrijver.Write(aantalGetallen);  
schrijver.Write(tekst);  
schrijver.Write(lottogetallen);  
schrijver.Write(winst);  
schrijver.Write(datum);*

*}*

Je creëert een **BinaryWriter** **object** waarmee je gegevens binair kan wegschrijven naar een bestand. Dit bestand wordt onder de vorm van een **FileStream** object als **parameter** van de constructor meegegeven. Je kan een FileStream object creëren met de method **Open()** van de class File: File.Open(@"C:\Data\Lottogetallen.bin", FileMode.Create))). Bij deze method wordt de naam (+ de locatie) van het bestand meegegeven (@"C:\Data\Lottogetallen.bin") en de manier waarop het bestand geopend wordt: er wordt een **nieuw** **bestand** **gecreëerd**. Als het bestand reeds bestaat, wordt het overschreven (FileMode.Create).

Om gegevens vanuit een binair bestand terug in te lezen gebruik je een **BinaryReader** object.

***using (BinaryReader lezer = new BinaryReader(File.Open(@"C:\Data\Lottogetallen.bin", FileMode.Open)))***

*{*

*aantalGetallen = lezer.ReadInt32();  
tekst = lezer.ReadString();  
lottogetallen = lezer.ReadBytes(7);  
winst = lezer.ReadSingle();  
datum = lezer.ReadString();*

*}*

Je creëert een BinaryReader object waarmee je gegevens kan inlezen uit een binair bestand. Dit bestand wordt onder de vorm van een FileStream object als parameter van de constructor meegegeven.  
Het FileStream object wordt gecreëerd met de method Open() van de class File:  
**File.Open**(@"C:\Data\Lottogetallen.bin", FileMode.Open))). Bij deze method wordt de naam (+ de locatie) van het bestand meegegeven (@"C:\Data\Lottogetallen.bin") en de manier waarop het bestand geopend wordt. Het bestand moet bestaan **(FileMode.Open).**

De class BinaryReader bevat verschillende methods om de respectievelijke gegevenstypes in te  
lezen: ReadInt32(), ReadSingle(), ReadString(), …  
**Vermits bij het binair wegschrijven van een string de lengte van de string ook weggeschreven  
wordt, kan de string achteraf perfect terug ingelezen worden.**

## Objecten wegschrijven – Serialization en Inlezen - deserialization

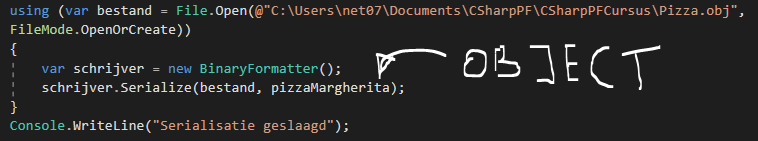
using System.Runtime.Serialization.Formatters.Binary;  
using System.Runtime.Serialization;

Een object wordt weggeschreven als een **stream van bytes**. Het wegschrijven van objecten wordt **serialisatie** genoemd. Het terug inlezen **deserialisatie**. De zogenaamde *state* van het object wordt weggeschreven: de waarde, het type van elke membervariabele en van het object.

**LET OP:** de class moet serialiseerbaar zijn. Dit doet men door de class te voorzien van het attribuut [***Serializable]***. Default worden alle public en private properties geserialiseerd.

[Serializable]  
public class Pizza  
{ }

* Als je objecten van een afgeleide klasse wil serialiseren, moet de base class ook serialiseerbaar zijn.
* Wachtwoorden of kredietkaartnummers kan je het attribuut [NonSerialized] geven.
* Properties die als type een reference type hebben moeten eveneens serialiseerbaar gemaakt worden (indien nodig).



* Creëer een **Pizza** **object**.
* Creëer een **FileStream** **object** waaraan het bestand, waarin je het Pizza object wil opslaan, gekoppeld wordt. Dit FileStream object wordt bij het serialiseren gebruikt.
* Om objecten naar een bestand weg te schrijven (serialiseren) heb je een **BinaryFormatter** object nodig. Dit object wordt hier gecreëerd.
* Met de method **Serialize()** van de class BinaryFormatter schrijf je het object weg naar het bestand. Het FileStream object (bestand) − en dus het gekoppelde bestand (Pizza.obj) − en het Pizza object (pizzaMargherita) worden als parameters aan deze method meegegeven.

In Notepad zal je merken dat de gegevens die met een *BinaryFormatter* zijn weggeschreven, niet echt leesbaar zijn.

using (var bestand = File.Open(@"C:\Data\Pizza.obj", FileMode.Open, FileAccess.Read))

{

var lezer = new BinaryFormatter();

Pizza pizza;

pizza = (Pizza)lezer.Deserialize(bestand);

