



samen sterk voor werk

C# 2013

Programming Fundamentals Cursus

Deze cursus is eigendom van VDAB Competentiecentra ©

[Opmerkingen]

versie: 30/11/2014

INHOUD

1	Inleiding	11
1.1	C# - .NET Framework.....	11
1.2	Visual Studio 2013	12
1.3	.NET Framework versies.....	12
1.4	De cursus	13
2	Visual Studio 2013 starten.....	14
3	De opbouw van een applicatie in VS.NET	18
3.1	Project - Solution	18
3.2	Projecttypes.....	19
4	Een eerste programma	24
4.1	Een nieuw project.....	24
4.2	De map References	26
4.3	Assembly	27
4.4	Opbouw van het project	29
4.5	Tekst op het scherm afbeelden.....	32
4.6	Het programma compileren en uitvoeren	32
4.7	Methods van de class Console	33
4.8	Gebruik van de Help-functie	35
4.9	Commentaar.....	37
4.10	Een opdracht over meerdere regels uitsplitsen.....	38
4.11	De VS IDE	39
5	Lokale variabelen	45
5.1	Algemeen.....	45
5.2	Naamgeving.....	45
5.3	Types variabelen.....	45

5.4	Declaratie	47
5.5	Variabelen invullen	48
5.6	Variabelen tegelijkertijd declareren en invullen	50
5.7	Typeconversies van Value types	51
6	Constanten	54
6.1	Algemeen	54
6.2	Syntax	54
6.3	Voorbeeld	55
7	Bewerkingen	56
7.1	Bewerkingen op getallen	56
7.2	Verkorte bewerkingen	57
7.3	Met één verhogen of verlagen	58
7.4	Vergelijgingsoperatoren	59
7.5	Bewerkingen op bool waarden	60
7.6	Prioriteiten van bewerkingen	61
7.7	Wiskundige constanten en methods	62
8	Het type DateTime	64
8.1	Algemeen	64
8.2	Datums vergelijken	65
8.3	Methods en properties	66
8.4	Veelgebruikte methods en properties bij DateTime variabelen	67
9	Het type string	69
9.1	Algemeen	69
9.2	Bewerkingen op strings	70
9.3	Strings vergelijken	72
9.4	Veelgebruikte methods en properties bij string variabelen	73

9.5	De individuele tekens van een string	75
9.6	Value types converteren naar string	75
9.7	Strings converteren naar Value types	76
9.8	Speciale tekens in een string	77
9.9	Verbatim strings	78
9.10	Een lege string	79
9.11	Een samengestelde string afbeelden	80
10	Invoer vanaf het toetsenbord	82
10.1	Console.ReadLine()	82
11	Selecties	83
11.1	if	83
11.2	if ... else	84
11.3	De ? operator	86
11.4	switch	87
12	Iteraties	90
12.1	while	90
12.2	for	91
13	Declaratieplaats van variabelen in een method	95
14	Arrays	96
14.1	Eén-dimensionele arrays	96
14.2	De elementen van een array initialiseren bij de declaratie	98
14.3	foreach	99
14.4	Het aantal elementen van een array als variabel geven	99
14.5	Twee-dimensionele arrays	100
14.6	De class Array	101
15	Enum	103

15.1	Algemeen	103
16	Classes, Objects en object variables	105
16.1	Algemeen	105
16.2	Class (Klasse)	105
16.3	Object	105
16.4	Reference Variabele	105
16.5	Een class creëren	106
16.6	De Class View	107
16.7	Een property aan de class toevoegen	107
16.8	Properties met mixed access levels	113
16.9	Reference variabelen declareren en objecten aanmaken	113
16.10	null	115
16.11	Declaratie en objectaanmaak in één opdracht	116
16.12	Meerdere verwijzingen naar één object	117
16.13	Een array van references	118
16.14	Garbage collection	119
16.15	ReadOnly en WriteOnly properties	119
17	Methods	121
17.2	Methods met een parameter	122
17.3	Methods met meerdere parameters	123
17.4	Methods creëren in het Class Diagram	124
17.5	Overloading	125
17.6	Named parameters	127
17.7	Optionele parameters	128
17.8	Parameters met het sleutelwoord ref	130
17.9	Parameters met het sleutelwoord out	131

17.10	Methods met een resultaatwaarde	132
17.11	Een toepassing: TryParse()	134
18	Constructors.....	136
18.1	Algemeen.....	136
18.2	Default constructor	138
18.3	Constructor(s) met parameters.....	139
18.4	this	140
18.5	Constructor chaining	142
18.6	Destructor.....	142
18.7	Classes zonder default constructor	143
18.8	Overzicht van constructors	145
19	Static members	146
19.1	Static variabelen, properties en methods.....	146
19.2	Static constructor	147
19.3	Static classes.....	148
20	Inheritance	150
20.1	Algemeen.....	150
20.2	Overriding.....	151
20.3	Inheritance en constructors	153
20.4	Overzicht van inheritance en constructors	155
20.5	Andere voorbeelden van Inheritance	155
20.6	De class Object	158
20.7	is	162
20.8	as	163
21	Abstract classes en sealed classes, abstract methods.....	166
21.1	Abstract classes	166

21.2	Sealed classes.....	167
21.3	Abstract members	168
22	Inheritance polymorphism	171
22.1	Algemeen	171
23	Aggregation.....	173
23.1	Algemeen	173
24	Nested en partial classes	177
24.1	Nested classes.....	177
24.2	Partial classes.....	180
25	Interfaces	182
25.1	Algemeen	182
25.2	Een interface ontwerpen	183
25.3	Een interface implementeren.....	184
25.4	Interface polymorphism	188
25.5	is	191
26	Namespaces	193
26.1	Algemeen	193
26.2	Geneste namespaces.....	193
26.3	Naamafspraken.....	194
26.4	Standaard namespace	194
26.5	Namespaces in de sources.....	194
26.6	using	197
26.7	Oplossing naamconflicten	197
26.8	Alias.....	199
27	Delegates en events	200
27.1	Delegates	200
27.2	Events.....	205

27.3	Anonymous methods	211
28	Exceptions	215
28.1	Algemeen.....	215
28.2	De class Exception	215
28.3	Exceptions opvangen met try - catch.....	215
28.4	Meerdere catch opdrachten	217
28.5	Meerdere try opdrachten	219
28.6	Een fout melden met throw	220
28.7	Een eigen exception class ontwerpen.....	222
28.8	Gegarandeerde tijdige opkuis van resources.....	225
29	Operator overloading	233
29.1	Algemeen.....	233
29.2	De class Breuk.....	233
29.3	De vergelijkingsoperatoren	235
29.4	De wiskundige operatoren met twee waarden	237
29.5	De operatoren die met één verhogen of verlagen (++,--).....	238
29.6	De verkorte operatoren (+=, -=, *=, /=, %=)	240
29.7	Conversie operatoren.....	240
29.8	De operatoren true en false	242
30	Indexers.....	245
30.1	Algemeen.....	245
31	Collections en Generics	249
31.1	Collections	249
31.2	Generic lists	251
32	Nullable types.....	253
32.1	Definitie	253

32.2	Een nullable type gebruiken	253
32.3	De operator ??	254
33	Extra taalelementen	255
33.1	type inference - Implicitly typed local variables	255
33.2	Extension methods	258
33.3	Automatic properties – Auto-Implemented properties	261
33.4	Object initializers	263
33.5	Collection initializers	265
33.6	Anonymous types	266
33.7	Partial methods	267
33.8	Lambda (λ) expressions	270
33.9	Action en Func delegates	279
34	LINQ - een introductie	283
34.1	LINQ - Extension methods - Lambda Expressies	283
34.2	LINQ - Query Comprehension Syntax/Query Expression Syntax	287
34.3	LINQ - Voorbeelden	291
35	Files and Streams – I/O	309
35.1	Streams – FileStream	309
35.2	Bestanden en directories	311
35.3	Schrijvers – Lezers	315
35.4	Objecten wegschrijven – Serialization	326
36	Asynchrone methods – async – await	332
36.1	Asynchroon programmeren – wat en waarom?	332
36.2	Synchrone code	334
36.3	Asynchrone code – async – await – Task	336
36.4	Asynchrone methods in het .NET Framework	341

37	De .NET softwarelibrary.....	342
37.1	Algemeen.....	342
38	Bijlage 1: Debugging	344
38.1	Soorten fouten	344
38.2	Een programma stap voor stap uitvoeren – Step Into.....	346
38.3	Breakpoints.....	348
38.4	Het Locals venster	348
38.5	Het Watch venster.....	349
38.6	Het Call Stack venster.....	349
39	Bijlage 2: PROCESSEN en THREADS.....	350
39.1	Processen en threads	350
39.2	Het verdelen van threads over processoren.....	350
40	Colofon.....	352

1 Inleiding

1.1 C# - .NET Framework

C# is een objectgeoriënteerde programmeertaal waarmee professionele applicaties ontwikkeld kunnen worden, gebaseerd op de .NET technologie van Microsoft. Deze toepassingen draaien op een Windows platform (Windows, Windows Server, Windows Phone, Windows Azure).

Met C# kan je zowat alle types moderne applicaties ontwikkelen:



Het zijn zowel applicaties voor standalone computers, tablets, smartphones als gedistribueerde netwerkapplicaties die op meerdere computers draaien en via een netwerk of het internet onderling communiceren.

Binnen de .NET technologie zijn verschillende programmeertalen beschikbaar, o.a. Visual Basic.NET, C#. Al deze talen voldoen aan de zogenaamde Common Language Specification (CLS), een gemeenschappelijke basis die ervoor zorgt dat de talen met dezelfde gegevenstypen werken, op dezelfde manier routines oproepen, Hierdoor kan een applicatie geschreven worden in meerdere talen, en kunnen ontwikkelaars dus werken met de taal die ze het prettigst vinden.

Het **.NET Framework** is het hart van de .NET technologie. Deze bibliotheek van classes 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 het beheren van de .NET applicaties. Het framework zorgt er o.a. voor dat onderdelen van een applicatie, 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.

Het .NET Framework wordt samen met Windows geïnstalleerd.

Door de jaren heen zijn er verschillende versies van het .NET Framework verschenen. Indien bij de installatie van een .NET applicatie het vereiste .NET Framework niet aanwezig is, zal het programma vragen om het framework te installeren. Er kunnen verschillende versies van het .NET Framework naast elkaar op hetzelfde toestel geïnstalleerd zijn.

Raadpleeg best regelmatig het internet voor een stand van zaken betreffende nieuwe updates.

1.2 Visual Studio 2013

De beste ontwikkelomgeving om een C# applicatie te maken is **Visual Studio.NET (VS.NET)**.

Dit is een geïntegreerde ontwikkelomgeving (Integrated Development Environment of IDE), die alle hulpmiddelen bevat waarmee de programmeur op een relatief snelle, efficiënte en visuele manier applicaties kan schrijven, uitvoeren, testen en publiceren. In Visual Studio.NET kan je meerdere programmeertalen, waaronder C#, gebruiken.

Door de jaren heen zijn er eveneens verschillende versies van Visual Studio .NET verschenen, die elk hun eigen versie van het .NET Framework bevatten. Elke VS.NET omgeving ondersteunt echter meerdere versies van het .NET Framework zodat de ontwikkelaar niet beperkt is tot het gebruik van de meegeleverde versie: VS.NET is “multi-targeting”.

In deze cursus werken we met **Visual Studio 2013 (update 4)**. Raadpleeg regelmatig het internet voor updates van Visual Studio.

Je kan Visual Studio 2013 installeren op Windows 8.1, Windows 8, Windows 7, Windows Server 2008 en Windows Server 2012.

Er zijn verschillende edities van Visual Studio 2013 beschikbaar o.a. de Professional, Premium en Ultimate Editions. Dit zijn betalende versies.

Daarnaast zijn er ook een aantal express edities beschikbaar waarmee je gratis aan de slag kan. Van elke editie vind je een uitgebreide beschrijving op de website van Microsoft.

1.3 .NET Framework versies

Informatie over de verschillende versies van het .NET Framework en de correlaties met Windows, Windows Server en Visual Studio vind je in de **MSDN** via “.NET Framework Versions and Dependencies”.

De **MSDN** of het **Microsoft Developer Network** is de uitgebreide documentatiedienst van Microsoft ter ondersteuning van ontwikkelaars die met Microsoft producten werken.



Windows 8.1 en Visual Studio 2013 bevatten standaard het .NET Framework 4.5.1. Het meest recente framework op dit ogenblik is 4.5.2.

Bij de ontwikkeling van een nieuwe applicatie kan je zelf een framework versie kiezen (zie verder).

Houd hierbij rekening met het volgende:

- Een applicatie ontwikkeld voor framework versie 4.5, draait zonder problemen op elk toestel waarop framework 4.5 of hoger geïnstalleerd is.
- Een applicatie ontwikkeld voor framework versie 4.5.1, kan niet uitgevoerd worden op een toestel dat enkel framework 4.5 bevat. Er zal aan de gebruiker gevraagd worden om het framework 4.5.1 te installeren.
M.a.w. je kan applicaties uitvoeren op eerdere versies van Windows op voorwaarde dat het juiste .NET framework geïnstalleerd is.

1.4 De cursus

1.4.1 Doelstelling

In deze module leer je de basissyntax van C# en leer je de objectgeoriënteerde principes toepassen in C#, zodat je een objectgeoriënteerde *console-applicatie* kan ontwerpen.

1.4.2 Vereiste voorkennis

- Gestructureerde programmatielogica
- Objectgeoriënteerde principes (OOP)

1.4.3 Afspraken

- Soms zie je in de programmacode van de cursus het teken ∘. In je eigen programmacode moet je dit teken door een spatie vervangen.
- De term Visual Studio.NET wordt verder in de cursus vervangen door de afkorting VS.

1.4.4 Werkwijze

In de cursus zijn opdrachten voorzien die je in de bijhorende takenbundel terugvindt (symbool). Het is de bedoeling dat je deze oefeningen eerst zelf oplost en daarna vergelijkt met de modeloplossingen, die je eveneens in de takenbundel terugvindt.

Visual Studio bevat een zeer uitgebreide Help-functie, de MSDN (Microsoft Developer Network), waar je informatie over de .NET programmeertalen en de verschillende classes van het .NET Framework kan terugvinden. Het is uiteraard ondoenbaar om deze documentatie volledig te bestuderen, maar het is wel aangewezen om de MSDN regelmatig te raadplegen. Zo leer je geleidelijk aan je weg te vinden binnen deze documentatie.

2 Visual Studio 2013 starten

Start Visual Studio.NET 2013 via de start-knop van Windows 7:
start – All Programs – Visual Studio 2013 – Visual Studio 2013

of via een snelkoppeling op de desktop:



of via een tegel op het startscherm van Windows 8.1:



Wanneer Visual Studio 2013 **voor de eerste maal** wordt opgestart na de installatie, worden de volgende dialoogvensters getoond:

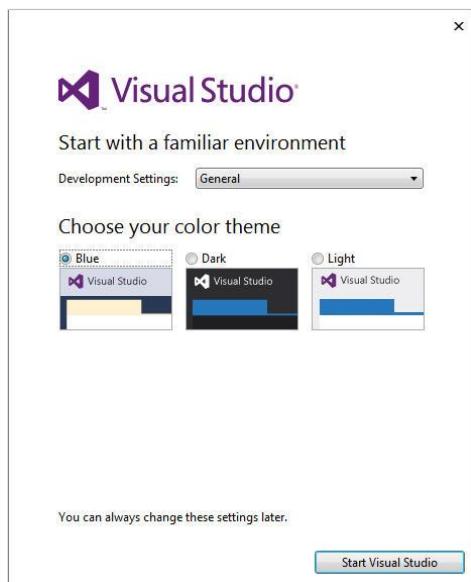
- Eerst wordt je gevraagd om optioneel aan te melden in Visual Studio met een Microsoft-account. Beschik je nog niet over een Microsoft-account, dan krijg je de gelegenheid om er één te creëren tijdens het aanmelden.



Aanmelden in Visual Studio biedt een aantal voordelen. Zo worden je persoonlijke voorkeursinstellingen van de ontwikkelomgeving automatisch ingesteld en gesynchroniseerd wanneer je je aanmeldt in Visual Studio op een andere computer.

Je hebt de keuze om je al dan niet aan te melden. Achteraf kan je nog altijd van gedacht veranderen en je keuze wijzigen via de link [Sign in](#) , rechts bovenaan in het scherm (zie verder).

- In een volgend dialoogvenster kan je een aantal voorkeuren instellen:



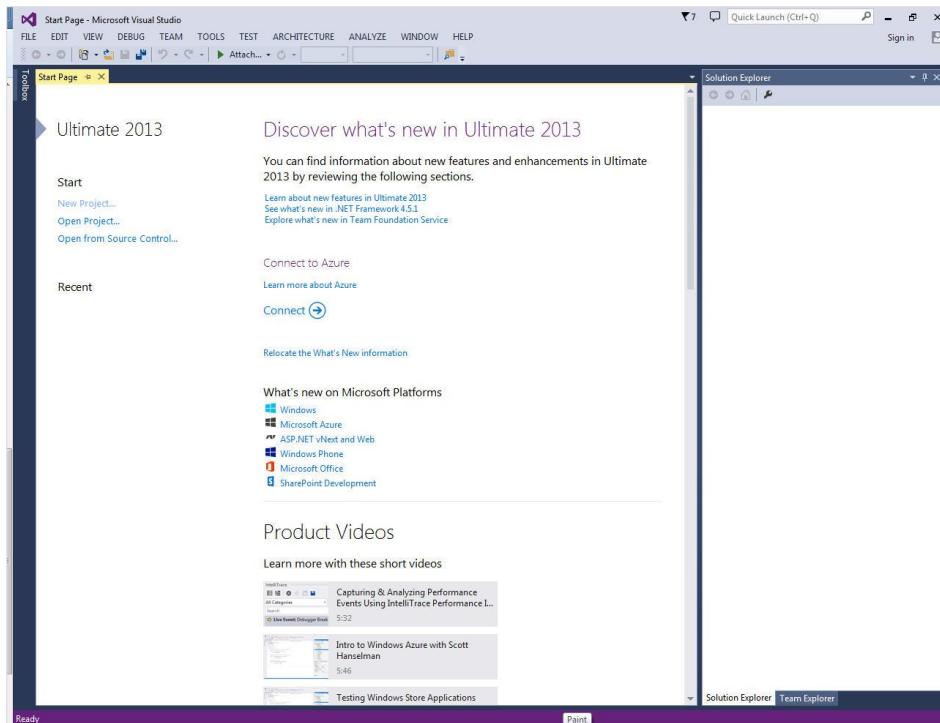
- Je kan in VS aangeven met welke instellingen je wil werken. Je kan kiezen voor algemene instellingen die onafhankelijk zijn van de gebruikte programmeertaal, of je kan de settings kiezen die afgestemd zijn op de programmeertaal waarin je gaat ontwikkelen.

Kies in de keuzelijst **Development Settings:** voor **Visual C#**.

Deze settings kan je indien nodig achteraf wijzigen via:

TOOLS – Import and Export Settings... en Reset all settings.

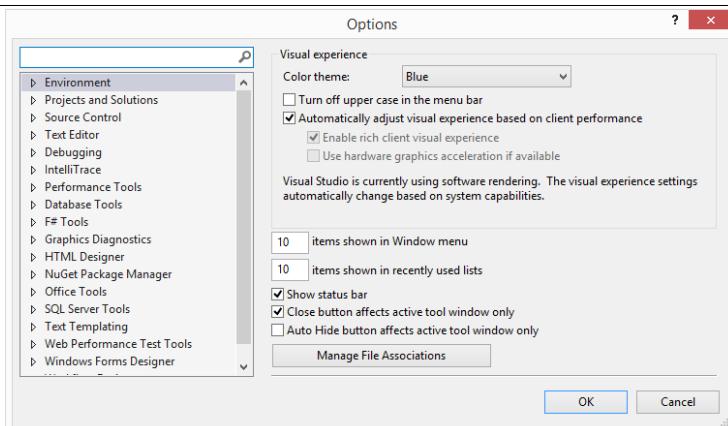
- Je kan hier eveneens een kleurenschema – Blue, Dark of Light – voor Visual Studio instellen.
 - Klik op de knop **Start Visual Studio** om Visual Studio effectief op te starten.
- Nu verschijnt het eigenlijke opstartscherms van Visual Studio.
Dit startscherm kan lichtjes afwijken van je eigen startscherms.



- Links wordt de **Start Page** getoond waarin je kan kiezen om een nieuw project te creëren of een bestaand project te openen. Verder word je op de hoogte gehouden van de laatste nieuwigheden van Microsoft en krijg je een aantal interessante links te zien.
- Rechts zie je de Solution Explorer. Deze komt verder in de cursus aan bod.
- Bemerk rechts bovenaan de link [Sign in](#) waarmee je je kan aanmelden in VS.NET. Indien je aangemeld bent, verschijnt hier de naam van je Microsoft-account.

 Wanneer Visual Studio in het vervolg opnieuw gestart wordt, zal enkel dit laatste opstartschermscherm worden getoond.

 Je kan achteraf heel wat voorkeursinstellingen van Visual Studio aanpassen via het menu **TOOLS – Options...**



Het *Color theme* kan je wijzigen via *TOOLS – Options...* en *Environment – General*.

Via *TOOLS – Options...* en *Environment – Synchronized Settings* kan je aangeven welke settings gesynchroniseerd mogen worden bij aanmelden in Visual Studio op een andere device.

Neem even de tijd om het dialoogvenster *TOOLS – Options...* te verkennen.

3 De opbouw van een applicatie in VS.NET

3.1 Project - Solution

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

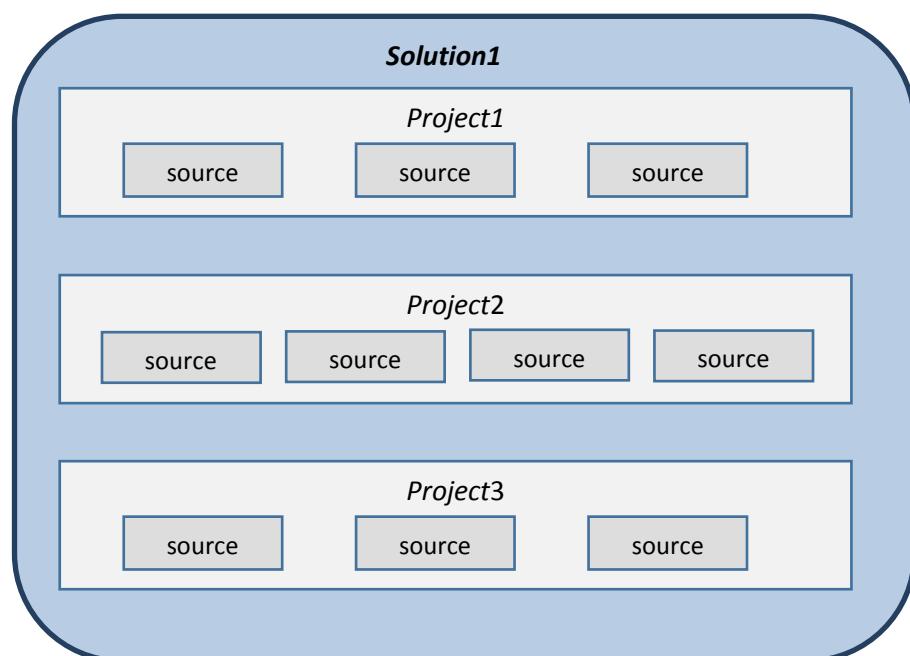
Een **project** is een applicatie die je met VS ontwikkelt. Het is een verzameling van gerelateerde bestanden waaronder de C# code zelf, eventuele afbeeldingen, gegevensbestanden, ... die samen de applicatie vormen en laten werken.