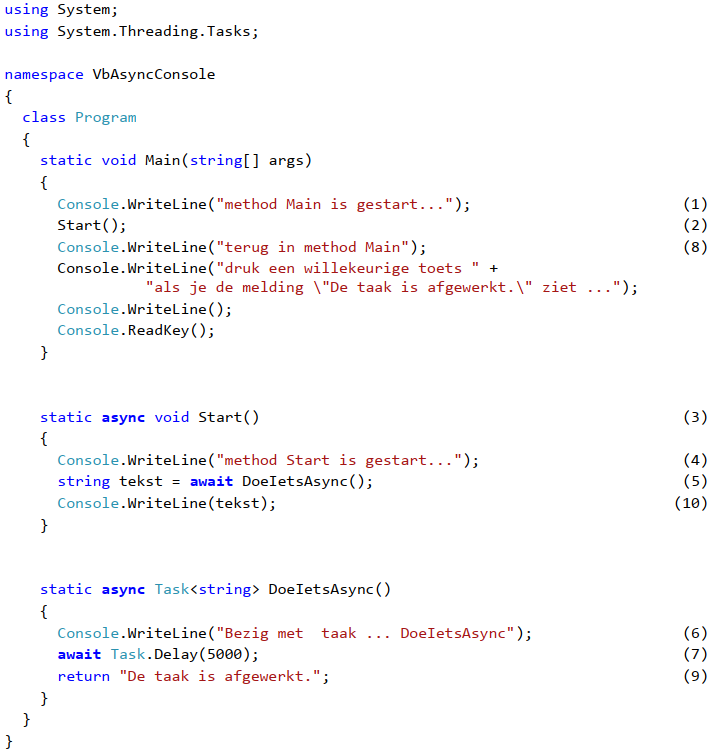
Console.WriteLine(pizza.Naam);

}

**LET OP:** *Deserialize()* geeft het ingelezen object als een type *Object* terug. (*Pizzas* als het om List object gaat)

**LET OP:** Je kan ook **meerdere objecten** tegelijk wegschrijven of lezen, verander het object *pizza* door een List van *Pizza* objecten.

# Asynchrone methods – async – await





! Schrijf je voor een method *async* maar gebruik je geen *await*, dan zal de method gewoon synchroon uitgevoerd worden.

(3) De method *Start()* heeft het sleutelwoord ***async*** om aan te geven dat we in deze method het sleutelwoord ***await*** gaan gebruiken om een andere method – *DoeIetsASync()* – **asynchroon** op te roepen.

(6) Door ***await*** te vermelden bij de oproep van *Task.Delay()*, krijgt de method *Start()* wel terug de programmacontrole terwijl *Task.Delay()* nog bezig is. m.a.w. het programmaverloop wordt ook hier niet geblokkeerd. Vermits *Start()* moet wachten op het resultaat van *DoeIetsAsync()*, gaat de programmacontrole wel terug naar de method *Main()*.

**OPMERKINGEN:**

* De **naam** van een asynchrone method kan best eindigen op **Async**.
* Een asynchrone method moet als **returntype** **Task<TResult>** of **Task** hebben (void kan ook, best enkel bij event handlers): *static async* ***Task<string>*** *DoeIetsAsync()* waarbij **string** staat voor het eigenlijke type dat wordt teruggegeven. Een **Task object** bevat methods en properties om de status van de operatie te monitoren. In zijn *Result* property zit de eigenlijke returnwaarde.
* Het sleutelwoord **await** zorgt ervoor dat er automatisch een casting gebeurt van het *Task<string>* object naar een string object. Je zou ook kunnen schrijven.
* Ook **lambda expressies en anonymous methods kunnen met async gemarkeerd worden.** Maar ze zullen pas vanaf een *await* statement uitgevoerd worden. Zonder deze await worden deze methods synchroon uitgevoerd.
* Een method gemarkeerd met async kan **meerdere await statements** bevatten.
* Het .NET Framework bevat reeds een groot aantal asynchrone methods. Je herkent deze aan het suffix Async. O.a. bij de classes **StreamReader**en **StreamWriter**

# Bijlage 1: Debugging

Programmacode fouten (syntaxfouten), runtime fouten (crash, hoofdstuk exceptions) of logische fouten (verkeerde resultaten).

Je kan de code stap voor stap uitvoeren dmv **DEBUG – Step Into** of **F11**. De programma houdt dan onmiddelijk stand bij de eerste accolade van *Main()*. Je kan steeds de waarde van een variabele bekijken door te hoveren over de naam. Met **DEBUG – Step Over** of **F10** wordt de opgeroepen method volledig uitgevoerd. Met **DEBUG – Step Out** kan je uit een method gaan. Door een rode stip aan te duiden voor een coderegel zal de volgende keer het debuggen de code uitvoeren tot aan die regel. Dit noemt men een **breakpoint**. Net hetzelfde met **Run To Cursor**.

Het ***Locals***venster toont alle variabelen tijdens het debuggen. In het ***watch*** venster kan je zelf expressies ingeven die worden opgevolgd tijdens het uitvoeren en in het laatste venster, het ***call stack*** venster, kan je alle methods zien die in uitvoering zijn. De method met een gele pijl is degeen waarvan de variabelen in het *locals* en *watch* venster worden getoond. Met een simpele dubbelklik kan men de variabelen van een andere method bekijken.

# Bijlage 2: PROCESSEN en THREADS

Ieder proces (programma) heeft meerdere threads (**multithreading**). Meerdere computerprocessoren handelen verschillende threads af (of processen = **multiprocessing**). Deze doen dit ondermeer via **timeslicing**.