Een project is op zich een container met de nodige source-bestanden voor de applicatie. Er zijn verschillende soorten bronbestanden:

- Class (): een sjabloon voor zelfgedefinieerde objecten
- WPF Window (): lay-out en code van een Windows venster
- Web Form (): lay-out en code van een webpagina van de applicatie
- Folders ()
- XML-files ()
- ...

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. Op die manier kan je bvb. één project van de applicatie uitwerken in C#, een ander project met Visual Basic.NET. Hoewel de projecten in verschillende talen geschreven zijn, kan je vanuit het ene project code oproepen van het andere project. Dit komt omdat, zoals reeds eerder aangegeven, alle programmeertalen in .NET voldoen aan de CLS - de Common Language Specification.



Iedere solution krijgt een eigen directory. Zo behoud je overzichtelijkheid als je meerdere applicaties ontwerpt. 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*.

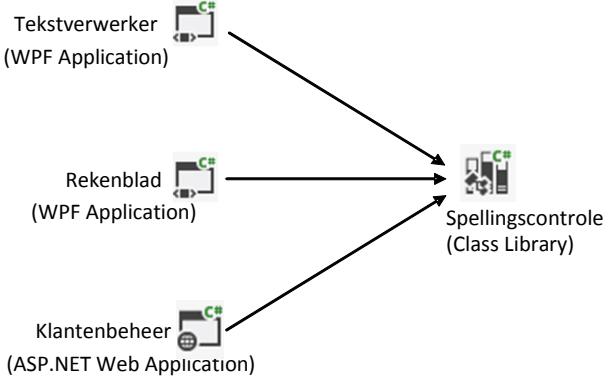
3.2 Projecttypes

Zoals reeds vermeld kan je met C# verschillende types moderne applicaties ontwikkelen. Naargelang het soort applicatie dat je wil maken, kies je een ander projecttype.

Een greep uit deze projecttypes:

Type applicatie	Omschrijving
Console Application 	Een eindgebruikersapplicatie met een <i>tekstgebruikersinterface</i> (witte letters op een zwarte achtergrond) = commandline-application. Het gecompileerde resultaat is een bestand met de extensie <i>exe</i> . De gebruiker start de console-applicatie door dit .exe bestand te starten.
Windows Client Application of Windows Desktop Application  	Een eindgebruikersapplicatie met een <i>grafische gebruikersinterface</i> (dialogvensters, menu's, knoppen, invoervakken, ...) die lokaal op een computer met Windows geïnstalleerd wordt. Het gecompileerde resultaat is een bestand met de extensie <i>exe</i> . De gebruiker start de windowsapplicatie door dit .exe bestand te starten. Voorbeelden van windowsapplicaties: Word, Excel, Visual Studio 2013, Photoshop, ... Voor de ontwikkeling van Windows Client Applications heb je de keuze tussen twee technologieën: <ul style="list-style-type: none"> • Windows Forms  • WPF  Een WPF Application is gebaseerd op het Windows Presentation Framework (WPF), het meest recente Microsoft platform om windowsapplicaties te ontwerpen. WPF maakt gebruik van een op XML gebaseerde taal – XAML (Extensible Application Markup Language) – om de lay-out van de gebruikersinterface vast te leggen.

<p>ASP.NET Web Application</p> 	<p>Een eindgebruikersapplicatie met webpagina's (HTML) als grafische gebruikersinterface, ook wel website genoemd. Het gecompileerde resultaat is een bestand met de extensie <i>dll</i>.</p> <p>Een webapplicatie draait op een Windows Webserver, de IIS of Internet Information Server. Tijdens de ontwikkeling kan je de webapplicatie uittekenen op de ingebouwde webserver van VS.NET.</p> <p>De gebruiker start de applicatie door met een browser naar de URL van de webapplicatie te surfen.</p> <p>Voor de ontwikkeling van webapplicaties zijn meerdere frameworks beschikbaar:</p> <ul style="list-style-type: none">• ASP.NET Web Forms<p>Dit framework bevat een verzameling controls (invoervakken, knoppen, keuzelijsten, ...) die je op een webform of webpagina kan slepen. Aan de bijhorende events wordt C# code gekoppeld die uitgevoerd wordt wanneer het event optreedt. Bvb. wanneer de gebruiker op een button klikt, wordt de code achter het Click event van deze knop uitgevoerd.</p>• ASP.NET MVC<p>Een ASP.NET MVC webapplicatie is gebaseerd op het Model-View-Controller design pattern (MVC). Het is een meer recent framework om webapplicaties te ontwikkelen.</p> <p>In één webapplicatie is het zelfs mogelijk de verschillende frameworks te combineren.</p>
--	---

Components and Controls	<p>Bibliotheken van herbruikbare componenten – al dan niet met een grafische interface – die in meerdere projecten kunnen gebruikt worden. VS.NET voorziet hiervoor verschillende projecttypes:</p> <ul style="list-style-type: none"> • Class Library  <p>Een verzameling classes zonder grafische gebruikersinterface. Het gecompileerde resultaat is een bestand met de extensie .dll. De gebruiker kan dit .dll bestand niet rechtstreeks starten. De classes van het .dll bestand kunnen wel gebruikt worden in de code van een .exe bestand.</p> <p>Een voorbeeld van een Class Library is de spellingscontrole. Als je deze uitwerkt als een Class Library, verkrijg je na compilatie een dll. Deze dll kan zowel opgeroepen worden vanuit een tekstverwerkingsprogramma dat je als WPF Application geschreven hebt, als vanuit een rekenbladprogramma dat je als WPF Application geschreven hebt. Ook een ASP.NET Web Application kan deze dll aanspreken. Een Class Library is dus interessant voor herbruikbare code:</p>  <pre> graph TD A[Tekstverwerker
(WPF Application)] --> C[Spellscontrole
(Class Library)] B[Rekenblad
(WPF Application)] --> C D[Klantenbeheer
(ASP.NET Web Application)] --> C </pre> <ul style="list-style-type: none"> • Portable Class Library  <p>In .NET beschikt elk type applicatie (WPF, Windows Phone, Windows Store app, ...) over een eigen framework. Hierdoor kan bvb. een WPF applicatie niet uitgevoerd worden op een Windows smartphone, daar het Windows Phone framework niet over de nodige WPF componenten beschikt. Dit betekent dat wanneer je dezelfde applicatie op meerdere windows platformen wil gebruiken, je meerdere versies van deze applicatie moet schrijven. Eén van de belangrijkste programmeerprincipes is echter het vermijden van dubbele code.</p> <p>Met het Portable Class Library projecttype kan je echter herbruikbare componenten ontwikkelen die in meerdere types .NET applicaties en op meerdere .NET platformen kunnen gebruikt worden.</p>
-------------------------	--

	<ul style="list-style-type: none"> Je kan de verzameling ingebouwde controls van WPF en Windows Forms uitbreiden met eigen controls door de ingebouwde controls uit te breiden (WPF Custom Control Library en Windows Forms Control Library) of volledig nieuwe controls te ontwerpen die samengesteld zijn uit meerdere ingebouwde controls (WPF User Control Library en Windows Forms Control Library).
Windows Service 	<p>Een verzameling classes zonder grafische gebruikersinterface. Het gecompileerde resultaat is een bestand met extensie exe. Een Windows Service loopt als een achtergrondproces in het Windows besturingssysteem en voert achtergrondtaken uit. Een Windows Service kan door de gebruiker gestart worden, of (wat meer voorkomt) kan starten zodra het Windows besturingssysteem start.</p>
Web Service	<p>Een web service is een applicatie zonder grafische gebruikersinterface, die diensten aanbiedt waarvan andere applicaties gebruik kunnen maken.</p> <p>Een web service maakt het mogelijk om op afstand – meestal via het internet – vanaf een client (bvb. een webapplicatie) een dienst op te vragen aan een server, bvb. het maken van een berekening, het opvragen en bijwerken van gegevens, het uitvoeren van een bepaalde taak ...</p> <p>Met web services kan je krachtige service-georiënteerde, gedistribueerde applicaties te ontwerpen.</p> <p>De methods van een web service kunnen opgeroepen worden vanuit de .NET programmeertalen (C#, VB.NET, ...) maar ook vanuit andere programmeertalen (Java, C++, COBOL, PHP, ...). Deze oproepen kunnen via een netwerk, via het intranet of via het internet gebeuren.</p> <p>C# voorziet twee technologieën om web services te ontwerpen:</p> <ul style="list-style-type: none"> WCF of Windows Communication Framework Meerdere communicatieprotocollen beschikbaar. Web API Hiervoor gebruik je het projecttype ASP.NET Web Application. De communicatie met dit type web service verloopt immers via het HTTP protocol.

ASP.NET Server Control 	<p>Een verzameling classes. Het gecompileerde resultaat is een bestand met extensie <i>dll</i>.</p> <p>Eén of meerdere van deze classes stellen een eigen geschreven herbruikbare control voor, die je gebruikt in een ASP.NET Web Application. Je kan bvb. een control maken die een taartgrafiek in de browser toont aan de hand van data die je vanuit je code aan de control meegeeft.</p>
Windows Phone App 	<p>Een applicatie voor een smartphone met het Windows Phone besturingssysteem. Windows Phone Apps kan je ontwikkelen met XAML/C#, maar enkel op een computer met Windows 8 of later.</p> <p>Deze apps worden gepubliceerd en zijn beschikbaar via de Windows Phone Store.</p> <p>Windows Phone apps kunnen ook ontwikkeld worden met HTML5/Javascript.</p>
Windows Store App 	<p>Een applicatie voor een computer of tablet met het Windows 8 of 8.1 besturingssysteem. Windows Store Apps kan je ontwikkelen met XAML/C#, maar enkel op een computer met Windows 8 of later.</p> <p>Deze apps worden gepubliceerd en zijn beschikbaar via de Windows Store.</p> <p>Windows Store apps kunnen ook ontwikkeld worden met HTML5/Javascript.</p>
Cloud based Application	<p>Met C# kan je eveneens applicaties ontwikkelen voor Windows Azure, het MS besturingssysteem voor de cloud. C# kan hierbij gebruik maken van SQL Azure, de cloud versie van Microsoft's SQL Server database server.</p>

C# kan ook gebruikt worden om grote, bedrijfskritieke applicaties te ontwikkelen, waarbij gegevens kunnen uitgewisseld worden via SQL Server, Sharepoint, Office, ...

M.a.w. de mogelijkheden van C# zijn nagenoeg onbegrensd.

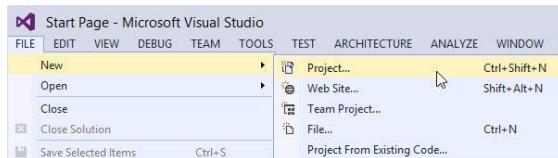


In deze cursus beperken we ons tot de ontwikkeling van **Console Applications**. Andere projecttypes komen in de volgende modules van het opleidingstraject aan bod.

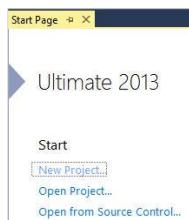
4 Een eerste programma

4.1 Een nieuw project

Kies in het opstartscherms van VS het menu **FILE – New – Project...**

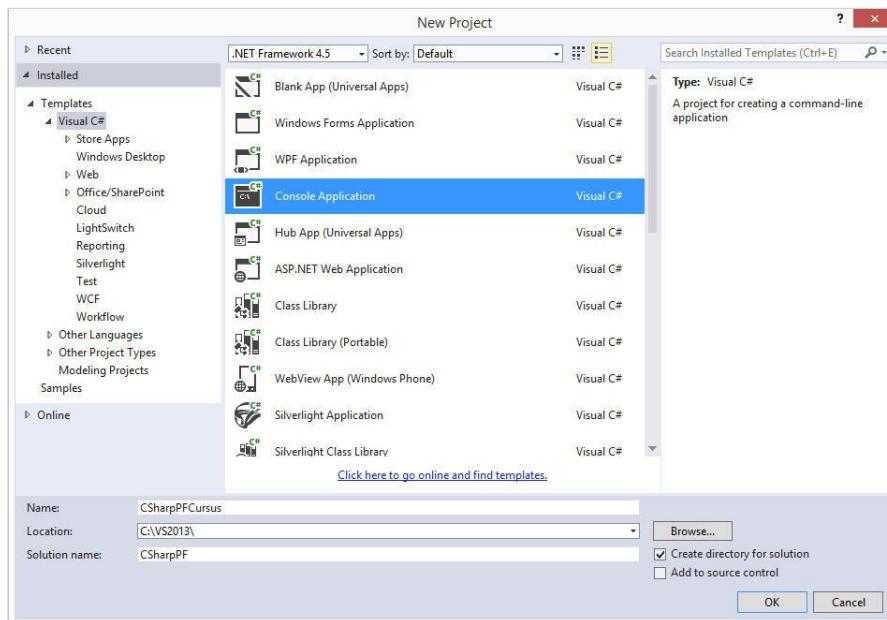


of klik op de link **New project ...** in de Start Page:

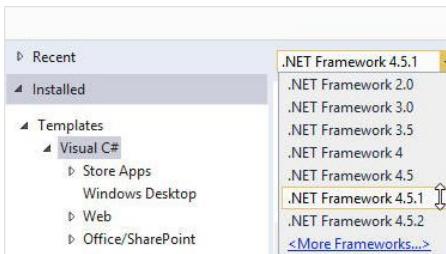


In het volgende dialoogvenster kan je nu als volgt een nieuw project aanmaken:

- Kies bij Installed onder Templates voor Visual C# en in het midden de template Console Application.

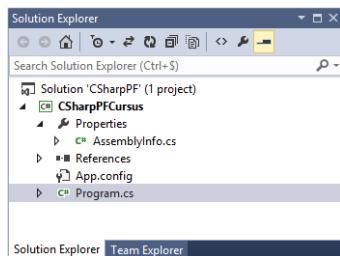


- Bovenaan in het dialoogvenster kan je aangeven welke versie van het .NET Framework je wil gebruiken bij het ontwikkelen van de applicatie. Hoe hoger de versie, hoe recenter het framework en hoe meer mogelijkheden je ter beschikking hebt. Je keuze wordt vooral bepaald door het besturingssysteem waarop de applicatie moet draaien. Het gebruikte Framework moet immers beschikbaar zijn voor dit besturingssysteem.
Kies hier voor **.NET Framework 4.5.1**:



- Typ bij *Name* de naam van het project: *CSharpPFCursus*.
- Typ bij *Solution Name* de naam van de solution: *CSharpPF* en vink de optie *Create directory for solution* aan.
- Kies bij *Location* de directory waarin VS een subdirectory *CSharpPF* voor de solution aanmaakt, met daarin een subdirectory *CSharpPFCursus* voor het project.
- Kies *OK*.

In de *Solution Explorer*, rechts op het scherm, zie je het nieuwe project dat alvast bestaat uit een source file (*Program.cs*), een folder *Properties*, een folder *References* (zie verder) en een configuratiebestand *App.config*.

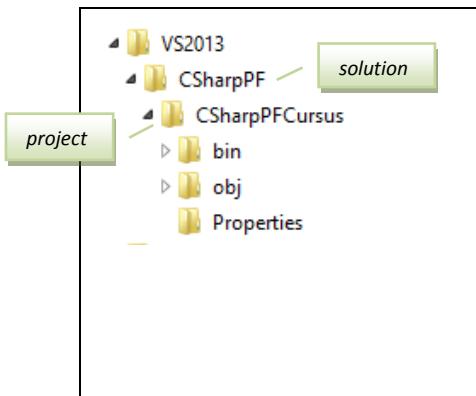
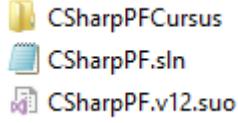
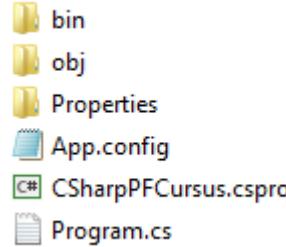


 Als je de *Solution Explorer* niet ziet, maak je deze zichtbaar via het menu *VIEW – Solution Explorer*.

 Als je in de *Solution Explorer* de naam van de solution niet ziet, bvb. als deze slechts één project bevat, dan schakel je via het menu *TOOLS – Options – Projects and Solutions – General* de optie *Always show solution* in

	Een alternatieve werkwijze is dat je eerst een lege solution aanmaakt via het menu <i>FILE – New – Project...</i> en <i>Installed</i> onder <i>Templates, Other Project Types, Visual Studio Solutions, Blank Solution</i> . Daarna kan je een project toe aan deze lege solution toevoegen via het menu <i>FILE – Add – New Project...</i> of door met de rechtermuisknop in de <i>Solution Explorer</i> op de naam van de solution te klikken en vervolgens te kiezen voor <i>Add – New Project...</i>
---	---

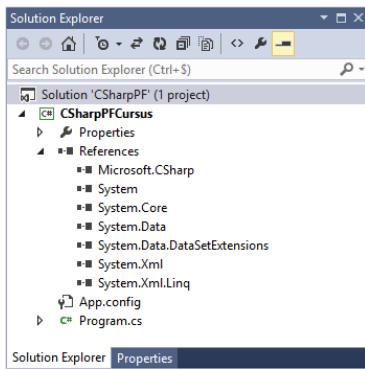
In de *Windows Explorer* kan je de mappen van de solution en het project, met de bijhorende bestanden bekijken.

	de inhoud van de solution folder: 	de inhoud van de project folder: 
--	--	---

4.2 De map References

Je ziet in de Solution Explorer in het project *CSharpPFCursus* een map *References*. Dit zijn verwijzingen naar codebibliotheeken (DLL bestanden) die je vanuit jouw programma kan aanspreken. Zoals reeds eerder vermeld bevat het .NET Framework een grote verzameling van classes die op basis van hun functionaliteit verder gegroepeerd worden in verschillende bibliotheken. D.m.v. references kan je aangeven van welke bibliotheken jouw programma gebruik maakt.

Standaard bevat elk nieuw project reeds enkele verwijzingen naar DLL's, afhankelijk van het projecttype:



- *Microsoft.Csharp* en *System*: bibliotheken met basisfunctionaliteit van .NET.
- *System.Data*: een bibliotheek die toegang tot databases voorziet.
- *System.XML*: een bibliotheek die toegang tot XML bestanden voorziet.

System.Core, *System.Data.DataSetExtensions* en *System.Xml.Linq* zijn pas beschikbaar vanaf het Framework 3.5.

Je kan indien nodig extra References naar andere codebibliotheeken aan een project toevoegen.

4.3 Assembly

De broncode van een applicatie – de instructies die het programma moet volgen en uitvoeren – wordt door de programmeur geschreven m.b.v. de programmeertaal C#, een tekstgebaseerde taal die echter niet door een computer kan gelezen worden. Een computer kan enkel nullen (0) en enen (1) begrijpen en verwerken. Vandaar dat de broncode eerst moet vertaald worden naar een voor de computer begrijpelijke code. Deze vertaling gebeurt door een zogenaamde compiler.

Een Assembly is de gecompileerde versie van een project en is verpakt in een bestand met extensie **EXE** of **DLL**.

In .NET is het echter zo dat de broncode eerst naar een “tussentaal” gecompileerd wordt, de zogenaamde *Common Intermediate Language of CIL* (soms ook gewoon *IL* genoemd), vooraleer de code effectief uitgevoerd wordt. Deze binaire taal bestaat uit processorinstructies die niet aan één bepaalde CPU gebonden zijn. Een processor kan deze gecompileerde code ook niet rechtstreeks uitvoeren.

Om deze CIL-code dan daadwerkelijk te kunnen uitvoeren, moet de CLR = Common Language Runtime (onderdeel van het .NET Framework) geïnstalleerd zijn op de betreffende computer. Zonder CLR op je computer, kan je geen .NET-applicatie laten draaien. Pas als het programma uitgevoerd wordt, wordt de CIL-code door de JIT Compiler (Just In Time Compiler, ook een onderdeel van het .NET Framework) omgezet naar processorinstructies. Daarna voert de processor deze processorinstructies uit.

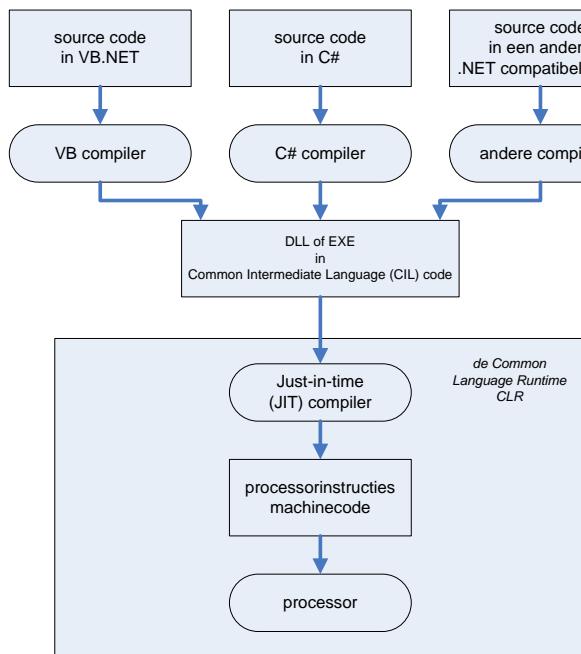
Alle programmeertalen, die voldoen aan de Common Language Specification (CLS), worden door de .NET compiler naar **dezelfde** CIL-code vertaald. De CLS is een soort contract waaraan de programmeertalen moeten voldoen, zodat een component, geschreven in de ene taal, kan gebruikt worden in alle andere talen die dit contract respecteren. Vandaar dat de onderdelen van een .NET-applicatie in verschillende (.NET compatibele) programmeertalen mogen geschreven worden. Zij worden uiteindelijk toch gecompileerd tot dezelfde CIL-code. Zo kan bvb. een webpagina die in C# geschreven is, gebruik maken van een VB.NET component en omgekeerd.

Nadeel van deze werkwijze is dat het programma trager is: bij iedere uitvoering van de applicatie vertaalt de JIT Compiler de CIL-code naar processorinstructies, wat tijd vraagt.

Voordeel is dat de JIT Compiler de vertaling kan optimaliseren voor de computer waarop je het programma uitvoert: het soort processor, het aantal processoren, de hoeveelheid geheugen, ... om een snellere uitvoering te bekomen. De resulterende machinecode is aangepast aan de processor, aan het besturingssysteem m.a.w. aan het platform waarop de applicatie moet draaien.

Ook vertaalt de JIT Compiler de CIL-code pas als het programma de code nodig heeft. Zo wint de JIT Compiler tijd: delen van het programma die niet uitgevoerd worden, moet de JIT Compiler niet vertalen. De JIT Compiler houdt bovendien de vertaalde code bij in een cache in het intern geheugen. Als het programma dezelfde code later opnieuw uitvoert, moet de vertaling niet meer gebeuren, maar haalt de JIT Compiler de vroeger bijgehouden code op uit de cache en voert deze onmiddellijk uit.

De volgende figuur geeft dit schematisch weer:



In de Solution Explorer zie je bij het project in de folder *Properties* een source *AssemblyInfo.cs*.

Als je dubbelklikt op deze source, zie je dat deze informatie bevat over de Assembly:

```

using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("CSharpPFCursus")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CSharpPFCursus")]
[assembly: AssemblyCopyright("Copyright © 2014")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("830ddb98-2644-4d93-80b8-7863e6e6ac9c")]

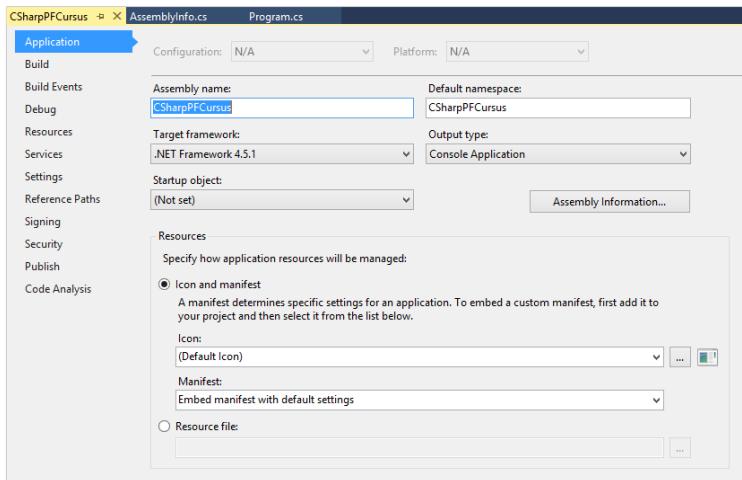
// Version information for an assembly consists of the following four values:
//
// Major Version
// Minor Version
// Build Number
// Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]

```

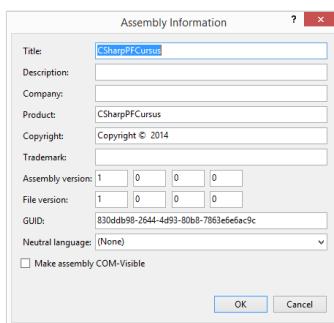
Je kan bvb. bij *AssemblyCopyright* tussen de dubbele aanhalingstekens je Copyright tekst tikken. Bij *AssemblyVersion* zie je tussen dubbele aanhalingstekens het versienummer van jouw project.

Deze info vind je ook terug via de properties van het project:

- Klik in de *Solution Explorer* met de rechtermuisknop op de naam van het project en kies uit het snelmenu de optie *Properties*:



- Kies links voor *Application* en klik vervolgens op de knop *Assembly Information...*



Sluit dit dialoogvenster.

 In het dialoogvenster van de project properties kan je bij *Target framework*: een andere frameworkversie kiezen indien nodig.

4.4 Opbouw van het project

Het project bevat een source voor een eerste class (*Program.cs*).

Eén source kan één of meerdere classes bevatten.

Als je in de Solution Explorer dubbelklikt op *Program.cs*, zie je de opbouw van de source:

```
using System; (1)
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPFCursus (2)
{ (3)
    class Program (4)
```

```

{
    static void Main(string[] args)
    {
    }
}

```

(5)
(6)
(7)
(8)
(9)
(10)



C# is een hoofdlettergevoelige taal. Als je bijvoorbeeld het sleutelwoord `class` tikt als `Class`, geeft dit een fout tijdens het compileren. De variabele `leeftijd` is bijvoorbeeld een andere variabele dan de variabele `Leeftijd`.

- (1) Als je een applicatie schrijft, wil je zoveel mogelijk code gebruiken uit de codebibliotheeken die Microsoft meelevert met .NET. Zo moet je zelf minder code typen en bedenken.
De codebibliotheeken zijn zeer uitgebreid en bevatten dus veel classes. Hoe meer classes je hebt, hoe meer kans er is op naamconflicten: bvb. twee keer dezelfde naam gebruiken voor een verschillende class. De class `Connection` zou bijvoorbeeld twee keer in de codebibliotheeken kunnen voorkomen: één keer om een databaseverbinding te beschrijven en één keer om een netwerkverbinding te beschrijven.
Om deze naamconflicten te vermijden heeft Microsoft de classes die bij elkaar horen, verzameld in een zogenaamde namespace. Een namespace is een woord (of combinatie van woorden) die je met een verzameling classes associeert.
Zo kan je in het voorbeeld de class `Connection` die de databaseverbinding voorstelt, in een namespace met de naam `Database` plaatsen.
Je kan de class `Connection` die de netwerkverbinding voorstelt, in een namespace met de naam `Network` plaatsen.
Als je in een applicatie een databaseverbinding én een netwerkverbinding nodig hebt, kan je de classes als volgt onderscheiden: je tikt de *namespace* van de class, gevolgd door een *punt*, gevolgd door de *naam van de class*. Voor de databaseverbinding tik je dus `Database.Connection` en voor de netwerkverbinding tik je `Network.Connection`.
Als je in een applicatie enkel netwerkverbindingen nodig hebt, is het wel vervelend dat je iedere keer de namespace én de class moet vermelden. Dit kan je vermijden door boven in de source één keer aan te geven dat je classes gebruikt uit de namespace `Network`.
Je geeft dit aan met het sleutelwoord `using` gevolgd door de naam van de namespace.
De opdracht `using System;` in deze eerste applicatie geeft dus aan dat je in deze applicatie classes wil aanspreken uit de namespace `System`, zonder voor iedere class van deze namespace `System`. te moeten schrijven. De namespace `System` bevat classes die je regelmatig nodig hebt, zoals de class `Console`, waarmee je tekst op het scherm toont. Zoals de meeste opdrachten sluit je de `using` opdracht af met een puntkomma.
Standaard worden de namespaces `System.Collections.Generic`, `System.Text`, `System.Linq` en `System.Threading.Tasks` eveneens geïmporteerd.
- (2) De class `Program` bevat een method `Main()`. Als de applicatie start, start ze met deze method. De class `Program` is dus de class die het opstartpunt (entry point) van de applicatie bevat. De class is ingekapseld in de namespace `CSharpPFCursus`. Zo kan je in dezelfde applicatie nog een class `Program` maken (met een andere betekenis), als ze maar ingekapseld is in een namespace met een andere naam dan `CSharpPFCursus`.
Je definiert een namespace met het sleutelwoord `namespace`, gevolgd door een spatie en de naam van de namespace. Deze opdracht eindigt niet op een puntkomma. Dit is het geval als na een opdracht een open accolade volgt.
Microsoft stelt de volgende naamgeving voor bij namespaces:

- Je begint de naam van een namespace met een hoofdletter, de rest tik je in kleine letters.
- Als de naam van een namespace uit meerdere woorden bestaat, begin je ieder woord terug met een hoofdletter.

De naam van de namespace (*CSharpPFCursus*) is standaard gelijk aan de naam van het project dat je maakte.

Binnen één namespace kan je meerdere classes definiëren.

- (3) De classes die in een namespace ingekapseld zijn, moet je tussen accolades tikken. Deze accolade geeft dus het begin van de namespace aan.
- (4) Je definieert een class met het sleutelwoord **class**, gevolgd door een spatie en de naam van de class. Microsoft stelt de volgende naamgeving voor bij classes:
 - Je begint de naam van een class met een hoofdletter, de rest tik je in kleine letters.
 - Als de naam van een class uit meerdere woorden bestaat, begin je ieder woord terug met een hoofdletter.
- (5) De members van een class (methods en properties) omring je met accolades. Deze accolade geeft het begin van de class aan.

- (6) Hier definieer je een method.

Het sleutelwoord **static** geeft aan dat dit een method met class bereik is.

Dit wil zeggen dat je de method kan oproepen zonder eerst een object van de bijbehorende class te creëren. Je kan dus de method *Main()* oproepen zonder eerst een object van de class *Program* te creëren (zie verderop in deze cursus).

Bij de opstartmethod van een applicatie is het sleutelwoord **static** verplicht.

Het sleutelwoord **void** geeft aan dat de method geen resultaatwaarde teruggeeft bij oproep. Daarna volgt de naam van de method.

Microsoft stelt de volgende naamgeving voor bij methods:

- Je begint de naam van een method met een hoofdletter, de rest tik je in kleine letters.
- Als de naam van een method uit meerdere woorden bestaat, begin je ieder woord terug met een hoofdletter.

Na de naam van de method volgen ronde haakjes. Tussen deze ronde haakjes kan je de parameters voor de method beschrijven. De opstartmethod *Main()* krijgt als parameter een array van strings binnen. 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*). De parameter *args* wordt bij het starten van de applicatie automatisch gevuld met de argumenten die de gebruiker meegeeft bij het starten van de applicatie. Deze argumenten heten *command line arguments*. Als de gebruiker de applicatie op de volgende manier start:



bevat de parameter *args* twee elementen: Asterix en Obelix.

- (7) De code van een method omring je met accolades. Deze accolade geeft het begin van de code van de method *Main()* aan.
- (8) Deze accolade sluit de method *Main()*.
- (9) Deze accolade sluit de class *Program*.
- (10) Deze accolade sluit de namespace *CSharpPFCursus*.

4.5 Tekst op het scherm afbeelden

Je voegt aan de method *Main()* een opdracht toe, waarmee je de tekst *Hallo* op het scherm afbeeldt:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hallo");
        }
    }
}
```

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 (zie verderop in deze cursus).

Bij deze method vermeld je tussen ronde haakjes wat je wil afbeelden.

Vaste tekst vermeld je in C# tussen dubbele aanhalingstekens.

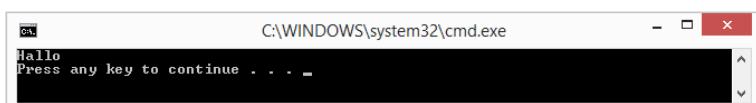
4.6 Het programma compileren en uitvoeren

Nadat de code van het programma ingetikt is, moet VS.NET deze code controleren en vertalen naar CIL-code. Daarna kan het programma uitgevoerd worden.

Je compileert en start het programma met het menu **DEBUG – Start Debugging**. Je kan ook de knop  in de werkbalk gebruiken of de functietoets **F5** drukken.

Het programma start, toont even *Hallo* op het scherm en stopt terug.

Om de uitvoer van het programma – *Hallo* – langer te kunnen bekijken, start je het programma met het menu **DEBUG – Start Without Debugging**, of je drukt *Ctrl+F5*. Nadat het programma uitgevoerd is, zie je de tekst *Press any key to continue....*



Door een willekeurige toets in te drukken, sluit je dit venster.

4.7 Methods van de class Console

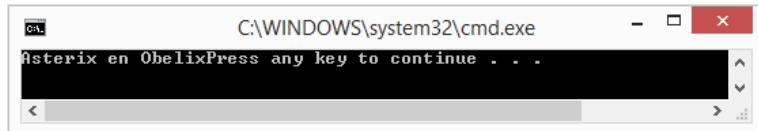
De class *Console* bevat twee methods om tekst op het scherm af te beelden:

Write()	Je toont de tekst op het scherm en plaatst de cursor onmiddellijk na deze tekst. Als je daarna terug tekst afbeeldt, komt deze tekst <i>op dezelfde regel als de vorige tekst</i> .
WriteLine()	Je toont de tekst op het scherm en plaatst de cursor op een volgende regel. Als je daarna terug tekst afbeeldt, komt deze tekst op een regel <i>onder de vorige tekst</i> .

Tekst tonen met de *Write()* method:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Asterix");
            Console.Write(" en Obelix");
        }
    }
}
```

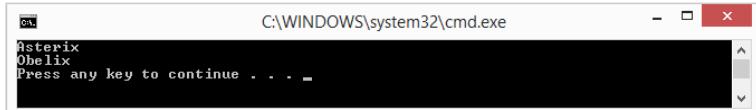
Je ziet Asterix en Obelix op het scherm:



Tekst tonen met de *WriteLine()* method:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Asterix");
            Console.WriteLine("Obelix");
        }
    }
}
```

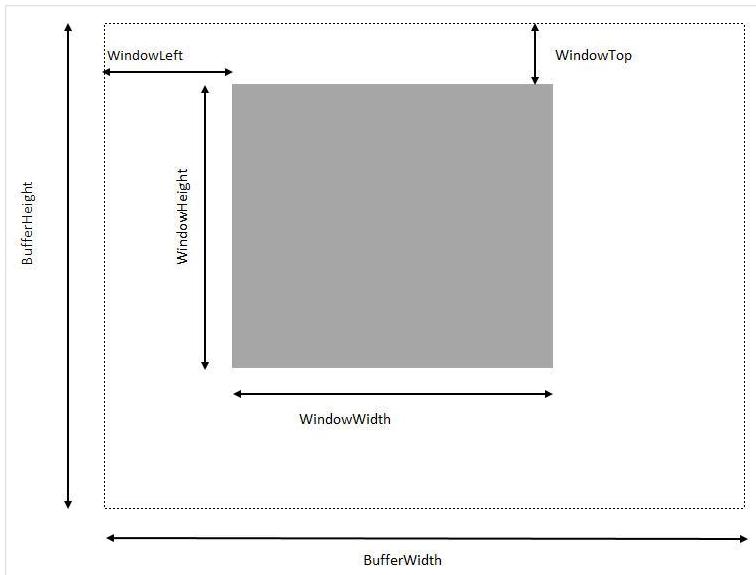
Je ziet Asterix en op een volgende regel Obelix op het scherm:



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().

Schematisch:

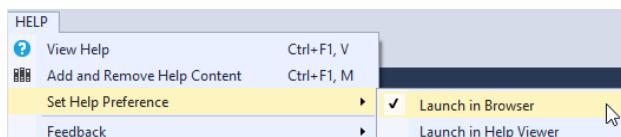


Clear()	Met deze method maak je de buffer helemaal leeg en dus ook de Console.
Title	Met deze property bepaal je welke tekst er in de titelbalk van het consolevenster verschijnt.
Read() en ReadLine()	Met deze methods lees je tekst in vanaf de console. Meer hierover in het hoofdstuk INVOER VANAF HET TOETSENBORD.
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.
CursorPosition, CursorTop en SetCursorPosition()	Via de properties CursorPosition 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.

4.8 Gebruik van de Help-functie

Zoals reeds eerder vermeld beschikt VS.NET over een uitgebreide documentatie, de MSDN. Je kan ervoor kiezen om de MSDN lokaal op je computer installeren, of om dit niet te doen en gebruik te maken van de online MSDN. Het voordeel van de online help is dat deze regelmatig bijgewerkt wordt. Het nadeel is dat je steeds een internetverbinding nodig hebt om de documentatie te raadplegen.

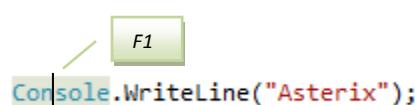
Via het menu *HELP - Set Help Preference* kan je switchen tussen beide mogelijkheden:



- Kies je voor *Launch in Browser*, dan wordt de online documentatie in een webbrowser getoond.
- Kies je voor *Launch in Help Viewer*, dan wordt de documentatie in een lokaal venster – de Help Viewer – getoond, uiteraard op voorwaarde dat de MSDN lokaal geïnstalleerd is. Zoniet, dan biedt deze optie je de mogelijkheid om dit alsnog te doen.

Als voorbeeld bekijk je de documentatie van de class *Console*.

Plaats hiervoor de cursor in de programmacode in het woord *Console* en druk dan op de toets *F1*.



Een venster met de documentatie wordt geopend, in dit voorbeeld in de lokale Help Viewer:

The screenshot shows the Microsoft Help Viewer window for the **Console Class**. At the top, there are tabs for "Console Class (System)" and "Manage Content". Below the title, there's a link to "Send Feedback on this topic to Microsoft". A description states: "Represents the standard input, output, and error streams for console applications. This class cannot be inherited." It also mentions: "To browse the .NET Framework source code for this type, see the [Reference Source](#)".

Inheritance Hierarchy

System.Object
System.Console

Namespace: System
Assembly: mscorelib (in mscorelib.dll)

Syntax

JavaScript C# C++ F# JScript VB

```
public static class Console
```

The **Console** type exposes the following members.

Properties

Show: Inherited Protected

	Name	Description
	BackgroundColor	Gets or sets the background color of the console.
	BufferHeight	Gets or sets the height of the buffer area.
	BufferWidth	Gets or sets the width of the buffer area.

- Je krijgt alle informatie over de *Console Class (namespace System)*. Dit is inderdaad de class waarover we meer informatie willen weten. Je moet er steeds voor opletten dat de juiste class van de juiste namespace of het juiste sleutelwoord geselecteerd is. Je kan eventueel via de zoekbox rechts bovenaan verder zoeken.
 - Je vindt hier een uitgebreide beschrijving van de class *Console*, de namespace waartoe de class behoort, of de class static is, eventueel codevoorbeelden ... en dit in verschillende programmeertalen.
- Via het juiste tabje kies je de gewenste taal, hier C#:

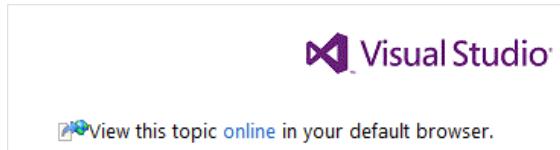
Namespace: System
Assembly: mscorelib (in mscorelib.dll)

Syntax

C# C++ F# VB

```
public static class Console
```

- Je ziet een beschrijving van alle members – properties, methods en events – van de class *Console*.
- Vanuit de **lokale MSDN** kan je steeds switchen naar de **online versie** via een link rechtsboven in de Help Viewer:



- Je krijgt dezelfde documentatie in een browservenster aangeboden, waarbij je in het linkerdeel meer specifiek kan selecteren. Je kan bvb. klikken op *Console Methods* om enkel de documentatie van de methods van de class *Console* te zien.



Het is op dit ogenblik niet de bedoeling om deze documentatie volledig te lezen.

4.9 Commentaar

Je kan een programma documenteren (voor jezelf en je collega programmeurs) door in je code commentaar te schrijven.

Je kan op twee manieren standaard commentaar voorzien in je source:

- Je begint commentaar met `//`.
Deze commentaar loopt tot het einde van de regel.
- Je begint commentaar met `/*`. Deze commentaar loopt tot je `*/` tikt.
Zo kan je commentaar schrijven over meerdere regels.

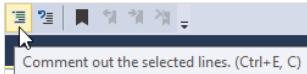
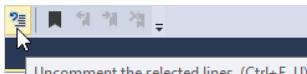
Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            //een voorbeeld
            Console.WriteLine("Hallo"); // ik zei hallo
            /*
            ik heb
            hallo gezegd
            */
        }
    }
}
```

}



Naast deze twee vormen van commentaar bestaat ook nog xml commentaar. Deze begint met `///` en eindigt bij het einde van de regel of begint met `/**` en eindigt met `*/`. Xml commentaar valt buiten het bereik van deze basiscursus.

	<p>Je kan op een eenvoudige manier meerdere regels code tegelijkertijd in commentaar plaatsen:</p> <ul style="list-style-type: none"> • Selecteer de code die je in commentaar wil plaatsen. • Klik op de knop  in de werkbalk <i>Text Editor</i>. <p></p> <p>Om meerdere lijnen code tegelijkertijd uit commentaar te halen ga je als volgt tewerk:</p> <ul style="list-style-type: none"> • Selecteer de betreffende code waarvan je de commentaar wil verwijderen. • Klik op de knop  in de werkbalk <i>Text Editor</i>. <p></p>
--	---

4.10 Een opdracht over meerdere regels uitsplitsen

Je kan een te lange opdracht over meerdere regels uitsplitsen. Dit bevordert de leesbaarheid van je code.

Daar je een opdracht in C# afsluit met een puntkomma, mag je de opdracht splitsen waar je wil, zolang je dit maar niet doet binnen een vaste tekst (een tekst tussen dubbele aanhalingstekens), of midden in een getal.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(
                "Hallo");
        }
    }
}
```



Een vast stuk tekst in je source mag je niet uitsplitsen over meerdere regels. Het volgende mag dus **niet**:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hal
                lo");
        }
    }
}
```

Als je dit voorbeeld intikt en enkele seconden wacht, zie je de fout met een rode kleur gegolfed onderlijnd. VS meldt je bepaalde codefouten reeds vóór je compileert.

Als je de muisaanwijzer op de gegolfde onderlijning laat rusten, zie je een korte omschrijving van de fout in een popup-venstertje:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hal
                lo");
        }
    }
}
```

4.11 De VS IDE

De Visual Studio.NET IDE helpt de ontwikkelaar bij het schrijven van de programmacode.

4.11.1 De Solution Explorer

De Solution Explorer geeft een overzicht van de verschillende onderdelen van een project en/of van een solution. Via de Solution Explorer kan de programmeur eenvoudig switchen tussen deze verschillende onderdelen.

Door bvb. dubbel te klikken op de klasse *Program.cs* zie je in het codevenster links de code van deze klasse. Als je dubbelklikt op de file *AssemblyInfo.cs*, zie je links de inhoud van dit bestand.

4.11.2 Tabs

Je kan ook via de tabs bovenaan het codevenster switchen tussen de verschillende geopende codevensters. De IDE plaatst een sterretje (*) bij de bestandsnaam indien de code nog niet bewaard werd (zie tab *Program.cs**).

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hallo");
            Console.WriteLine(" iedereen");
            Console.WriteLine("Einde programma");
        }
    }
}

```

Gele en groene balkjes in de marge van het codevenster geven aan dat de code op die plaats gewijzigd werd sinds dit codevenster geopend werd: groen voor opgeslagen wijzigingen, geel voor nog niet bewaarde wijzigingen.

4.11.3 Hulp bij het schrijven van code

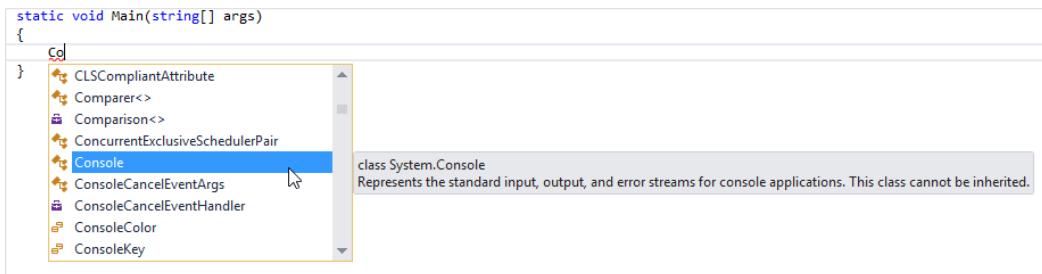
VS geeft je helpinformatie tijdens het intypen van programmacode d.m.v. popup-vensters. Deze feature heet *IntelliSense* en is zeer nuttig bij het schrijven van code.

Als je de eerste letter(s) van een gereserveerd woord van C# intypt, toont VS een popup-venster met een lijst van items die beginnen met de ingetikte letter(s). Hieruit kan je een keuze maken door:

- dubbel te klikken op het gewenste item
of
- met de ↑ en ↓ - pijltjes het gewenste item te kiezen en met de *Enter*-toets of *Tab*-toets de keuze te bevestigen

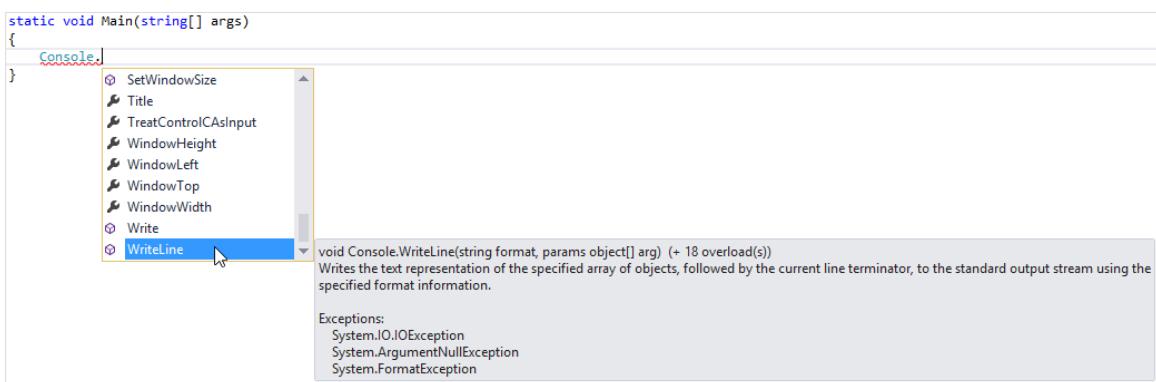
	Je kan ook enkele letters intikken en vervolgens de toetsencombinatie <i>Ctrl spatiebalk</i> ingeven om het popupvenster IntelliSense te openen.
--	--

Voorbeeld: typ de letters *Co* en kies dan het woord *Console* uit de voorgestelde items.



Merk op dat er telkens bij elk item een korte uitleg verschijnt, in de vorm van een tooltip.

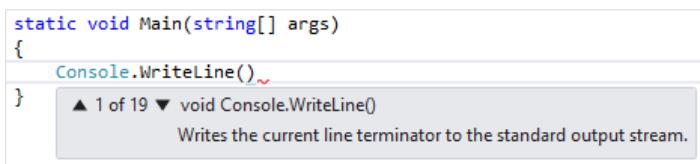
Typ nu een *. (punt)* en merk op dat er weer een popup-venster verschijnt met een lijst van properties en methods van de class *Console*:



Kies hier de method *WriteLine*.

Nu zie je het volgende in de code-editor: *Console.WriteLine*.

Typ nu een *((haakje)*. Weer verschijnt er een popup-venster:



VS stelt een aantal (19) verschillende schrijfwijzen voor voor de method *WriteLine()*. Door op de pijltjes te klikken kan je bladeren tussen de verschillende mogelijkheden.

Vervolledig de code als volgt: *Console.WriteLine("Hallo")*.

4.11.4 Foutmeldingen

Veel fouten in de code worden onmiddellijk gesigneerd d.m.v. een rode gegolfde lijn in de code en in het *Error List* venster onderaan het scherm. Via het menu *VIEW – Error List* kan je dit *Error List* venster eventueel zichtbaar maken.

Andere fouten worden pas gesigneerd bij het compileren van de programmacode.

Wanneer je een programma uitvoert, wordt de code eerst gecompileerd. Indien tijdens deze compilatie fouten ontdekt worden, wordt het programma niet uitgevoerd. Door dubbel te klikken op het rode symbooltje  naast de foutmelding, springt de cursor naar de plaats van de fout in de code. Deze fouten moeten eerst verbeterd worden vooraleer de code kan uitgevoerd worden.

	<p>Via het menu <i>BUILD – Build Solution</i> of <i>BUILD – Build CSharpPFCursus</i> kan je respectievelijk de solution of het project compileren.</p> <p>Je kan ook in de <i>Solution Explorer</i> met de rechtermuisknop klikken op de naam van de solution of van het project en dan kiezen voor <i>Build Solution/Rebuild Solution</i> of <i>Build/Rebuild</i> voor het project.</p>
---	--

4.11.5 De lay-out van de programmacode herschikken

Indien nodig kan je op een snelle manier de code netjes herschikken door de cursor ergens in de code te plaatsen en te kiezen voor het menu *EDIT – Advanced – Format Document*. Je kan ook een deel van de code herschikken door deze code te selecteren en te kiezen voor het menu *EDIT – Advanced – Format Selection*.

Hierdoor wordt de code beter leesbaar.

<pre>using System; namespace CSharpPFCursus { class Program { static void Main(string[] args) { Console.WriteLine("Hallo"); } } }</pre>		<pre>using System; namespace CSharpPFCursus { class Program { static void Main(string[] args) { Console.WriteLine("Hallo"); } } }</pre>
---	---	---

Indien het herschikken niet lukt, zit er waarschijnlijk nog ergens een fout in de code.

	<p>Code herschikken kan je ook via de toetsencombinatie Ctrl E D.</p>
---	--

4.11.6 De lay-out van VS.NET

De ontwikkelomgeving VS.NET bevat verschillende vensters:

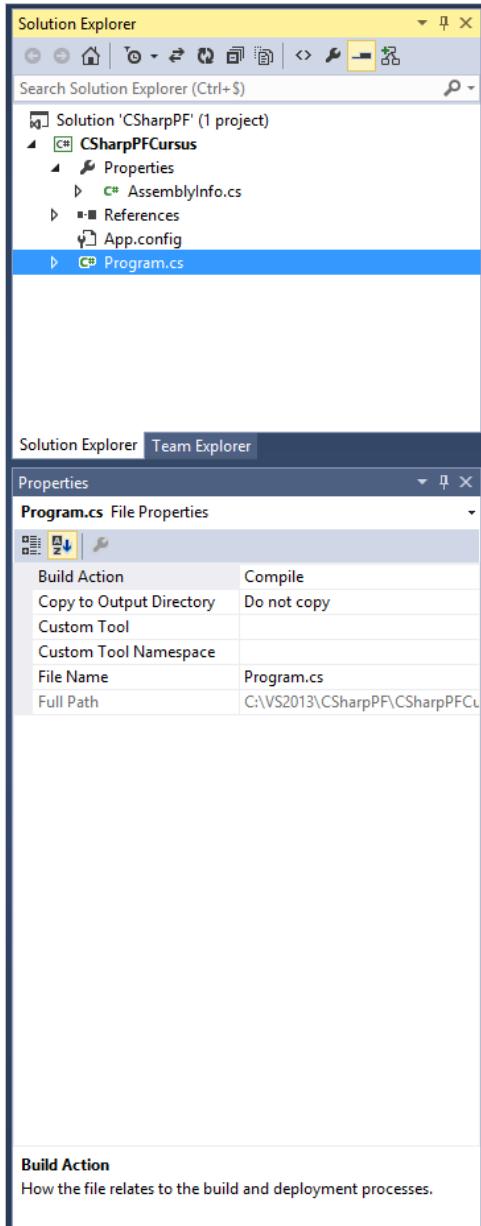
- het *Solution Explorer* venster
- het *Properties* venster

Dit kan je zichtbaar maken via het menu *VIEW – Properties Window* of met de knop  in de werkbalk van de *Solution Explorer* of met de functietoets **F4**.

Het *Properties* venster verschijnt standaard onder de *Solution Explorer* en toont de properties

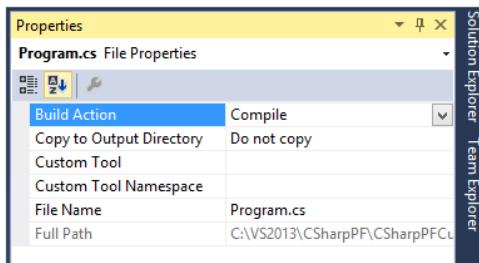
van een geselecteerd onderdeel in de ontwikkelomgeving. Je kan hier indien nodig ook properties wijzigen.

Selecteer je bvb. in de Solution Explorer *Program.cs*, dan zie je het volgende:

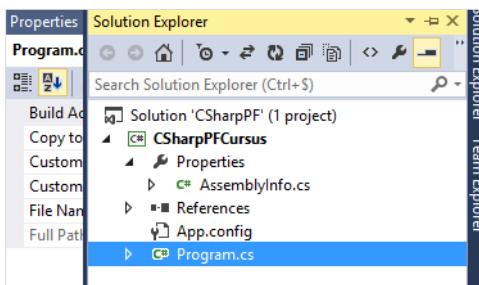


Elk venster bevat bovenaan een *Auto Hide* (/) knop. Klik je op de knop van de *Solution Explorer*, dan verdwijnt het venster en komt er een *tab* in de plaats.

Het *Properties* venster is nu blijvend zichtbaar, de *Solution Explorer* vind je terug als een tab.



Klik je met de muis op deze *tab*, dan komt de *Solution Explorer* terug tevoorschijn:

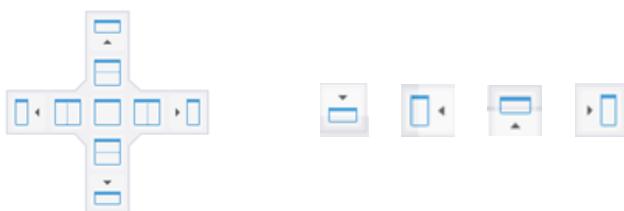


Met de knop kan je het *Solution Explorer* venster terug blijvend zichtbaar maken.

De plaats van de verschillende vensters ligt niet definitief vast. Je kan een venster naar eigen keuze van plaats verwisselen door het venster naar de gewenste positie te slepen.

Tijdens het verslepen verschijnen er verschillende symbolen op het scherm:

in het midden en respectievelijk links, bovenaan, rechts, onderaan op het scherm:



Wanneer je het venster tijdens het slepen op één van deze symbolen plaatst, verschijnt er een blauwe zone die aangeeft waar het venster bij het loslaten van de muis terechtkomt. Je kan hiermee experimenteren.



Via het menu *WINDOW – Reset Window Layout* kan je op een snelle manier de lay-out van de ontwikkelomgeving terug naar de oorspronkelijke lay-out aanpassen.

5 Lokale variabelen

5.1 Algemeen

Je gebruikt lokale variabelen om tijdelijke waarden (getallen, teksten, ...) te onthouden terwijl je bewerkingen (bvb. berekeningen) uitvoert in een *method*.

C# onthoudt variabelen in het intern geheugen van de PC.

Elke variabele heeft een:

Naam	Waarmee je naar die variabele verwijst vanuit je code.
Type	Welk soort informatie kan de variabele bevatten (bvb. enkel gehele getallen)?
Waarde	De inhoud van de variabele.

5.2 Naamgeving

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 tikken in kleine letters (bvb. leeftijd). Als een variabelenaam uit meerdere woorden bestaat, begin je de eerste letter van ieder woord na het eerste woord met een hoofdletter (bvb. aantalKinderen). Dit verhoogt de leesbaarheid.

5.3 Types variabelen

C# heeft twee fundamentele categorieën van types variabelen:

Value types	De variabele bevat zelf zijn waarde in zich.
Reference types	De variabele bevat een verwijzing (reference) naar een object. Het object bevat zelf één of meerdere waardes en/of verwijzingen naar andere objecten.

Reference types komen later aan bod. Eerst bespreken we de Value types.

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 $1.5 \cdot 10^{-45}$ tot $3.4 \cdot 10^{38}$ (precisie ± 7 decimalen)	4
double	Double	Decimaal getal tussen $5.0 \cdot 10^{-324}$ tot $1.7 \cdot 10^{308}$ (precisie ± 15 decimalen)	8
decimal	Decimal	Decimaal getal tussen $1.0 \cdot 10^{-28}$ tot $7.9 \cdot 10^{28}$ (precisie ± 28- decimalen)	16
char	Char	Eén teken	2
bool	Boolean	true of false	1

- De typenaam in C# kan afwijken van de typenaam in de CLR (Common Language Runtime). Dit is niet erg: de C# compiler vertaalt de typenamen van C# naar typenamen van de CLR.
- Een variabele van het type `char` bevat zijn teken volgens de Unicode standaard (UTF-16). Unicode is de opvolger van ASCII. Unicode gebruikt 2 bytes om een teken te coderen, waar ASCII 1 byte gebruikt. Op die manier kan Unicode meer tekens uit meer talen een unieke code geven.
- Voor gehele getallen is het type `int` het snelst. Daarom gebruik je dit type soms waar je ook kleinere types (`byte`, `short`) zou kunnen gebruiken.
- Berekeningen op het type `float` of `double` kunnen soms leiden tot kleine afrondingsfouten. Berekeningen op het type `decimal` leiden nooit tot afrondingsfouten. Het type `decimal` is wel trager dan de types `float` en `double` en neemt meer geheugen in beslag. Het type `decimal` wordt gebruikt voor financiële bedragen, financiële en wetenschappelijke berekeningen.
- Met uitzondering van het data type `bool` hebben alle bovenstaande data types een property *MinValue* en *MaxValue*. De volgende code beeldt de maximumwaarde van een uint af:
`Console.WriteLine("uint max value: " + uint.MaxValue);`

5.4 Declaratie

Je moet een variabele declareren vóór je ze gebruikt.

Je declareert een variabele door eerst het type van de variabele te tikken, gevolgd door een spatie, gevolgd door de naam van de variabele.

Je sluit de declaratieopdracht af met een puntkomma:

`variabeleType naamVariabele;`

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            byte statenUSA;
            ushort belgischeBieren;
            uint afstandNaarMaan;
            ulong aantalMensen;
            float alcoholDuvel;
            char geslacht;
            bool gehuwd;
```

```

        }
    }
}
}
```

Je kan meerdere variabelen van hetzelfde type in één opdracht declareren:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            float alcoholDuvel, alcoholPalm;
            byte statenUSA, aantalKinderen;
        }
    }
}
```

5.5 Variabelen invullen

Bij de declaratie van een lokale variabele krijgt deze variabele in C# geen beginwaarde. Je vult een variabele met een waarde d.m.v. de toekenningsopdracht.

Hierbij tik je de naam van de variabele, het teken = en daarna de waarde die je in de variabele wil invullen. Je sluit de opdracht af met een puntkomma:

```
naamVariabele=waarde;
```

Een letterlijke waarde die je in de code in een variabele invult, heet een *literal*. Ook een literal heeft een type. Hierbij gelden de volgende regels:

- Een **bool** variabele kan enkel de waarde **true** of **false** bevatten.
- Een **char** variabele kan één teken bevatten. Je omsluit dit teken met enkele aanhalingstekens, bvb. **char letter='a';**.
- Als je een literal gebruikt die een geheel getal voorstelt en die geen suffix heeft (zie verder) dan is het type van deze literal het eerste van de volgende types waarin de waarde van het getal kan opgeslagen worden: **int**, **uint**, **long** en **ulong**. VS kiest dus zo weinig mogelijk bytes om het getal op te slaan. Is de literal groter dan de maximum waarde van het type **ulong** dan krijg je een '*integral constant is too large*' compileerfout.

Als je een literal, die een geheel getal voorstelt, toekent aan een variabele van het type **byte**, **sbyte**, **short**, of **ushort** en de waarde valt buiten de toegelaten onder- en bovengrenzen van deze types, dan converteert C# dit getal naar **byte**, **sbyte**, **short** of **ushort**.

Om conversies te vermijden en aldus aan performantie te winnen kan je gebruik maken van suffixen. Een geheel getal literal met bijvoorbeeld een suffix **u** of **U** wordt door C# beschouwd als een **uint** of **ulong**, meer bepaald het eerste type waarin de waarde kan opgeslagen

worden. Een overzicht:

Een getal met de volgende suffix...	... is in C# ...
u of U	uint of ulong
l of L	long of ulong
UL , Ul , uL , ul , LU , Lu , lU of lu	ulong

- Als je een getal met decimalen tikt, gebruik je het punt als scheidingsteken tussen eenheden en decimalen. C# beschouwt zo'n getal standaard als **double**.

Wens je een decimale literal toe te kennen aan een **float** dan moet je een suffix **f** of **F** gebruiken. Deze suffix maakt van de literal een **float** i.p.v. een **double**. Uiteraard moet het getal wel tussen de onder- en bovengrenzen van een **float** liggen.

Wens je een decimale literal toe te kennen aan een **decimal** dan moet je een suffix **m** of **M** gebruiken. Deze suffix maakt van de literal een **decimal** i.p.v. een **double**. Uiteraard moet het getal wel tussen de onder- en bovengrenzen van een **decimal** liggen.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            byte statenUSA;
            ushort belgischeBieren;
            uint afstandNaarMaan;
            ulong aantalMensen;
            float alcoholDuvel;
            char geslacht;
            bool gehuwd;
            statenUSA = 50;
            belgischeBieren = 1400;
            afstandNaarMaan = 382170;
            aantalMensen = 6122567014ul;
            alcoholDuvel = 8.5f;
            geslacht = 'M';
            gehuwd = true;
            Console.WriteLine(statenUSA);
            Console.WriteLine(belgischeBieren);
            Console.WriteLine(afstandNaarMaan);
            Console.WriteLine(aantalMensen);
            Console.WriteLine(alcoholDuvel);
```

```
        Console.WriteLine(geslacht);
        Console.WriteLine(gehuwd);
    }
}
```

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

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            float alcoholDuvel, alcoholHapkin;
            alcoholDuvel = alcoholHapkin = 8.5f;
            Console.WriteLine(alcoholDuvel);
            Console.WriteLine(alcoholHapkin);
        }
    }
}
```

5.6 Variabelen tegelijkertijd declareren en invullen

Tot nu declareerde je eerst een variabele en vulde je de variabele daarna in met een waarde. Je kan ook tegelijkertijd een variabele declareren én er een beginwaarde aan geven:

```
variabeleType variabeleNaam=beginWaarde;
```

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            byte statenUSA = 50;
            ushort belgischeBieren = 1400;
            uint afstandNaarMaan = 382170;
            ulong aantalMensen = 6122567014ul;
            float alcoholDuvel = 8.5f;
            char geslacht = 'M';
            bool gehuwd = true;
            Console.WriteLine(statenUSA);
            Console.WriteLine(belgischeBieren);
            Console.WriteLine(afstandNaarMaan);
            Console.WriteLine(aantalMensen);
            Console.WriteLine(alcoholDuvel);
        }
    }
}
```

```
        Console.WriteLine(geslacht);
        Console.WriteLine(gehuwd);
    }
}
}
```

De beginwaarde kan ook een berekening zijn. In die berekening mag je getallen, maar ook variabelen gebruiken die al gedeclareerd zijn.

Je kan een variabele *weddeInEuro* dus initialiseren als een berekening van de wedde in BEF gedeeld door de koers van de €:

```
decimal weddeInEuro=5000m/40.3399m;
```

5.7 Typeconversies van Value types

Als je een variabele toekent aan een andere variabele kan het gebeuren dat de variabele links van = niet hetzelfde type heeft als de variabele rechts van =.

Als de variabele links van = een type heeft met een groter bereik dan de variabele rechts van =, hoeft je geen speciale maatregelen te treffen.

Met bereik bedoelen we de kleinste en grootste waarde die het type ondersteunt.

Een toekenning van een *int* variabele aan een *double* variabele:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int eenWedde = 1500;
            double eenTweedeWedde = eenWedde;
            Console.WriteLine(eenTweedeWedde);
        }
    }
}
```

Als de variabele links van = een type heeft met een kleiner bereik dan de variabele rechts van =, moet je een typeconversie gebruiken:

```
variabeleMetKleinerType=(kleinerType)variabeleMetGroterType;
```

Je vermeldt dus na = tussen ronde haakjes het type van de variabele links van =.

Als je bijvoorbeeld een variabele van het type *double* toekent aan een variabele van het type *int*, schrijf je dit op deze manier:

```
using System;
namespace CSharpPFCursus
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            double eenWedde = 1500.78;
            int eenTweedeWedde = (int)eenWedde;
            Console.WriteLine(eenWedde);
            Console.WriteLine(eenTweedeWedde);
        }
    }
}
```

Bij de typeconversie gaan cijfers na de komma verloren als de variabele links van = een geheel getal type is en de variabele rechts van = een type met decimalen is.

Let er ook voor op dat de waarde van de variabele rechts van = past binnen het bereik van de variabele links van de =, anders krijg je onverwachte resultaten.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int getal1 = 1500;
            byte getal2 = (byte)getal1;
            Console.WriteLine("getal1: " + getal1);
            Console.WriteLine("getal2: " + getal2);
        }
    }
}
```

Het resultaat van deze code is:



M.a.w. de waarde van de tweede variabele is totaal verschillend van deze van de eerste variabele.

Je kan de typeconversies ook gebruiken los van de toekenningsoptdracht:

```
using System;
namespace CSharpPFCursus
{
```

```
class Program
{
    static void Main(string[] args)
    {
        double eenWedde = 1500.78;
        Console.WriteLine((int)eenWedde);
    }
}
```

Je gebruikt de (`int`) typeconversie ook om de Unicode van een `char` te kennen:

`Console.WriteLine((int)'A');` → geeft 65.

Je gebruikt de (`char`) typeconversie ook om het teken van een Unicode-waarde te kennen:

`Console.WriteLine((char)65);` → geeft A.



Bij het programmeren maak je wel eens een fout. De ontwikkelomgeving VS.NET biedt je een aantal hulpmiddelen om fouten in de programmacode op te sporen = debugging.

In **Bijlage 1: Debugging** worden een aantal debugging tools van VS.NET beschreven.

Probeer deze nu uit!

6 Constanten

6.1 Algemeen

Je gebruikt constanten om **onveranderlijke** waarden (getallen, teksten, datums, ...) te onthouden in het intern geheugen van de PC.

Voorbeelden van onveranderlijke gegevens:

- PI: 3,1415926535897932384626433832795
- De koers van de € t.o.v. de BEF: 40,3399
- Het aantal cm in een inch: 2,54
- De gulden snede: de ideale verhouding tussen de lengte en de breedte van een voorwerp of gebouw zodat het aangenaam is voor het menselijk oog:
 $(\sqrt{5}+1)/2$

Elke constante heeft een:

Naam	Waarmee je naar die constante verwijst vanuit je code. Microsoft raadt aan de naam van een constante met een hoofdletter te beginnen. Zo onderscheid je ze gemakkelijk van variabelen. Als de naam uit meerdere woorden bestaat, begin je ieder woord terug met een hoofdletter.
Type	Bepaalt welk soort informatie de constante kan bevatten. Je beschikt over dezelfde types als bij de variabelen.
Waarde	De inhoud van de constante.

6.2 Syntax

6.2.1 Readonly

Als je een *formule* nodig hebt om de inhoud van de constante te bepalen, gebruik je het sleutelwoord **readonly**:

```
readonly constanteType ConstanteNaam =formule;
```

Voorbeeld:

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

(met de method *Sqrt* van de class *Math* bereken je een vierkantswortel)

Constanten uitgedrukt met **readonly** krijgen hun inhoud bij de uitvoering van het programma.

Als je een **readonly** waarde in een class definieert, zal deze **readonly** waarde per object van deze class in het geheugen voorkomen. Als je 10 objecten van de class hebt, zal de **readonly** waarde 10

keer in het geheugen voorkomen. In het geval van de gulden snede is dit zinloos: de gulden snede kan maar één waarde hebben en heeft dus voor elk object dezelfde waarde.

Opdat de `readonly` waarde maar één keer in het geheugen zou voorkomen, ongeacht het aantal objecten van de class waarin de `readonly` gedefinieerd is, declareer je de `readonly` met het sleutelwoord `static`:

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

Het sleutelwoord `static` komt verder in deze cursus aan bod.

6.2.2 Const

Als je geen formule nodig hebt om de inhoud van de constante te bepalen, gebruik je het sleutelwoord `const`:

```
const constanteType ConstanteNaam=inhoud;
```

Voorbeeld:

```
const decimal EuroKoers=40.3399m;
```

Constanten uitgedrukt met `const` krijgen hun inhoud bij de compilatie van het programma, wat de snelheid van uitvoering bevordert.

6.3 Voorbeeld

Een programma dat een afstand in centimeters omzet naar een afstand in inches:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const float CmInch = 2.54f;
        static readonly double GuldenSnede = (Math.Sqrt(5.0) + 1.0) / 2.0;
        static void Main(string[] args)
        {
            float cm = 100.0f;
            float inch = cm / CmInch;
            Console.WriteLine(cm);
            Console.WriteLine(inch);
            Console.WriteLine(GuldenSnede);
        }
    }
}
```

7 Bewerkingen

7.1 Bewerkingen op getallen

C# bevat de volgende arithmetic operators (rekenkundige bewerkingen):

Bewerking	Betekenis
-	Negatie (het teken van een getal omkeren)
+	Optellen
-	Aftrekken
*	Vermenigvuldigen
/	Delen Als de deler én het deeltal gehele getallen zijn, kapt C# het resultaat af tot een geheel getal. <code>Console.WriteLine(5/2);</code> geeft 2. Als de deler en/of het deeltal een getal met decimalen is, behoudt C# de decimalen in het resultaat. <code>Console.WriteLine(5/2.0);</code> geeft 2.5.
%	Restbepaling bij deling <code>Console.WriteLine(20%3);</code> geeft 2.

Voorbeeld:

Om na te gaan of een btw-nummer (notatie 000.000.000) correct is, moet je de eerste 7 cijfers van het btw-nummer delen door 97. De rest van deze deling trek je af van 97, het resultaat van dit verschil moet gelijk zijn aan de laatste 2 cijfers van het btw-nummer.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const ulong BtwNummer = 213252520UL; (1)
        static void Main(string[] args)
        {
            ulong deeltal = BtwNummer / 100UL; (2)
            int rest = (int)(deeltal % 97UL); (3)
            int controle = (int)(BtwNummer % 100); (4)
            Console.WriteLine("97 - de rest van de deling " +
                "van de eerste 7 cijfers door 97: {0}", 97 - rest); (5)
            Console.WriteLine("Twee laatste cijfers: {0}", controle); (6)
        }
    }
}
```

- (1) Vervang eventueel door een ander btw-nummer.
- (2) Om de eerste 7 cijfers te bekomen, deel je het btw-nummer door 100, met afkapping van decimalen.
- (3) Bereken de rest van de deling van de eerste 7 cijfers door 97.
- (4) Om de laatste 2 cijfers te bekomen, bereken je de rest van de deling van het btw-nummer door 100.
- (5) Je toont het verschil van 97 en de rest van de deling van de eerste 7 cijfers van het btw-nummer door 97.
- (6) Dit moet gelijk zijn aan de laatste 2 cijfers.

7.2 Verkorte bewerkingen

Als je een variabele met een waarde verhoogt, verlaagt, vermenigvuldigt of deelt en het resultaat terug in dezelfde variabele stopt, kan je dat kort schrijven:

Korte schrijfwijze	Lange schrijfwijze	Betekenis
wedde+=2	wedde=wedde+2	Verhoog de variabele <i>wedde</i> met 2.
wedde-=3	wedde=wedde-3	Verlaag de variabele <i>wedde</i> met 3.
wedde*=2	wedde=wedde*2	Vermenigvuldig de variabele <i>wedde</i> met 2.
wedde/=2	wedde=wedde/2	Deel de variabele <i>wedde</i> door 2.
getal%=5	getal=getal%5	Stop de rest van de deling van <i>getal</i> door 5 terug in <i>getal</i> .

De waarde na `+=`, `-=`, `*=`, `/=` of `%=` kan een variabele (bvb. `wedde += opslag`), een constante of een formule (bvb. `wedde += opslag * 2`) zijn:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const float CmInch = 2.54f;
        static void Main(string[] args)
        {
            float afstand = 100000.0f;                                (1)
            afstand /= CmInch;                                         (2)
            Console.WriteLine(afstand);
        }
    }
}
```

- (1) De variabele afstand invullen met een afstand in centimeters.

(2) De afstand omzetten naar inches.

7.3 Met één verhogen of verlagen

Met de bewerking `++` verhoog je een variabele met één.

- `++` vóór de variabele verhoogt de variabele bij het begin van de opdracht.
- `++` ná de variabele verhoogt de variabele bij het einde van de opdracht.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int aantalKinderen = 0;
            Console.WriteLine(++aantalKinderen);          (1)
            Console.WriteLine(aantalKinderen++);           (2)
            Console.WriteLine(aantalKinderen);             (3)
        }
    }
}
```

- (1) De `++` bewerking staat vóór de variabele `aantalKinderen`. Eerst verhoogt C# de variabele (die op nul stond) met één. Daarna toont C# de variabele.
Je ziet dus 1 op het scherm.
- (2) De `++` bewerking staat ná de variabele `aantalKinderen`. C# verhoogt de variabele pas op het einde van de opdracht met één. De opdracht zelf beeldt dus nog de waarde 1 af die de variabele heeft voor ze verhoogd wordt.
- (3) Nu zie je de inhoud van de variabele nadat ze verhoogd werd bij (2).

De `--` bewerking heeft dezelfde eigenschappen als de `++` bewerking, maar verlaagt een variabele met één.

	<p>Je kan op een <code>char</code> variabele ook de operatoren <code>++</code> en <code>--</code> toepassen. Als je de <code>++</code> operator toepast, wordt de nieuwe waarde van de <code>char</code> variabele het volgende teken in het Unicode alfabet. Voorbeeld:</p>
---	--

```
char letter = 'A';
letter++;
Console.WriteLine(letter); → geeft B.
```

Als je de `--` operator toepast, wordt de nieuwe waarde van de `char` variabele het vorige teken in het Unicode alfabet.

Om te weten hoeveel plaatsen verschil er is tussen twee tekens in het Unicode alfabet,

	kan je twee char waarden van elkaar aftrekken: <code>Console.WriteLine('C' - 'A');</code> → geeft 2.
--	--

7.4 Vergelijgingsoperatoren

C# voorziet de volgende vergelijgingsoperatoren op de Value types:

Bewerking	Betekenis
<code>==</code>	Gelijk aan
<code>!=</code>	Verschillend van
<code>></code>	Groter dan
<code>>=</code>	Groter dan of gelijk aan
<code><</code>	Kleiner dan
<code><=</code>	Kleiner dan of gelijk aan

Het resultaat van iedere vergelijkingsbewerking is **true** of **false**.

Het resultaat van `4>3` is bvb. **true** omdat 4 groter is dan 3.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(4 == 3);
            Console.WriteLine(4.2 != 4);
            Console.WriteLine('A' < 'E'); (1)
        }
    }
}
```

(1) C# vergelijkt **char** waarden op basis van hun Unicode-waarde, je ziet dus **true**.

7.5 Bewerkingen op bool waarden

C# voorziet de volgende bewerkingen op **bool** waarden:

Bewerking	Betekenis
! waarde	Negatie: true wordt false , false wordt true .
waarde1 && waarde2	And: true als beide waarden true zijn, anders false . Als de eerste waarde false is, evalueert C# de tweede waarde niet meer.
waarde1 waarde2	Or: false als beide waarden false zijn, anders true . Als de eerste waarde true is, evalueert C# de tweede waarde niet meer.
waarde1 ^ waarde2	Exclusive Or: false als beide waarden gelijk zijn, anders true .

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            bool gehuwd = true;
            Console.WriteLine(!gehuwd);
            Console.WriteLine(4 < 3 && 1 > 2); (1)
            Console.WriteLine(4 > 3 || 1 > 2); (2)
        }
    }
}
```

- (1) De waarde vóór && ($4 < 3$) is **false**. De totale && expressie is dus **false**.
C# evalueert daarom de waarde ná && niet.
- (2) De waarde vóór || ($4 > 3$) is **true**. De totale || expressie is dus **true**.
C# evalueert daarom de waarde ná || niet.

7.6 Prioriteiten van bewerkingen

Je kan in één opdracht meerdere bewerkingen combineren.

C# voert de bewerkingen dan uit met de volgende prioriteit (van hoog naar laag):

x++ x--
- (negatie) ! ++x --x (T)x
*
/ %
+
-
< > <= >=
== !=
&&
= *= /= %= += -=



In dit overzicht betekent (T)x een typeconversie
(vb. (`int`)`getal` om de `double` variabele `getal` naar `int` om te zetten).

Als in één opdracht meerdere bewerkingen voorkomen met dezelfde prioriteit, voert C# deze bewerkingen van links naar rechts uit. In de opdracht
`inhoud = lengte * breedte * hoogte;`
berekent C# eerst `lengte * breedte` en vermenigvuldigt daarna dit resultaat met `hoogte`.

Je kan uitzonderingen maken op de prioriteiten door bewerkingen die C# eerst moet uitvoeren tussen ronde haakjes te plaatsen:

```
lidgeld = (aantalJongens + aantalMeisjes) * lidgeldPerKind;
```

C# berekent eerst de som van `aantalJongens + aantalMeisjes`.

Daarna vermenigvuldigt C# deze som met `lidgeldPerKind`.

Had je echter geschreven:

```
lidgeld = aantalJongens + aantalMeisjes * lidgeldPerKind;  
dan had C# eerst aantalMeisjes * lidgeldPerKind berekend (* is prioritair t.o.v. +) en daarna het resultaat van deze vermenigvuldiging opgeteld bij aantalJongens!
```

7.7 Wiskundige constanten en methods

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 dus deze methods via de class *Math* zelf oproepen.

Enkele constanten en methods uit de class *Math*:

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. Let op: <code>Console.WriteLine(Math.Round(1.5));</code> → 2 <code>Console.WriteLine(Math.Round(2.5));</code> → ook 2 Er wordt afgerond naar het dichtsbijzijnde even getal. Met een extra parameter kan je de manier van afronden beïnvloeden: <code>Console.WriteLine(Math.Round(1.5, MidpointRounding.AwayFromZero));</code> → 2 <code>Console.WriteLine(Math.Round(2.5, MidpointRounding.AwayFromZero));</code> → 3 Een uitgebreide documentatie vind je in de MSDN.
Round(getal, decim)	Geeft de afgeronde waarde van getal op <i>decim</i> decimalen.
Sin(getal)	Geeft de sinus van getal.
Sqrt(getal)	Geeft de vierkantswortel van getal.

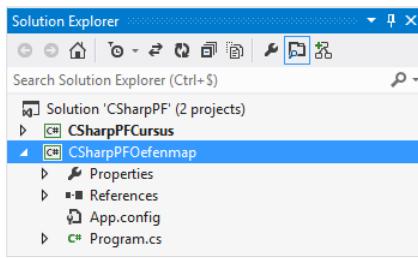
Tan(getal)	Geeft de tangens van getal.
------------	-----------------------------

Voorbeeld (oppervlakte van een cirkel berekenen):

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            double straal = 10.5;
            Console.WriteLine(Math.Pow(straal, 2) * Math.PI);
        }
    }
}
```



oefeningen: Variabelen, constanten en bewerkingen

	<p>Je kan voor de oefeningen eventueel een nieuw project <i>CSharpPFOefenmap</i> toevoegen aan de solution <i>CSharpPF</i>. Klik met de rechtermuisknop op de naam van de solution en kies <i>Add – New Project....</i>. Kies hier voor <i>Console Application</i> en vul als <i>Name:</i> <i>CSharpPFOefenmap</i> in. In de Solution Explorer zie je nu de twee projecten:</p>  <p>Om de oefeningen te kunnen uitvoeren moet je het project <i>CSharpPFOefenmap</i> als opstartproject instellen. Klik met de rechtermuisknop op het project <i>CSharpPFOefenmap</i> en kies <i>Set as StartUp Project</i>. De naam van het opstartproject wordt in het vet weergegeven.</p>
---	--

8 Het type DateTime

8.1 Algemeen

Met een variabele van het type *DateTime* stel je een datum of een combinatie van een datum én een tijd voor.

Een variabele van het type *DateTime* valt onder de categorie Value types, zoals `int`, `double`, ...: een variabele van het type *DateTime* bevat zelf de waarde van de datum in zich en is geen verwijzing naar een datum die zich in een andere plaats van het intern geheugen bevindt.

Je geeft een *DateTime* variabele een beginwaarde met een uitgebreide versie van de toekenningsopdracht:

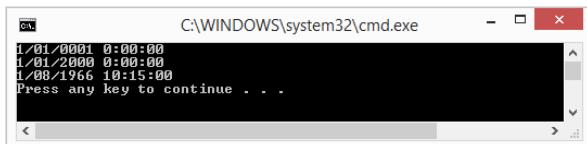
```
DateTime variabeleNaam=new DateTime(...);
```

Tussen de ronde haakjes kan je op verschillende manieren aangeven welke datum of datum-tijd combinatie je in de variabele wil invullen. In het volgende voorbeeld zie je drie manieren:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime beginTijdRekening = new DateTime();                                (1)
            DateTime beginEuro = new DateTime(2000, 1, 1);                            (2)
            DateTime mijnGeboorte = new DateTime(1966, 8, 1, 10, 15, 0);           (3)
            Console.WriteLine(beginTijdRekening);
            Console.WriteLine(beginEuro);
            Console.WriteLine(mijnGeboorte);
        }
    }
}
```

- (1) Als je geen waarde meegeeft tussen de ronde haakjes, krijg je 1 januari van het jaar 1, middernacht.
- (2) Bij deze manier geef je eerst het jaartal, dan de maand en als laatste de dag mee van de datum die je wil invullen. De datum krijgt als tijd middernacht.
- (3) Bij deze manier geef je eerst het jaartal, dan de maand, dan de dag, dan het uur, dan de minuten en als laatste de seconden mee van de datum-tijd combinatie die je wil invullen.

Bij het afbeelden van de datums houdt .NET rekening met de landinstellingen van het configuratiescherm van de computer waarop het programma draait. Bij een Belgische landinstelling zie je de variabelen als dag/maand/jaar uur:minuten:seconden.



```
C:\WINDOWS\system32\cmd.exe
1/01/0001 0:00:00
1/01/2000 0:00:00
1/08/1966 10:15:00
Press any key to continue . . .
```

	<p>Je kan ook een voorgedefinieerd datumformaat kiezen of zelf een datumformaat samenstellen.</p> <pre>DateTime eenDatum = new DateTime(2014, 9, 1); Console.WriteLine(eenDatum.ToShortDateString()); → 1/09/2014 (afhankelijk van de landinstellingen) Console.WriteLine(eenDatum.ToString("yyyy-MM-dd")); → 2014-09-01</pre>
---	---

8.2 Datums vergelijken

Om datums te vergelijken gebruik je de operatoren ==, !=, >, <, >= en <=.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime datum1 = new DateTime(2014, 1, 1);
            DateTime datum2 = new DateTime(2014, 1, 1);
            DateTime datum3 = new DateTime(2014, 1, 2);
            Console.WriteLine(datum1 == datum2); //true
            Console.WriteLine(datum3 > datum2); //true
        }
    }
}
```

8.3 Methods en properties

Een DateTime variabele heeft methods en properties.

- Een method is een handeling die een variabele voor je kan uitvoeren.
Je bereikt een method van een variabele door de naam van de variabele te tikken, gevolgd door een punt, gevolgd door de naam van de method, gevolgd door ronde haakjes.
Tussen de ronde haakjes kan je bij sommige methods parameterwaarden meegeven, waarmee je de method bijstuurt.
Soms geeft een method een resultaatwaarde van de handeling terug.

Met de *AddDays(n)* method van een *DateTime* variabele vraag je *n* aantal dagen bij de waarde van de *DateTime* variabele op te tellen:

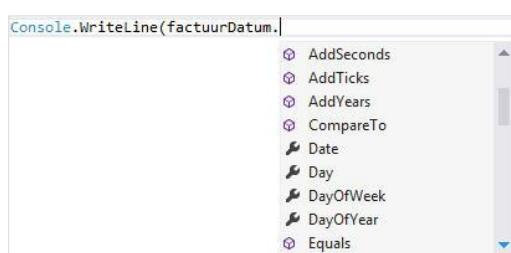
```
DateTime factuurDatum=new DateTime(2000,1,1);
Console.WriteLine(factuurDatum.AddDays(40));
```

toont 10/02/2000 0:00:00 op het scherm. De waarde van *factuurDatum* zelf in dit voorbeeld blijft ongewijzigd.

- Een property is een eigenschap van een variabele.
Je bereikt een property van een variabele door de naam van de variabele te tikken, gevolgd door een punt, gevolgd door de naam van de property.
De *DayOfYear* property van een *DateTime* variabele geeft je het volgnummer van de dag in het jaar:

```
DateTime factuurDatum=new DateTime(2000,2,15);
Console.WriteLine(factuurDatum.DayOfYear);
⇒ toont 46 op het scherm.
```

Zodra je na een variabele een punt tikt, toont VS de methods en properties van deze variabele in een popup-venster:



- Methods hebben het pictogram
- Properties hebben het pictogram

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.

De property *Today* van het type *DateTime* geeft je de systeemdatum. Dit is een **static** property. Je kan deze property niet opvragen via een *DateTime* variabele (*factuurdatum.Today*), maar wel via *DateTime* zelf (*DateTime.Today*).

8.4 Veelgebruikte methods en properties bij DateTime variabelen

We gebruiken een variabele *geboorte* 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 <i>dagen</i> verwijderd is van de datum. <i>geboorte.AddDays(3)</i> geeft 4/8/1966 4:30:15.
AddYears(jaren)	Geeft een nieuwe datum die <i>jaren</i> verwijderd is van de datum. <i>geboorte.AddYears(-5)</i> geeft 1/8/1961 4:30:15. Soortgelijke methods: <i>AddMonths(maanden), AddHours(uren), AddMinutes(minuten), AddSeconds(seconden), AddMilliseconds(milliseconden)</i> .
Year	Het jaartal van de datum. <i>geboorte.Year</i> 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. <i>geboorte.DayOfWeek</i> geeft Monday. (int) <i>geboorte.DayOfWeek</i> geeft 1.
DayOfYear	De hoeveelste dag in het jaar de datum is. <i>geboorte.DayOfYear</i> geeft 213.
DaysInMonth(jaar, maand)	Het aantal dagen in de maand van jaar (static property). <i>DateTime.DaysInMonth(2000, 2)</i> geeft 29.
Today	De systeemdatum zonder tijd (static property). <i>DateTime.Today</i> geeft de systeemdatum zonder tijd.
Now	De systeemdatum met tijd (static property). <i>DateTime.Now</i> geeft de systeemdatum met tijd.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime vandaag = DateTime.Today; (1)
            Console.WriteLine(vandaag.DayOfWeek);
            Console.WriteLine(vandaag.DayOfYear);
            Console.WriteLine(vandaag.AddDays(1)); (2)
        }
    }
}
```

```
    }  
}  
}
```

- (1) Een variabele opvullen met de huidige datum.
- (2) De datum van morgen afbeelden.

9 Het type string

9.1 Algemeen

- Een variabele van het type *char* kan slechts één teken (een Unicode teken) bevatten.
- Een variabele van het type *string* kan 0 of meer (Unicode) tekens bevatten.

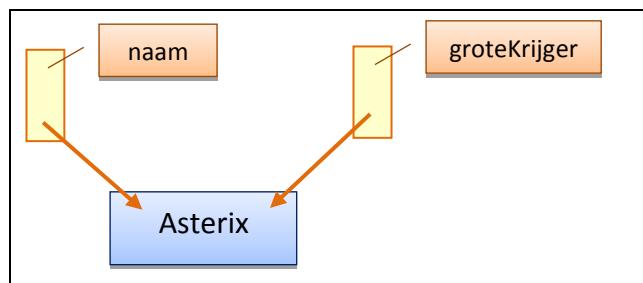
Variabelen van het type **string** zijn *reference* variabelen, geen *value* variabelen. De variabele bevat niet de tekst zelf, maar een verwijzing (reference) naar de geheugenplaats met de tekst.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string naam; (1)
            naam = "Asterix"; (2)
            string groteKrijger; (3)
            groteKrijger = naam; (4)
            Console.WriteLine(groteKrijger);
        }
    }
}
```

- (1) Je maakt een variabele *naam*. Dit is een reference variabele, maar deze bevat nog geen verwijzing (reference) naar een tekst.
- (2) Je laat de variabele *naam* verwijzen naar de tekst *Asterix* in het geheugen.
- (3) Je maakt een variabele *groteKrijger*. Dit is een reference variabele die nog geen verwijzing (reference) bevat.
- (4) Je laat de variabele *groteKrijger* verwijzen naar dezelfde tekst *Asterix* waar de variabele *naam* reeds naar verwijst.
- (5) Je toont de tekst waar de variabele *groteKrijger* naar verwijst, op het scherm.

De twee variabelen verwijzen naar dezelfde tekst in het geheugen:



In de CLR is het type **string** gekend als de class *String*. Je kan dus ook schrijven: **String naam="Asterix";**
string is een alias voor de class **String**.

9.2 Bewerkingen op strings

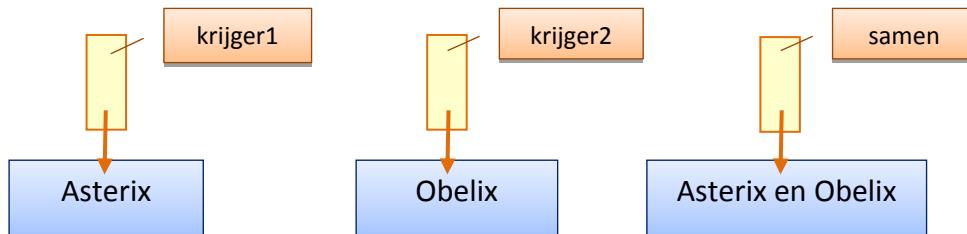
Met de + bewerking voeg je twee waardes samen tot één **string**.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string krieger1 = "Asterix";
            string krieger2 = "Obelix";
            string samen = krieger1 + " en " + krieger2;
            Console.WriteLine(samen);
        }
    }
}
```

Opmerking: je moet zelf spaties voorzien in de verschillende delen vóór en ná de + operator .

In het geheugen gebeurt het volgende:



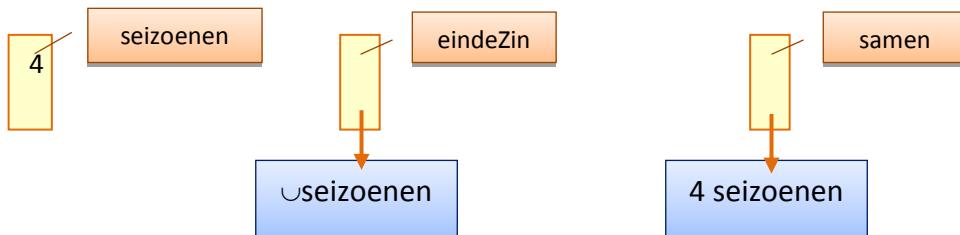
De waardes die je gebruikt bij de + operator moeten niet alle van het type **string** zijn. Als een waarde niet van het type **string** is, converteert C# deze waarde eerst naar het type **string**, en voegt daarna de waarden samen:

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int seizoenen = 4;
            string eindeZin = "useizoenen";
            string samen = seizoenen + eindeZin;
            Console.WriteLine(samen);
        }
    }
}
```

```
}
```

In het geheugen gebeurt het volgende:



Ook hier bestaat een verkorte versie van de + operator:

Korte schrijfwijze	Lange schrijfwijze	Betekenis
bericht+="daag";	bericht=bericht+"daag";	de tekst in de variabele bericht samenvoegen met de tekst daag.

	<p>Strings zijn “<i>immutable</i>”. D.w.z. dat éénmaal de string een waarde gekregen heeft, deze inhoud niet kan gewijzigd worden, ook al lijkt dit te gebeuren in het voorbeeld.</p> <pre>string bericht = "hallo";</pre> <p>De string variabele <i>bericht</i> verwijst naar de tekst “hallo” in het intern geheugen:</p> <p><i>bericht</i> += "iedereen";</p> <p>Hier gebeurt het volgende:</p> <p>Er wordt een nieuw string object gecreëerd met de tekst “hallo iedereen” en de string variabele <i>bericht</i> verwijst nu naar deze nieuwe tekst. Er is geen verwijzing (reference) meer naar de oorspronkelijke tekst “hallo”.</p> <p>Telkens wanneer de inhoud van een string variabele gewijzigd/uitgebreid wordt, wordt er dus een nieuw string object (een nieuwe tekst) in het geheugen gecreëerd, hetgeen geheugenruimte in beslag neemt.</p>
--	---

	<p>Om dit te vermijden kan je de class <i>StringBuilder</i> (uit de namespace <i>System.Text</i>) gebruiken i.p.v. de class <i>String</i> als de inhoud van de string vaak moet aangepast worden.</p> <p>Voorbeeld:</p> <pre>System.Text.StringBuilder bericht = new System.Text.StringBuilder("hallo"); Console.WriteLine(bericht); //toont hallo bericht.Append("iedereen"); Console.WriteLine(bericht); //toont hallo iedereen</pre> <p>Hier wordt de oorspronkelijke inhoud ("hallo") van de <i>StringBuilder</i> variabele <i>bericht</i> aangevuld met de tekst "iedereen", zonder een nieuw tekstobject te creëren.</p>
--	--

9.3 Strings vergelijken

- De `==` vergelijkingsoperator geeft enkel `true` als twee strings dezelfde inhoud hebben.
- De `!=` vergelijkingsoperator geeft enkel `true` als twee strings een verschillende inhoud hebben.
- 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)`
 - Als de `string` gelijk is aan `andereString`, krijg je 0.
 - Als de `string` alfabetisch vóór `andereString` komt, krijg je -1.
 - Als de `string` alfabetisch na de `andereString` komt, krijg je 1.

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

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string krieger1 = "Asterix";
            string krieger2 = "Obelix";
            string groteKrijger = "Asterix";
            string kleineKrijger = "asterix";
            Console.WriteLine(krieger1 == krieger2); (1)
            Console.WriteLine(krieger1 == groteKrijger); (2)
            Console.WriteLine(krieger1 == kleineKrijger); (3)
            Console.WriteLine(krieger1.CompareTo(krijger2)); (4)
        }
    }
}
```

(1) De twee `strings` hebben niet dezelfde inhoud, dus je krijgt `false`.

(2) De twee `strings` hebben dezelfde inhoud, dus je krijgt `true`.

- (3) De twee **strings** hebben dezelfde letters, maar bij de ene string is de a geschreven als hoofdletter en bij de andere string als kleine letter, dus je krijgt **false**.
- (4) De inhoud van krijger1 (Asterix) komt in het alfabet vóór de inhoud van krijger2 (Obelix), dus je krijgt -1.

9.4 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 <i>laatste</i> 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 <i>positie</i> . 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 <i>positie</i> over <i>aantal</i> posities. naam.Substring(2, 5) geeft terix.
Insert(positie, invoegtekst)	De tekst waarin op <i>positie</i> de <i>invoegtekst</i> is ingevoegd. naam.Insert(0, "De vrienden") geeft De vrienden Asterix en Obelix.
Remove(positie, aantal)	De tekst waarin op <i>positie aantal</i> tekens verwijderd zijn. naam.Remove(8,3) geeft Asterix Obelix.
Replace(oudetekst, nieuwetekst)	De tekst waarbij <i>oudetekst</i> vervangen is door <i>nieuwetekst</i> . 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.

	Als je één van deze methods toepast op een string , wijzigt die method deze string niet. De method geeft een nieuwe string terug met het gewijzigde beeld van de originele string . Zoals reeds eerder vermeld: eenmaal een string is aangemaakt, kan je deze niet meer wijzigen, een string is immutable.
---	--

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string naam = "Asterix en Obelix";
            Console.WriteLine(naam.ToUpper()); (1)
            Console.WriteLine(naam.ToLower()); (2)
            Console.WriteLine(naam); (3)
        }
    }
}
```

- (1) Je toont met de ToUpper() method de hoofdletterversie van de string.
- (2) Je toont met de ToLower() method de kleine letterversie van de string.
- (3) De string zelf is niet gewijzigd.

9.5 De individuele tekens van een string

Iedere individueel teken van een **string** heeft als type **char**.

Je leest één teken uit een **string** door na de naam van de **string** tussen vierkante haakjes het volgnummer te vermelden van de letter die je wil lezen. De eerste letter van een **string** krijgt het volgnummer 0.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string naam = "Asterix";
            char eersteLetter = naam[0];                                (1)
            Console.WriteLine(eersteLetter);
            char laatsteLetter = naam[naam.Length - 1];                (2)
            Console.WriteLine(laatsteLetter);
        }
    }
}
```

(1) Je vraagt de letter met het volgnummer 0 uit de string. Dit is de eerste letter (A).

(2) Je vraagt de lengte van de string (7). Hiervan trek je één af. Je vraagt dus de letter met het volgnummer 6 (x). Dit is de laatste letter van de string.

9.6 Value types converteren naar string

Ieder Value type heeft een method *ToString()*, die de inhoud van een variabele als een **string** waarde teruggeeft.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int kinderen = 7;
            string kinderenWoord = kinderen.ToString();
            Console.WriteLine(kinderenWoord);
        }
    }
}
```

9.7 Strings converteren naar Value types

9.7.1 Met de Convert class

Je converteert een **string** naar een Value type met 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

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string kinderenWoord = "7";
            byte kinderen = Convert.ToByte(kinderenWoord);
            Console.WriteLine(kinderen);
        }
    }
}
```

	Als de conversie mislukt (bvb. als de variabele <code>kinderenWoord</code> in het voorbeeld de waarde A bevat), krijg je een exception (fout). Je programma stopt op de lijn met de conversiefout. Later in de cursus zie je hoe je dit soort fouten kan opvangen, zodat je programma niet abrupt afbreekt.
---	---

9.7.2 Met de `Parse()` method

Elk Value type heeft een method `Parse()`. Met deze method kan je een string omzetten in het gewenste Value type.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string kinderenWoord = "7";
            byte kinderen = byte.Parse(kinderenWoord);
            Console.WriteLine(kinderen);
        }
    }
}
```

Deze conversiemethode verdient de voorkeur boven de conversie met de `Convert` class, omdat ze sneller is én intuïtiever in gebruik.

	Als de conversie mislukt (bvb. als de variabele <code>kinderenWoord</code> in het voorbeeld de waarde A bevat), krijg je een exception (fout). Je programma stopt op de lijn met de conversiefout. Later in de cursus zie je hoe je dit soort fouten kan opvangen, zodat je programma niet abrupt afbreekt.
---	---

9.8 Speciale tekens in een string

Bepaalde tekens kan je niet zomaar in een `string` gebruiken in de code, omdat deze tekens ook een andere betekenis hebben in C#. Om het teken toch als het letterlijke teken in een `string` te kunnen gebruiken in je code, gebruik je een formulering die begint met een *backslash* (\). Zo'n formulering heet een *escape sequence*.

In de volgende tabel vind je veel gebruikte *escape sequences*:

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**.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Dit is een dubbel aanhalingsteken:\".");
            Console.WriteLine("Standaard windows directory is" +
                "\\"c:\\windows");
            char aanhalingsteken = '\'';
            Console.WriteLine(aanhalingsteken);
        }
    }
}
```

(1)

- (1) Voor alle duidelijkheid: in deze opdracht staan voor de puntkomma twee enkele aanhalingstekens, niet één dubbel aanhalingsteken.

9.9 Verbatim strings

Als je een string in code laat voorafgaan door het @ teken is dit een *verbatim string*. In een verbatim string heeft bvb. het teken \ terug zijn letterlijke betekenis.

Om in een verbatim string een dubbel aanhalingsteken te vermelden, tik je twee dubbele aanhalingstekens na elkaar.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
```

```

    static void Main(string[] args)
    {
        Console.WriteLine(@"Dit is een dubbel aanhalingssteken: "".");
        Console.WriteLine(@"De standaard windows directory is c:\windows");
    }
}

```

	<p>Verbatim strings kan je splitsen over meerdere regels. In het volgende voorbeeld wordt de tekst letterlijk getoond zoals weergegeven in de code, dus op twee regels en met spaties:</p> <pre>Console.WriteLine(@"De standaard windows directory is c:\windows");</pre>
---	---

9.10 Een lege string

Je kan een lege **string** in je source op twee manieren aangeven:

- Als ""
- Als **string**.Empty
Hierbij gebruik je de constante *Empty* uit de class **string**.
Dit is een constante met class bereik. Je kan deze dus rechtstreeks via de class **string** aanroepen.

De opdracht **string** zangeresZonderNaam=""; en de opdracht **string** zangeresZonderNaam=**string**.Empty; declareren dus allebei een variabele zangeresZonderNaam als een **string** variabele en vullen die variabele in met een lege **string**. De tweede schrijfwijze (met **string**.Empty) wordt meestal beschouwd als beter leesbaar.

	<ul style="list-style-type: none"> • Om na te gaan of een string variabele <i>geen waarde</i> heeft (dus <i>null</i> is) of een <i>lege</i> string is, kan je de static method <i>IsNullOrEmpty(string eenString)</i> van de class string gebruiken. Voorbeeld: <pre>string tekst = null; Console.WriteLine(string.IsNullOrEmpty(tekst)); //geeft true tekst = string.Empty; Console.WriteLine(string.IsNullOrEmpty(tekst)); //geeft true tekst = "hallo"; Console.WriteLine(string.IsNullOrEmpty(tekst)); //geeft false</pre> • Om na te gaan of een string variabele <i>geen waarde</i> heeft (dus <i>null</i> is) of een <i>lege</i> string is of enkel <i>white-space tekens</i> (spatie, tab, enter) bevat, kan je de static method <i>IsNullOrEmptyWhiteSpace(string eenString)</i> van de class string gebruiken.
---	---

	<p>Voorbeeld:</p> <pre>string tekst = null; Console.WriteLine(string.IsNullOrWhiteSpace(tekst)); //geeft true tekst = string.Empty; Console.WriteLine(string.IsNullOrWhiteSpace(tekst)); //geeft true tekst = "\t\u0000\n"; Console.WriteLine(string.IsNullOrWhiteSpace(tekst)); //geeft true tekst = "hallo"; Console.WriteLine(string.IsNullOrWhiteSpace(tekst)); //geeft false</pre>
--	---

9.11 Een samengestelde string afbeelden

Met de “+” operator kan je meerdere waardes samenvoegen tot een string en als één string afbeelden (zie paragraaf BEWERKINGEN OP STRINGS).

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string krijger1 = "Asterix";
            string krijger2 = "Obelix";
            Console.WriteLine(krijger1 + "en" + krijger2 + "zijn goede vrienden.");
        }
    }
}
```

Je kan echter ook op de volgende manier een samengestelde string afbeelden:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string krijger1 = "Asterix";
            string krijger2 = "Obelix";
            Console.WriteLine("{0} en {1} zijn goede vrienden.",krijger1,krijger2);
        }
    }
}
```

- Als eerste parameter geef je bij de method `Console.WriteLine()` de af te beelden string mee. Deze samengestelde string bevat een combinatie van vaste tekst en genummerde parameters `{0}`, `{1}`, ..., zogenaamde placeholders. De nummering begint vanaf 0, de nummers staan tussen `{ }`.
- Vervolgens som je de variabelen/waarden op – gescheiden door een komma – die bij het afbeelden van de string in volgorde ingevuld worden in de genummerde parameters: `placeholder {0}` wordt vervangen door de waarde van `krijger1`, `placeholder {1}` wordt vervangen

door de waarde van *krijger2*, ...

Het *aantal* variabelen/waarden moet uiteraard gelijk zijn aan het aantal voorziene placeholders in de string.

Deze werkwijze verdient de voorkeur boven het “+” teken o.w.v. de leesbaarheid.

- Je kan de placeholders in de samengestelde string optioneel voorzien van een opmaakcode. Je plaatst deze code binnen de { }, ná het nummer van de placeholder, gescheiden van het nummer door een dubbele punt.

Voorbeeld:

```
DateTime datum = new DateTime(2014, 1, 1);
```



//zonder opmaakcode

```
Console.WriteLine("De datum is {0}", datum);  
→ toont "De datum is 1/01/2014 0:00:00".
```

//met opmaakcode

```
Console.WriteLine("De datum is {0:d MMMM yyyy}", datum);  
→ toont "De datum is 1 januari 2014".
```

Raadpleeg indien nodig de MSDN voor meer informatie betreffende de opmaakcodes voor getallen, datums, ...

10 Invoer vanaf het toetsenbord

10.1 Console.ReadLine()

Met de method *ReadLine()* van de class *Console* kan je een tekst, die de gebruiker met het toetsenbord intikt, inlezen. Dit is een **static** method (een method met class bereik).

De method *ReadLine()* wacht tot je een waarde intikt en op de *Enter*-toets drukt. De method geeft je daarna de ingetikte waarde als een **string** terug. Als je deze waarde aan een variabele van een ander type wil toekennen, moet je de waarde eerst converteren met één van de methods uit de *Convert* class of met de *Parse()* method.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const float CmInch = 2.54f;
        static void Main(string[] args)
        {
            Console.WriteLine("Tik een lengte in cm:");
            float cm = float.Parse(Console.ReadLine()); (1)
            Console.WriteLine("lengte in inch: {0}", cm / CmInch);
        }
    }
}
```

- (1) De waarde die de gebruiker intikt, is van het type **string**.

Je wil deze waarde invullen in de variabele *cm* die van het type **float** is.

Je converteert deze waarde naar het type **float** met de *Parse()* method.



Als de gebruiker een waarde intikt die C# niet naar een getal kan converteren (bvb. letters), stopt het programma met een fout. Later in de cursus leer je deze fout opvangen.

11 Selecties

11.1 if

Je gebruikt de opdracht **if** om één of meerdere opdrachten enkel uit te voeren als een bepaalde voorwaarde **true** is.

Schrijfwijze waarbij je één opdracht enkel uitvoert als een voorwaarde **true** is:

```
if (voorwaarde)
    opdracht
opdracht
```

C# voert deze opdracht enkel uit als de voorwaarde **true** is.
C# voert deze opdracht altijd uit.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tik je leeftijd:");
            int leeftijd = int.Parse(Console.ReadLine());
            Console.WriteLine("Je bent");
            if (leeftijd < 18)
                Console.WriteLine("Uwiet");
            Console.WriteLine("Utoegelaten.");
        }
    }
}
```

Als je meerdere opdrachten enkel wil uitvoeren als een voorwaarde **true** is, omsluit je deze opdrachten met *accooldes*:

```
if (voorwaarde)
{
    opdracht
    opdracht
}
opdracht
```

C# voert deze opdrachten enkel uit als de voorwaarde **true** is.
C# voert deze opdracht altijd uit.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tik je leeftijd:");
            int leeftijd = int.Parse(Console.ReadLine());
```

```

        if (leeftijd < 18)
    {
        Console.WriteLine("Je bent niet toegelaten.");
        Console.WriteLine("Kom later terug.");
    }
    Console.WriteLine("Dag");
}
}
}

```

11.2 if ... else ...

Je gebruikt een **if** **else** structuur om bepaalde opdrachten enkel uit te voeren als een voorwaarde **true** is en andere opdrachten enkel uit te voeren als dezelfde voorwaarde **false** is.

Ook hier geldt dat als je *meerdere* opdrachten enkel wil uitvoeren als de voorwaarde **true** is, je deze opdrachten omsluit met *accoades*. Bevat het **else** block meerdere opdrachten, dan omsluit je deze opdrachten eveneens met *accoades*.

```

if (voorwaarde)
    opdracht of { opdrachten } — C# voert deze opdracht(en) enkel uit als de voorwaarde true is.
else
    opdracht of { opdrachten } — C# voert deze opdracht(en) enkel uit als de voorwaarde false is.

```

Voorbeeld:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik je leeftijd:");
            int leeftijd = int.Parse(Console.ReadLine());
            if (leeftijd < 18)
            {
                Console.WriteLine("Niet toegelaten.");
                Console.WriteLine("Kom later terug.");
            }
            else
            {
                Console.WriteLine("Kom binnen.");
                Console.WriteLine("Enjoy the show!");
            }
        }
    }
}

```

Je kan **if** structuren nesten binnen andere **if** structuren:

```
if (voorwaarde1)
{
    opdracht
    opdracht
    opdracht
    if (voorwaarde2)
    {
        opdracht
        opdracht
        opdracht
    }
    else
    {
        opdracht
        opdracht
        opdracht
    }
}
else
{
    if (voorwaarde3)
    {
        opdracht
        opdracht
        opdracht
    }
    opdracht
    opdracht
    opdracht
}
```

C# voert deze opdrachten enkel uit als voorwaarde1 **true** is.

C# voert deze opdrachten enkel uit als voorwaarde1 **true** is en voorwaarde2 **true** is.

C# voert deze opdrachten enkel uit als voorwaarde1 **true** is en voorwaarde2 **false** is.

C# voert deze opdrachten enkel uit als voorwaarde1 **false** is en voorwaarde3 **true** is.

C# voert deze opdrachten enkel uit als voorwaarde1 **false** is.

Je kan ook de volgende syntax gebruiken voor **if ... else** opdrachten:

```
if (voorwaarde1)
{
    opdracht
}
else if (voorwaarde2)
{
    opdracht
}
else if (voorwaarde3)
{
    opdracht
}
else
{
    opdracht
}
```

C# voert deze opdrachten uit als voorwaarde1 **true** is, de overige opdrachten worden niet uitgevoerd.

C# voert deze opdrachten uit als voorwaarde2 **true** is, de overige opdrachten worden niet uitgevoerd.

C# voert deze opdrachten uit als voorwaarde3 **true** is, de overige opdrachten worden niet uitgevoerd.

C# voert deze opdrachten uit als aan geen enkele van de vorige voorwaarden voldaan is.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik een teken in: ");
            char c = (char)Console.Read(); (1)
            if (Char.ToUpper(c))
            {
                Console.WriteLine("Het teken is een hoofdletter.");
            }
            else if (Char.ToLower(c))
            {
                Console.WriteLine("Het teken is een kleine letter.");
            }
            else if (Char.IsDigit(c))
            {
                Console.WriteLine("Het teken is een cijfer.");
            }
            else
            {
                Console.WriteLine("Het teken is geen letter of cijfer.");
            }
        }
    }
}
```

- (1) Met de static method *Read()* van de class *Console* kan je één teken, dat de gebruiker via het toetsenbord intikt, inlezen. De method *Read()* wacht tot je een waarde intikt en op de *Enter*-toets drukt. De method geeft je daarna de ingetikte waarde als een *int* terug. Je kan deze waarde aan een variabele van het type *char* toekennen door deze waarde te converteren met de method *ToChar()* van de class *Convert* of met de typeconversie (*char*).

11.3 De ? operator

Sommige *if ... else* opdrachten kan je korter schrijven met de **?** operator.
De syntax van de **?** operator:

voorwaarde ? waarde1 : waarde2

- Als de voorwaarde *true* geeft, geeft het geheel van deze opdracht *warde1* terug.
- Als de voorwaarde *false* geeft, geeft het geheel van deze opdracht *warde2* terug.

Voorbeeld:

```
using System;
```

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik je leeftijd:");
            byte leeftijd = byte.Parse(Console.ReadLine());
            Console.WriteLine(leeftijd < 18 ? "Niet toegelaten" :
                "Kom maar binnen");
        }
    }
}

```

- Als de ingetikte leeftijd kleiner is dan 18, geeft de uitdrukking `leeftijd < 18 ? "Niet toegelaten" : "Kom maar binnen";` de waarde tussen `? en :` terug (Niet toegelaten).
- Als de ingetikte leeftijd niet kleiner is dan 18, geeft de uitdrukking `leeftijd < 18 ? "Niet toegelaten" : "Kom maar binnen";` de waarde na `:` terug (Kom maar binnen).

11.4 switch

Je gebruikt `switch` om één waarde (bvb. een variabele of het resultaat van een expressie) te vergelijken met verschillende vergelijkwaarden en per vergelijkwaarde andere opdrachten uit te voeren:

```

switch(expressie)
{
    case vergelijkwaarde1:
        opdracht
        ...
        break;
    case vergelijkwaarde2:
        opdracht
        ...
        break;
    default:
        opdracht
        ...
        break;
}

```

`opdracht` — C# voert deze opdrachten enkel uit als *expressie* gelijk is aan *vergelijkwaarde1*.

`opdracht` — C# voert deze opdrachten enkel uit als *expressie* gelijk is aan *vergelijkwaarde2*.

`opdracht` — C# voert deze opdrachten enkel uit als *expressie* verschilt van de vergelijkwaarden vermeld bij de vorige case(s). `default` is niet verplicht in een switch.

De expressie moet als type een *geheel getal, char, string, bool* of een *enum* hebben. Enums worden verder in de cursus behandeld.

Iedere *case* en ook de *default* dien je af te sluiten met het sleutelwoord ***break***, tenzij de laatste opdracht van een *case* of *default* de opdracht *return* (zie verder in de cursus) of de opdracht *throw* (zie verder in de cursus) is.

Een voorbeeldprogramma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik eerste getal:");
            decimal getal1 = decimal.Parse(Console.ReadLine());
            Console.Write("Tik tweede getal:");
            decimal getal2 = decimal.Parse(Console.ReadLine());
            Console.Write("Tik bewerking (+,-,/, : of *):");
            char bewerking = char.Parse(Console.ReadLine());
            switch (bewerking)
            {
                case '+':
                    Console.WriteLine(getal1 + getal2);
                    break;
                case '-':
                    Console.WriteLine(getal1 - getal2);
                    break;
                case '*':
                    Console.WriteLine(getal1 * getal2);
                    break;
                case '/':
                case ':':
                    if (getal2 == 0m)
                        Console.WriteLine("Kan niet delen door nul");
                    else
                        Console.WriteLine(getal1 / getal2);
                    break;
                default:
                    Console.WriteLine("Verkeerde bewerking ingetikt.");
                    break;
            }
        }
    }
}
```

	<p>Merk op dat in dit voorbeeld het programma een deling uitvoert als de gebruiker / of : intikt. Als je één of meerdere (dezelfde) opdrachten wil uitvoeren bij meer dan één vergelijkingswaarde (delen bij / en : als vergelijkingswaarde), vermeld je deze als aaneensluitende case opdrachten. Enkel de laatste van deze case opdrachten mag code bevatten. In dit voorbeeld mag de case '/' geen code bevatten, wel de case ':', want het is de laatste in een reeks aaneengesloten case opdrachten.</p>
---	---



oefeningen: Selecties

12 Iteraties

12.1 while

Je gebruikt een **while** structuur om één of meerdere opdrachten uit te voeren zolang een bepaalde voorwaarde **true** is.

Je kan **while** en de voorwaarde vóór de te herhalen opdracht(en) schrijven:

```
while (voorwaarde)
    opdracht of {opdrachten}
```

C# test de voorwaarde. Als deze **true** is, voert C# de iteratie uit. C# test opnieuw de voorwaarde. Als ze **true** is, voert C# de iteratie opnieuw uit. Pas als de voorwaarde **false** wordt, gaat C# verder met de rest van het programma.

Je kan ook eerst **do** schrijven, gevolgd door de te herhalen opdracht(en), gevolgd door **while** en de voorwaarde:

```
do
    opdracht of {opdrachten}
while (voorwaarde);
```

C# voert de iteratie zeker één keer uit en test dan pas de voorwaarde. Als deze **true** is, voert C# de iteratie nog eens uit. Pas als de voorwaarde **false** wordt, gaat C# verder met de rest van het programma.

Een voorbeeld met de voorwaarde vóór de te herhalen opdrachten. Het programma berekent het totaal van een reeks ingetikte getallen.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal, totaal = 0m;
            Console.WriteLine("Tik een reeks getallen, stop de reeks met 0:");
            while ((getal = decimal.Parse(Console.ReadLine())) != 0m)          (1)
                totaal += getal;
            Console.WriteLine("Totaal: {0}", totaal);
        }
    }
}
```

(1) Op deze regel gebeuren meerdere dingen:

- De gebruiker tikt iets in op het toetsenbord en drukt Enter.
`Console.ReadLine()`
- Je converteert de ingetikte waarde naar een **decimal**.
`decimal.Parse()`
- Je kent deze waarde toe aan de variabele *getal*.
`getal=`

- Je test of de variabele getal verschilt van nul.
 $\neq 0m$
Met het resultaat van deze test (**true** of **false**) stuur je de iteratie aan.

Hetzelfde voorbeeld met de voorwaarde ná de te herhalen opdrachten:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal, totaal = 0m;
            Console.WriteLine("Tik een reeks getallen, stop de reeks met 0:");
            do
                totaal += getal = decimal.Parse(Console.ReadLine());                                (1)

                while (getal != 0m);
                Console.WriteLine("Totaal: {0}" ,totaal);
            }
        }
    }
}
```

(1) Op deze regel gebeuren meerdere dingen:

- De gebruiker tikt iets in op het toetsenbord en drukt Enter.
`Console.ReadLine()`
- Je converteert de ingetikte waarde naar een **decimal**.
`decimal.Parse()`
- Je kent deze waarde toe aan de variabele getal.
`getal=`
- Je telt de inhoud van getal bij bij de variabele totaal.
`totaal+=`

12.2 for

Je gebruikt een **for** iteratie om een aantal opdrachten een aantal keer uit te voeren.

Dit doe je door eerst één of meerdere variabelen te initialiseren, vervolgens een voorwaarde te testen, en zolang die voorwaarde **true** is één of meerdere opdrachten uit te voeren. Bij iedere iteratie verhoog of verlaag je één of meerdere variabelen.

```
for(initialisatie(s); voorwaarde; verhoging(en) en/of verlagingen)
    opdracht of {opdrachten} →
```

C# voert de initialisatie(s) uit. Meerdere initialisaties scheid je met een komma.
C# test de voorwaarde. Zolang deze voorwaarde **true** is, voert C# de opdracht(en) uit.
Bij iedere iteratie voert C# de verhoging(en) en/of verlaging(en) uit.
Meerdere verhoging(en) en/of verlaging(en) scheid je met een komma.

Een voorbeeldprogramma: je berekent het totaal van 10 ingetikte getallen.
Je gebruikt één initialisatie en één verhoging.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal totaal = 0m;
            for (int teller = 1; teller < 11; teller++) (1)
            {
                Console.WriteLine("Tik getal {0}: ", teller);
                totaal += decimal.Parse(Console.ReadLine());
            }
            Console.WriteLine("Totaal: {0}", totaal);
        }
    }
}
```

(1) Als C# de eerste keer de iteratie binnentkomt, declareer je een variabele teller van het type `int` en geef je deze variabele een beginwaarde 1.

Gezien je de variabele in de `for` declareert, is ze enkel binnen die `for` gekend.

`int teller=1`

Bij iedere iteratie test C# of de variabele teller kleiner is dan 11.

`teller<11`

Op het einde van iedere iteratie verhoogt C# de variabele teller met één.

`teller++`

Een voorbeeldprogramma: je onderzoekt of een ingetikt woord een palindroom is (een woord dat je op dezelfde manier kan lezen van voor naar achter als van achter naar voor, zoals de woorden lepel en parterretrap).

Je gebruikt twee initialisaties en twee verhogingen.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tik een woord:");
            string woord = Console.ReadLine();
            bool palindroom = true;
            for (int vanafBegin = 0, vanafEinde = woord.Length - 1; (1)
                 vanafBegin < vanafEinde; vanafBegin++, vanafEinde--)
                if (woord[vanafBegin] != woord[vanafEinde])
                    palindroom = false;
            if (palindroom)
                Console.WriteLine("Dit woord is een palindroom");
        }
    }
}
```

```
        else
            Console.WriteLine("Dit woord is geen palindroom");
    }
}
```

- (1) Als C# de eerste keer de iteratie binnentreedt, declareer je een variabele *vanafBegin* van het type **int** en geef je deze variabele een beginwaarde 0. Op dit moment declareer je ook een variabele *vanafEinde* en geef je deze variabele een beginwaarde lengte van het woord min één. Gezien je deze variabelen in de **for** declareert, zijn ze enkel binnendie **for** gekend.

int vanafBegin=0, vanafEinde=woord.Length-1

Als je meerdere variabelen initialiseert, moeten ze hetzelfde type hebben.

Bij iedere iteratie test C# of *vanafBegin* kleiner is dan *vanafEinde*.

vanafBegin<*vanafEinde*

Op het einde van iedere iteratie verhoogt C# *vanafBegin* met één

(*vanafBegin*++) en verlaagt C# *vanafEinde* met één (*vanafEinde*--).

Als je het woord appelt, vergelijkt het programma bij de eerste iteratie de letter a met de letter l. Op dit moment staat al vast dat het woord geen palindroom is. Toch zal het programma in een tweede (overbodige) iteratie de letter p met de letter e vergelijken.

Je kan de **for** lus voortijdig verlaten met de **break** opdracht, zodra je merkt dat het woord geen palindroom is:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik een woord:");
            string woord = Console.ReadLine();
            bool palindroom = true;
            for (int vanafBegin = 0, vanafEinde = woord.Length - 1;
                 vanafBegin < vanafEinde; vanafBegin++, vanafEinde--)
                if (woord[vanafBegin] != woord[vanafEinde])
                {
                    palindroom = false;
                    break;
                }
            if (palindroom)
                Console.WriteLine("Dit woord is een palindroom");
            else
                Console.WriteLine("Dit woord is geen palindroom");
        }
    }
}
```



Je kan ook andere iteraties (**while**, **do ... while**) voortijdig verlaten met de **break** opdracht.

Door de **if** voorwaarde die binnen de **for** staat toe te voegen aan de voorwaarde van de **for** zelf, kan je het programma korter schrijven:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tik een woord:");
            string woord = Console.ReadLine();
            int vanafBegin, vanafEinde; (1)
            for (vanafBegin = 0, vanafEinde = woord.Length - 1;
                 vanafBegin < vanafEinde && woord[vanafBegin] == woord[vanafEinde]; (2)
                 vanafBegin++, vanafEinde--);
            if (woord[vanafBegin] == woord[vanafEinde])
                Console.WriteLine("Dit woord is een palindroom");
            else
                Console.WriteLine("Dit woord is geen palindroom");
        }
    }
}
```

- (1) Deze keer declareer je deze variabelen vóór de **for** iteratie, omdat je deze variabelen nog aanspreekt ná de **for** iteratie.
- (2) Het vergelijken van de letters stuurt nu mee de voorwaarde van de **for** iteratie zelf aan. Bemerk de ; na de ronde haakjes van de **for**: de for itereert zonder enige opdracht binnen de for iteratie zelf.

De drie onderdelen van de **for** (initialisatie(s), voorwaarde, verhoging(en) en/of verlaging(en)) zijn elk op zich optioneel. Het volgende programma geeft bij uitvoering een eindeloze iteratie, die je met **Ctrl+C** kan afbreken.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            for ( ; ; )
                Console.WriteLine("En ze draaiden maar door ...");
        }
    }
}
```



oefeningen: Iteraties

13 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 binnennin een **if, else, switch, while** of **for** declareert, is deze variabele enkel gekend tot het einde van die **if, else, switch, while** of **for**.

Een voorbeeldprogramma dat het aantal spaties in een zin telt:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Tik een zin:");
            string naam = Console.ReadLine(); (1)
            int aantalSpaties = 0; (2)
            for (int teller = 0; teller < naam.Length; teller++) (3)
                if (naam[teller] == ' ')
                    aantalSpaties++;
            Console.WriteLine(aantalSpaties);
        }
    }
}
```

- (1) Deze variabele *naam* is gekend vanaf hier tot het einde van de method Main.
- (2) Deze variabele *aantalSpaties* is gekend vanaf hier tot het einde van de method Main.
- (3) Deze variabele *teller* is enkel binnen de **for** iteratie gekend.

14 Arrays

14.1 Eén-dimensionele arrays

Een één-dimensionele array (tabel) is een rij veranderlijke waarden (variabelen), die **van hetzelfde type** zijn.

Je bereikt elk element uit de array via zijn uniek indexnummer (volgnummer).

Het eerste element krijgt het indexnummer nul.

Je kan je een array van 12 elementen als volgt voorstellen:



Je declareert een array als volgt:

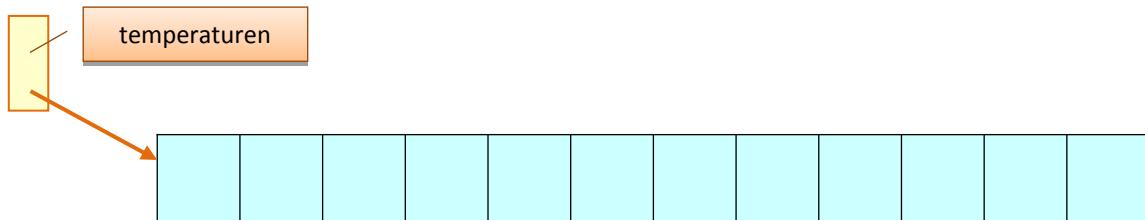
- het type van de elementen van de array, gevolgd door []
(ieder element in een array moet hetzelfde type hebben)
- de naam van de array
(hier gelden dezelfde regels als bij de naamgeving van een gewone variabele)
- het = teken
- het sleutelwoord **new**, terug gevolgd door het type van de elementen van de array, gevolgd door het aantal elementen van de array tussen vierkante haakjes (maximaal 2^{64}).

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 een reference variabele. De variabele bevat zelf niet de array, maar een verwijzing naar de array in het intern geheugen:



Je kan ook eerst de reference variabele declareren, die nog “nergens” naar verwijst. Daarna maak je met een aparte opdracht de array in het geheugen en verwiss je met de reference variabele naar deze array:

double[] temperaturen; (1)

temperaturen=new double[12]; (2)

- (1) Je maakt de reference variabele. Deze verwijst nog niet naar een array.
- (2) Je maakt de array met het sleutelwoord **new**. Je geeft tussen vierkante haakjes het aantal elementen aan. Je laat de reference variabele temperaturen naar de array verwijzen met **temperaturen=**.

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.

De schrijfwijze om naar één element van een array te verwijzen:

naam van de array, gevolgd door het volgnummer van het element tussen vierkante haakjes.

Bvb. verwijzen naar het eerste element uit de array temperaturen:

`temperaturen[0]`

Als je naar een niet-bestand element uit een array (een te laag of te hoog volgnummer) verwijst, stopt het programma met een fout (een exception).

Een voorbeeldprogramma waarmee je per maand een temperatuur bijhoudt:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const int AantalMaanden = 12;
        static void Main(string[] args)
        {
            double[] temperaturen = new double[AantalMaanden];
            int maand;
            for (maand = 1; maand <= AantalMaanden; maand++)
            {
                Console.Write("Tik de temperatuur voor maand {0}: ", maand);
                temperaturen[maand - 1] = double.Parse(Console.ReadLine());
            }
            Console.WriteLine("Je tikte volgende temperaturen:");
            for (maand = 1; maand <= AantalMaanden; maand++)
                Console.WriteLine("Maand {0}: {1}", maand, temperaturen[maand - 1]);
        }
    }
}
```



De *Length* property van een array geeft je het aantal elementen:

`Console.WriteLine(temperaturen.Length); → 12`

14.2 De elementen van een array initialiseren bij de declaratie

Je kan bij de declaratie van de array een beginwaarde voor de elementen meegeven. Je vermeldt het type van de array, gevolgd door vierkante haakjes, gevolgd door de naam van de array, gevolgd door =, gevolgd door de beginwaarden van de elementen van de array tussen accolades. Je scheidt de beginwaarden met een komma. De array bevat evenveel elementen als je beginwaarden opgeeft.

```
type[] arrayNaam={waarde 1° element,waarde 2° element,...};
```

Een voorbeeldprogramma dat maanden en dagen per maand toont voor een niet-schrikkeljaar:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const int AantalMaanden = 12;
        static void Main(string[] args)
        {
            int[] dagenPerMaand = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
            for (int maand = 0; maand < dagenPerMaand.Length; maand++)
                Console.WriteLine(dagenPerMaand[maand]);
        }
    }
}
```

	<p>.NET bevat kennis over het aantal dagen per maand. De method <i>DaysInMonth(eenJaar,eenMaand)</i> van het type <i>DateTime</i> geeft je het aantal dagen van de maand, aangeduid door de parameters <i>eenJaar</i> en <i>eenMaand</i>. Deze method houdt automatisch rekening met schrikkeljaren.</p>
---	--

Dit programma toont het aantal dagen van de maanden van het huidige jaar:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        const int AantalMaanden = 12;
        static void Main(string[] args)
        {
            int huidigJaar = DateTime.Today.Year;
            for (int maand = 1; maand <= 12; maand++)
                Console.WriteLine(DateTime.DaysInMonth(huidigJaar, maand));
        }
    }
}
```

14.3 foreach

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

```
foreach (type variabeleNaam in array)
    opdracht of {opdrachten}
```

De variabele die je bij **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 1° iteratie de waarde van het 1° element van de array.

De variabele bevat bij de 2° iteratie de waarde van het 2° element van de array, ...

De iteratie stopt automatisch als alle elementen van de array doorlopen zijn.

Je kan met **foreach** de elementen enkel lezen, niet wijzigen.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] stripHelden = {"Asterix", "Obelix", "Idefix"};
            foreach (string stripHeld in stripHelden)
                Console.WriteLine(stripHeld);
        }
    }
}
```

14.4 Het aantal elementen van een array als variabel gegeven

Het aantal elementen, dat je meegeeft bij de aanmaak van de array, mag ook een variabele of berekening zijn. Je kan dus tijdens de uitvoering van het programma beslissen hoeveel elementen de array moet bevatten.

Als voorbeeld onthoud je de nummers van bankrekeningen in een array. Gezien je bij het ontwerp van het programma niet kan voorspellen hoeveel rekeningen iemand heeft, vraag je eerst het aantal rekeningen. Pas daarna maak je een array met het benodigde aantal elementen.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Aantal bankrekeningen:");
            int aantalRekeningen = int.Parse(Console.ReadLine()); (1)
            ulong[] rekeningNrs = new ulong[aantalRekeningen]; (2)
```

```
for (int teller = 1; teller <= aantalRekeningen; teller++)
{
    Console.Write("Nummer van {0}º rekening: ", teller);
    rekeningNrs[teller - 1] = ulong.Parse(Console.ReadLine());
}
Console.WriteLine("Dit zijn de nummers van je rekeningen:");
foreach (ulong rekeningNr in rekeningNrs)
    Console.WriteLine(rekeningNr);
}
```

- (1) Je vraagt het aantal rekeningen.
 - (2) Je maakt een array met evenveel elementen als het aantal rekeningen.
 - (3) Je laat ieder rekeningnummer intikken.
 - (4) Je toont ieder rekeningnummer.

14.5 Twee-dimensionele arrays

Een twee-dimensionele array is een matrix (met rijen en kolommen).

Je kan je een array van 4 op 7 elementen als volgt voorstellen:

0							
1							
2							
3							
	0	1	2	3	4	5	6

De declaratie van deze array waarbij ieder element van het type double is:

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

Naast twee-dimensionele arrays bestaan nog drie-dimensionele arrays, ...

Als je bij de declaratie van een twee-dimensionele array beginwaarden wil meegeven voor de elementen van de array, moet je alle beginwaarden omsluiten met accolades, en iedere rij beginwaarden nog eens omsluiten met accolades.

Je scheidt iedere rij met een komma:

```
string[,] hoofdstedenLanden={{"Cuba","Havana","Spaans"},  
 {"Brazilië","Brasilia","Portugees"}};
```

geeft de volgende array:

Cuba	Havana	Spaans
Brazilië	Brasilia	Portugees

Je bereikt elk element uit de array via de combinatie van zijn rijnummer en kolomnummer. Het eerste element krijgt als rijnummer én als kolomnummer nul.

De uitdrukking `hoofdstedenLanden[1,1]` geeft bijvoorbeeld `Brasilia`.

14.6 De class Array

Een class is een prototype van iets. Later in de cursus wordt dit in detail behandeld.

De class `Array` is het prototype van alle arrays die je maakt.

De class `Array` bevat static methods (methods met class bereik) waarmee je arrays kan doorzoeken, sorteren, ...:

- **Sort(array)**
sorteert de één-dimensionale array `array` oplopend.
- **Reverse(array)**
wisselt de elementen in de één-dimensionale 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.
 - parameter `aantal`:
het aantal te kopiëren elementen.
- **Copy(vanArray,vanafElement,naarArray,naarElement, aantal)**
Kopieert elementen van de array `vanArray` naar de array `naarArray`.
 - parameter `vanafElement`:
vanaf welk element uit de array `vanArray` begint het kopiëren.
 - parameter `naarElement`:
vanaf welk element uit de array `naarArray` begint het kopiëren.
 - parameter `aantal`:
het aantal te kopiëren elementen.
- **IndexOf(array,zoekWaarde)**
Geeft het volgnummer van het eerste element uit de één-dimensionale 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-dimensionale 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`, ...)

 - parameter `vanaf`:
het volgnummer van het element waar het initialiseren begint.
 - parameter `aantal`:
aantal te initialiseren elementen vanaf het element met volgnummer `vanaf`.

Een voorbeeldprogramma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] namen = { "Asterix", "Obelix", "Idefix" };
            Console.WriteLine("Beginsituatie:");
            foreach (string naam in namen)
                Console.WriteLine(naam);
            Array.Sort(namen); (1)
            Console.WriteLine("Na Sort:");
            foreach (string naam in namen)
                Console.WriteLine(naam);
            Array.Reverse(namen); (2)
            Console.WriteLine("Na Reverse:");
            foreach (string naam in namen)
                Console.WriteLine(naam);
            string[] kopie = new string[namen.Length]; (3)
            Array.Copy(namen, kopie, namen.Length); (4)
            Console.WriteLine("Kopie:");
            foreach (string naam in kopie)
                Console.WriteLine(naam);
            Console.WriteLine("1° Idefix: " + (5)
                Array.IndexOf(namen, "Idefix")); (6)
        }
    }
}
```

- (1) Je sorteert de array.
- (2) Je wisselt de elementen van de array om.
- (3) Je maakt een array met evenveel elementen als de array `namen`.
- (4) Je kopieert alle elementen uit de array `namen` naar de array `kopie`.
- (5) Je toont de kopie.
- (6) Je vraagt het volgnummer van het 1° element dat gelijk is aan “Idefix”.



oefening: Arrays

15 Enum

15.1 Algemeen

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.

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 (als het gaat over een blad in een printer).

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.

Je kan een **enum** laten voorafgaan door een sleutelwoord dat het toegangsniveau van deze **enum** aangeeft. Je kan één van de volgende sleutelwoorden gebruiken:

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).

	<ul style="list-style-type: none">• 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.
---	--

Maak als voorbeeld een **enum** *Seizoen* met als mogelijke waarden *Lente, Zomer, Herfst* en *Winter*:

```
using System;
namespace CSharpPFCursus
{
    public enum Seizoen          (1)
    {
        Lente, Zomer, Herfst, Winter   (2)
    }                                (3)

    class Program
    {

        static void Main(string[] args)
        {
            Seizoen plukseizoen = Seizoen.Herfst;      (4)
            Console.WriteLine(plukseizoen);             (5)
            Console.WriteLine((int)plukseizoen);         (6)
        }
    }
}
```

- (1) Je begint de definitie van een **enum** met het sleutelwoord **enum**, gevolgd door de naam van de **enum** en een open accolade.
- (2) Je definieert de mogelijke waarden van de **enum**, gescheiden door komma's.
- (3) Je sluit de definitie van de **enum** af met een sluit accolade.
- (4) 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.
- (5) Je ziet de naam van de waarde uit de **enum**.
- (6) 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
}
```

16 Classes, Objects en object variables

16.1 Algemeen

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#.

16.2 Class (Klasse)

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.

In een banktoepassing beschrijf je de classes *Klant*, *Rekening*, *Overschrijving*, ...

In een bibliotheektoepassing beschrijf je de classes *Lezer*, *Boek*, *Auteur*, *Uitgever*, *Ontlening*, ...

Je beschrijft in een class de **eigenschappen (properties)** van het gegevenstype. In de class *Boek* van een bibliotheek beschrijf je bvb. de eigenschappen *ISBN-nr*, *titel*, *auteur*, *genre*, *prijs*, ...

Naast eigenschappen beschrijf je in een class ook **handelingen (methods)** die het gegevenstype kan uitvoeren. In de class *Rekening* van een bank beschrijf je bvb. de methods *Storten* en *Afhalen*. Methods kunnen parameters hebben. De method *Storten* kan de parameter *bedragTeStorten* bevatten.

16.3 Object

Een class is enkel een beschrijving (een prototype) van een gegevenstype.

Nadat je een class gedefinieerd hebt, maak je *objecten* op basis van die class. Objecten krijgen plaats in het intern geheugen. Hoeveel intern geheugen ze krijgen, is afhankelijk van het aantal en de grootte (o.a. bepaald door het type) van de properties (eigenschappen) die je in de bijbehorende class gedefinieerd hebt.

16.4 Reference Variabele

Je verwijst in je code naar een object via een *reference variabele* (ook object variabele genoemd). Je kan via de reference variabele

- de properties (eigenschappen) van het object invullen en opvragen.
- de methods (handelingen) van het object uitvoeren.

16.5 Een class creëren

Maak als voorbeeld een class *Werknemer* die een werknemer beschrijft.

Beschrijf in de class *Werknemer* de volgende properties van een werknemer: *naam*, *geslacht* (man of vrouw) en *indienst* (datum van indiensttreding).

Microsoft raadt aan de naam van een class met een hoofdletter te beginnen, en ieder apart woord binnen de naam van de class terug 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 (*Werknemer.cs*) aan en plaatst de grove structuur van de class in deze source file:

```
using System; (1)
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpPFCursus (2)
{
    class Werknemer (3)
    {
    } (4)
}
```

(5)

- (1) De opdrachten beginnend met **using** geven aan dat je in deze applicatie classes wil aanspreken uit de namespaces *System*, *System.Collections.Generic*, *System.Linq*, *System.Text* en *System.Threading.Tasks*, zonder voor elk van deze classes de volledige namespacenaam te schrijven.
De namespace *System* bevat classes die je regelmatig nodig hebt, zoals de class *Console*, waarmee je tekst op het scherm toont.
- (2) 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), als ze maar ingekapseld is in een namespace met een naam die verschilt van *CSharpPFCursus*.
- (3) Je kan de definitie van een class beginnen met een sleutelwoord dat het toegangsniveau van de class aangeeft. Dit zijn dezelfde toegangsniveau-sleutelwoorden als uitgelegd in het hoofdstuk ENUM. Indien er geen toegangsniveau is aangegeven, dan is de class **internal**. Dit betekent dus

dat je de class enkel kan gebruiken in de assembly waarin je de class declareert. Wil je de class ook buiten de assembly kunnen gebruiken, dan moet je het sleutelwoord **public** voorzien.

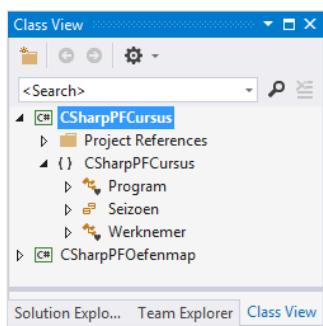
- (4) Een open accolade opent de eigenlijke definitie van de class.
- (5) Een sluit accolade sluit de definitie van de class.

16.6 De Class View

VS biedt je een grafische visie op de classes van je applicatie.

Kies het menu *VIEW – Class View*.

Je ziet in een venster (normaal rechts op het scherm) de opbouw van je applicatie:



Je ziet de volgende symbolen:

- | | |
|--|----------------------------------|
| | Project: <i>CSharpPFCursus</i> |
| | NameSpace: <i>CSharpPFCursus</i> |
| | Class: <i>Program, Werknemer</i> |
| | Enum: <i>Seizoen</i> |

16.7 Een property aan de class toevoegen

Per property voorzie je in de class drie onderdelen:

- Een interne (**private**) variabele waarin je de waarde van de property bijhoudt. Enkel methods binnen de class kunnen deze variabele aanspreken (data hiding, zie cursus Objectgeoriënteerde principes).

Voor de property *InDienst* maak je bvb. een private *DateTime* variabele *inDienstValue*. Met het sleutelwoord **private** bescherm je de variabele:

een programmeur die een *Werknemer* object creëert of gebruikt, kan deze variabele **niet** rechtstreeks wijzigen door bvb. te schrijven *ik.inDienstValue = waarde;*.

Zo verhinder je dat hij in de variabele *inDienstValue* bijvoorbeeld een ongeldige waarde (bvb. 1/1/1200) invult.

Deze bescherming van de properties van een class is essentieel om beveiligde classes te creëren. Als je de variabele *inDienstValue* **public** zou declareren, kan een programmeur die een *Werknemer* object creëert of gebruikt, deze variabele wel rechtstreeks invullen, zonder dat er gecontroleerd wordt welke waarde die programmeur invult. Dit is niet bevorderlijk voor de stabiliteit en de verdere werking van het programma.

- Een beveiligde toegangsweg waarmee een programmeur de waarde van de property kan invullen.
Deze toegangsweg begin je met het sleutelwoord **set**. Na **set** tik je tussen accolades de code om de property in te vullen. Deze code controleert

o.a. of de nieuwe waarde een geldige waarde is, zoniet wordt de nieuwe waarde niet geaccepteerd.

De **set** heeft dus als grove structuur:

```
set
{
}
```

C# roept de **set** automatisch op als een programmeur de property van een Werknemer object invult (bvb. `ik.InDienst=new DateTime(2014,1,1);`).

De **set** krijgt de in te vullen waarde voor de property binnen via het sleutelwoord **value**. In de **set** kan je deze waarde controleren vóór je ze invult in de **private** variabele *inDienstValue*. Je kan in de **set** bvb. enkel waardes toelaten die groter zijn dan of gelijk zijn aan de opstartdatum van de firma (1/1/1980).

- Een toegangsweg waarmee een programmeur de waarde van de property kan opvragen. Deze toegangsweg begin je met het sleutelwoord **get**. Na **get** tik je tussen accolades de code om de property op te halen. De **get** heeft dus als grove structuur:

```
get
{
}
```

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 (bvb. `Console.WriteLine(ik.InDienst);`).



Samenvatting: de **set** en de **get** zijn beveiligde toegangswegen naar de **private** variabele die de waarde van de property bijhoudt.

Op zich zijn de **set** en de **get** in de source van je class genest in een andere structuur. Deze structuur noemt men *property*.

Deze structuur begint met een sleutelwoord dat het toegangsniveau van de property aangeeft (**public**, **private**,...), gevolgd door het type van de property, gevolgd door de naam van de property.

Deze structuur omsluit de **set** en de **get** met accolades:

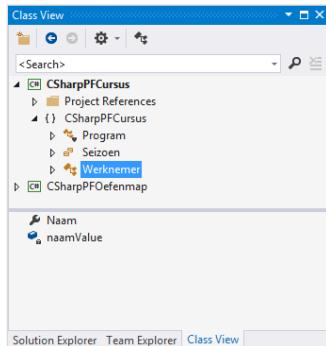
```
public typeVanDeProperty naamVanDeProperty
{
    get
    {
    }
    set
    {
    }
}
```

Maak de 3 onderdelen voor de property *Naam*:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        private string naamValue;
        public string Naam
        {
            get
            {
                return naamValue;
            }
            set
            {
                if (value != string.Empty)
                    naamValue = value;
            }
        }
    }
}
```

- (1) Het toegangs niveau van de class declareren we als **public**.
- (2) Ieder object van het type *Werknemer* heeft in het intern geheugen zijn eigen variabele *naamValue*. Zo onthoudt ieder *Werknemer* object zijn eigen naam.
- (3) De beschrijving van de property *Naam*. Deze property krijgt als type **string**. Normaal kies je hetzelfde type als de **private** variabele die de waarde van de property onthoudt (*naamValue*).
- (4) Een **get** routine binnen de property.
C# roept deze routine op als een programmeur de property *Naam* van een *Werknemer* object opvraagt (bvb. `Console.WriteLine(ik.Naam);`). Dan wordt de waarde van de **private** variabele *naamValue* opgehaald en deze waarde wordt vanuit de class naar de buitenwereld gestuurd met de **return** opdracht.
- (5) Een **set** routine binnen de property.
C# roept deze routine op als een programmeur de property *Naam* van een *Werknemer* object invult (bvb. `ik.Naam="Asterix";`). De in te vullen waarde komt binnen via het sleutelwoord **value**. Deze waarde kan eerst op geldigheid gecontroleerd worden. Zo wordt er hier voor gezorgd dat de naam geen lege string is. Enkel een goedgekeurde waarde wordt onthouden in de bijbehorende **private** variabele *naamValue*.

In de *Class View* zie je ook deze toevoegingen aan de class:



Je ziet de volgende extra symbolen:

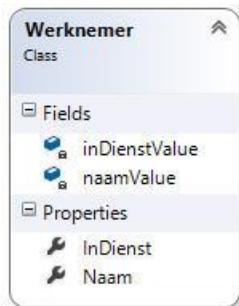
- Property:** *Naam*
- Private variabele:** *naamValue*

Maak de 3 onderdelen voor de property *InDienst*.

Deze keer tik je de code niet zelf in, maar gebruik je een *Class Diagram*, een grafische voorstelling van de class:

- Klik in de *Solution Explorer* of in de *Class View* met de rechtermuisknop op de class *Werknemer*.
- Kies in het popup menu de opdracht *View Class Diagram*.
- Klik met de rechtermuisknop op het vak *Werknemer*.
- Kies in het vervolgmenu de opdracht *Add* en vervolgens *Field*.
- Tik de naam van het Field in: *inDienstValue*.
- In het propertiesvenster kies je bij *Access* voor *private*.
- Tik bij *Type* het type van de variabele: *DateTime*.
- Klik met de rechtermuisknop bovenaan in het vak *Werknemer*.
- Kies in het vervolgmenu de opdracht *Add* en vervolgens *Property*.
- Tik de naam van de property in: *InDienst*.
- In het propertiesvenster kies je bij *Access* voor *public*.
- Tik bij *Type* het type van de property: *DateTime*.

Het Class Diagram ziet er als volgt uit:



Vervolledig in de source van de class *Werknemer* de property *InDienst* als volgt:

```
public DateTime InDienst
{
    get
    {
        return inDienstValue;
    }
    set
    {
        inDienstValue = value;
    }
}
```

De lijn code `throw new System.NotImplementedException()` mag je verwijderen. Exceptions komen pas later in deze cursus aan bod.

De volgende property die je toevoegt aan de class *Werknemer*, is de property *Geslacht*. Geslacht is een ideaal type om als `enum` te beschrijven: een geslacht kan enkel *man* of *vrouw* bevatten.

Je beschrijft deze `enum` in een aparte source file:

- Kies in het menu *PROJECT* de opdracht *Add New Item...* of klik in de Solution Explorer met de rechtermuisknop op de naam van het project *CSharpPFCursus* en kies de opdracht *Add – New Item...*.
- Kies bij *Templates voor Code File*.
- Tik bij *Name* de naam van het nieuwe bronbestand: *Geslacht*.
- Klik op de knop *Add*.

Tik in dit nieuwe bronbestand de volgende code:

```
namespace CSharpPFCursus
{
    public enum Geslacht
    {
        Man, Vrouw
    }
}
```



Je declareert *Geslacht* in dezelfde namespace als de class *Werknemer* en de class *Program* (die op zich de routine *Main* bevat).

Je voegt aan de class *Werknemer* de onderdelen toe voor een property *Geslacht* van het type *Geslacht*.

- Klik in het *Class Diagram* met de rechtermuisknop op de class *Werknemer*.
- Kies in het vervolgmenu de opdracht *Add* en vervolgens *Field*.
- Tik de naam van het Field in: *geslachtValue*.
- In het propertiesvenster kies je bij *Access* voor *private*.
- Tik bij *Type* het type van de variabele: *Geslacht*.
- Klik in het *Class Diagram* met de rechtermuisknop op de class *Werknemer*.
- Kies in het vervolgmenu de opdracht *Add* en vervolgens *Property*.
- Tik de naam van het Field in: *Geslacht*.

- In het propertiesvenster kies je bij Access voor *public*.
- Tik bij Type het type van de variabele: *Geslacht*.

Wijzig in de source van de class Werknemer de property *Geslacht* als volgt:

```
public Geslacht Geslacht
{
    get
    {
        return geslachtValue;
    }
    set
    {
        geslachtValue = value;
    }
}
```



Een Class Diagram kan meerdere classes bevatten. Reeds bestaande classes kunnen aan het Class Diagram toegevoegd worden door ze er vanuit de Solution Explorer naartoe te slepen.

	<p>De VS.NET ontwikkelomgeving helpt je bij het schrijven van code d.m.v. zogenaamde code snippets.</p> <p>Zo kan je bvb. op een snelle manier een property <i>Naam</i> aan een class toevoegen:</p> <ul style="list-style-type: none"> • positioneer de cursor in de code van de class waar je de property wil toevoegen en tik de shortcut <i>propfull</i> en druk vervolgens tweemaal op de Tab-toets of • klik met de rechtermuisknop in de code van de class waar je de property wil toevoegen en kies <i>Insert Snippet...</i> kies vervolgens <i>Visual C#</i> en verder <i>propfull</i> <p>Je ziet nu de volgende code:</p> <p></p> <pre>private int myVar; public int MyProperty { get { return myVar; } set { myVar = value; } }</pre> <ul style="list-style-type: none"> • tik het juiste type (<i>string</i>) van de private variabele en druk op de Tab-toets • tik de naam (<i>naamValue</i>) van de private variabele en druk op de Tab-toets • tik de naam (<i>Naam</i>) van de property en druk op de Tab-toets <p>Meer uitleg over code snippets vind je in de MSDN via de trefwoorden "<i>visual c# code snippets</i>".</p>
--	--

16.8 Properties met mixed access levels

Tot hertoe heb je properties altijd het access level *public* gegeven. Zoals hierboven beschreven kan je ook andere access levels toekennen aan een property. Dit access level geldt dan voor zowel de set- als de get-method van de property. Als je dat wenst kan je echter deze twee methods een verschillend access level geven.

Zo is het bijvoorbeeld mogelijk dat je iedereen toelaat een property *Salaris* te lezen (get). De set-routine mag echter alleen maar door een method uit de class zelf uitgevoerd worden. Dit schrijf je als volgt:

```
private decimal salarisValue;
public decimal Salaris
{
    get
    {
        return salarisValue;
    }
    private set
    {
        if (value >= 0m)
            salarisValue = value;
    }
}
```

Wens je de set-method ook in een afgeleide klasse te gebruiken, dan kan je deze method het access level *protected* geven.



Als je properties in een afgeleide klasse wil overriden (zie verder), dan moet je deze in de afgeleide klasse dezelfde access levels geven als in de moederklasse.

16.9 Reference variabelen declareren en objecten aanmaken

Eens de class gedefinieerd is, kan je in code variabelen declareren met als type deze class. Deze variabelen zijn **reference variabelen** of **object variabelen**. Met deze reference variabelen kan je naar objecten verwijzen.

Reference variabelen gedragen zich anders dan *value variabelen*.

Bij value variabelen vul je de waarde van de variabele rechtstreeks in in de variabele:

```
int aantalVrachtwagens;
aantalVrachtwagens=10;
```

10

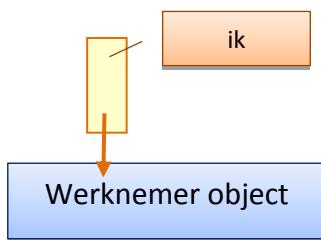
aantalVrachtwagens

Reference variabelen zijn echter slechts *verwijzingen* naar objecten.

Je maakt een object aan in het intern geheugen en laat de reference variabele naar het nieuwe object verwijzen:

```
Werknemer ik; (1)
ik=new Werknemer(); (2)
```

- (1) Je declareert een reference variabele *ik* die als type de class Werknemer heeft. Deze reference variabele verwijst **nog niet** naar een Werknemer object.
- (2) Je maakt een nieuw Werknemer object met het sleutelwoord **new**, gevuld door de naam van de class, gefolgd door ronde haakjes. De bedoeling van deze ronde haakjes wordt later duidelijk, bij de uitleg van constructors. Daarna verwijst je met de reference variabele *ik* naar het nieuwe object met de toekenningsoptreden (=).



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

```
ik.Naam = "Asterix";
```

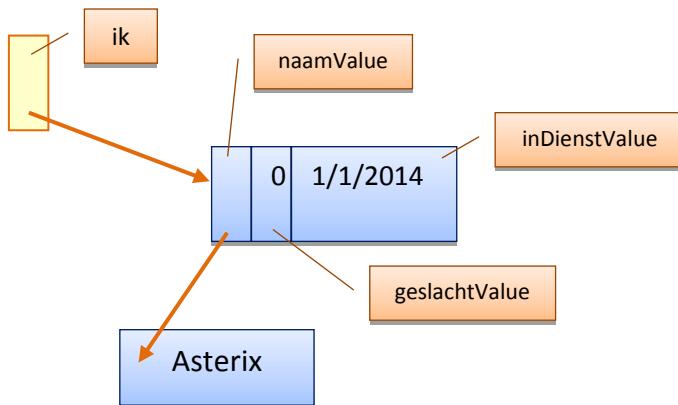
Gebruik een *Werknemer* object in de class Program:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik; (1)
            ik = new Werknemer(); (2)
            ik.Naam = "Asterix"; (3)
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Console.WriteLine(ik.Naam); (4)
            Console.WriteLine(ik.Geslacht);
            Console.WriteLine(ik.InDienst);
        }
    }
}
```

- (1) Je declareert een reference variabele van het type Werknemer. Op dit moment wijst deze reference variabele nog niet naar een Werknemer object.

- (2) Je maakt een nieuw Werknemer object en laat de reference variabele *ik* naar dit nieuwe object verwijzen.
- (3) Je vult de property *Naam* in. C# roept automatisch de **set** van deze property op, die enkel een niet-lege naam onthoudt in de variabele *naamValue*.
- (4) Je vraagt de property *Naam* op. C# roept automatisch de **get** routine van deze property op, die de waarde uit de variabele *naamValue* haalt.

In het geheugen gebeurt het volgende:



- De variabele *ik* verwijst naar het Werknemer object, waarin de **private** variabelen *naamValue*, *geslachtValue* en *inDienstValue* zijn opgenomen.
- De variabele *naamValue* is van het type **string** en is dus op zich ook een reference variabele naar de tekst *Asterix* in het geheugen.

16.10 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.
- Je kan testen of een reference variabele “nergens naar” verwijst, door de reference variabele te vergelijken met de waarde **null**.

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik; (1)
            ik = new Werknemer(); (2)
            if (ik == null) (3)
                Console.WriteLine("niet verbonden");
            else
                Console.WriteLine("verbonden");
        }
    }
}
  
```

```

        ik = null;                                (4)
        if (ik == null)                          (5)
            Console.WriteLine("niet verbonden");
        else
            Console.WriteLine("verbonden");
    }
}
}

```

- (1) Je declareert een lokale reference variabele van het type Werknemer. Een lokale reference variabele krijgt in C# niet automatisch een beginwaarde.
- (2) Je maakt een nieuw Werknemer object en verwijst met de reference variabele *ik* naar dit object.
- (3) Je test of de reference variabele "nergens naar" verwijst.
- (4) Je laat de reference variabele naar "nergens" (**null**) verwijzen.
- (5) Je test opnieuw of de reference variabele "nergens naar" verwijst.



Met de **?** operator wordt de source korter.

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            Console.WriteLine(ik == null ? "niet verbonden" : "verbonden");
            ik = null;
            Console.WriteLine(ik == null ? "niet verbonden" : "verbonden");
        }
    }
}

```

16.11 Declaratie en objectaanmaak in één opdracht

Je kan ook tegelijkertijd een reference variabele declareren en laten verwijzen naar een object:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();          (1)
            ik.Naam = "Asterix";
        }
    }
}

```

```
    ik.Geslacht = Geslacht.Man;
    ik.InDienst = new DateTime(2014, 1, 1);
    Console.WriteLine(ik.Naam);
}
}
}
```

- (1) Je declareert een reference variabele die je meteen laat verwijzen naar een nieuw *Werknemer* object.

16.12 Meerdere verwijzingen naar één object

Gezien een object variabele een verwijzing naar een object is, kan je ook met meerdere object variabelen naar hetzelfde object verwijzen.

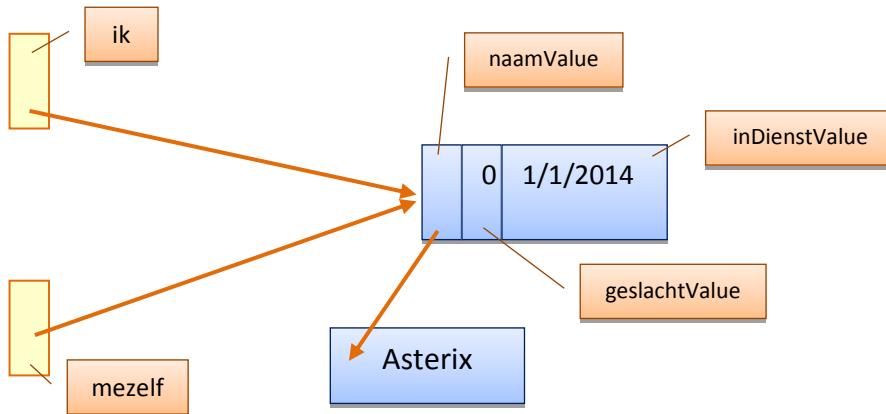
Je kan controleren of twee reference variabelen naar hetzelfde object verwijzen met de == vergelijkingsoperator:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Console.WriteLine(ik.Naam);
            Werknemer mezelf;
            mezelf = ik;                                (1)
            Console.WriteLine(mezelf.Naam);                (2)
            Console.WriteLine(ik == mezelf);              (3)
            ik = null;                                  (4)
            Console.WriteLine(ik == mezelf);              (5)
            Console.WriteLine(mezelf.Naam);                (6)
        }
    }
}
```

- (1) Je verwijst met de variabele mezelf naar hetzelfde *Werknemer* object als waar de variabele ik reeds naar verwijst.
(2) Gezien mezelf verwijst naar hetzelfde object als ik, zie je ook dezelfde naam.
(3) Beide object variabelen verwijzen naar hetzelfde object.
De == operator geeft true terug.
(4) De variabele ik verwijst niet meer naar het *Werknemer* object, maar de variabele mezelf verwijst wel nog naar het *Werknemer* object.

- (5) Beide object variabelen verwijzen niet meer naar hetzelfde object.
De == operator geeft false terug.
- (6) Gezien mezelf nog verwijst naar het oorspronkelijke Werknemer object, zie je ook nog de oorspronkelijke waarde van de property Naam.

Op het moment (1) verwijzen beide reference variabelen naar hetzelfde object:



16.13 Een array van references

Het type van een array kan ook een class (bvb. Werknemer) zijn, om een verzameling gelijkaardige objecten (bvb. werknemers) voor te stellen:

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

Ieder element van de array is slechts een reference, maar bevat zelf geen Werknemer object. De inhoud van ieder element is **null**, de references verwijzen nu nog niet naar Werknemer objecten.

Als je echt een verzameling Werknemer *objecten* nodig hebt, moet je per arrayelement een *Werknemer* object creëren en het arrayelement naar het object laten verwijzen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer[] onzeWerknemers = new Werknemer[20]; (1)
            for (int teller = 0; teller < onzeWerknemers.Length; teller++)
                onzeWerknemers[teller] = new Werknemer(); (2)
        }
    }
}
```

(1) Declaratie van een array met 20 Werknemer references.

- (2) De arrayelementen met `for` doorlopen.
- (3) Per arrayelement een Werknemer object aanmaken en het arrayelement naar het object laten verwijzen.

16.14 Garbage collection

Garbage collection is een techniek om het intern geheugen terug vrij te maken van objecten die niet meer in gebruik zijn.

C# voorziet automatische garbage collection: C# voert het opkuiswerk van het intern geheugen regelmatig uit, zonder dat de programmeur er zich zorgen om moet maken.

Een object leeft vanaf het moment dat je het aanmaakt (`ik=new Werknemer();`) totdat geen enkele reference variabele meer naar het object verwijst.

Een reference variabele verwijst niet meer naar een object als de reference variabele zelf uit het geheugen verdwijnt (bvb. de variabele `ik` verdwijnt op het einde van de routine `Main`, waarin de variabele gedeclareerd is).

Een reference variabele verwijst ook niet meer naar een object als je de reference variabele naar niets laat verwijzen (`ik=null;`).

C# doet de garbage collection niet onmiddellijk nadat het object niet meer gebruikt wordt. C# doet de garbage collection als er te weinig intern geheugen vrij is.

16.15 ReadOnly en WriteOnly properties

ReadOnly property:

- 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.

WriteOnly property:

- 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.

Maak in de class `Werknemer` een `ReadOnly` property `VerjaarAncien`.

Deze property geeft `true` terug als de werknemer vandaag x aantal jaren in de firma werkt. Anders geeft deze property `false` terug.

- Klik in het *Class Diagram* met de rechtermuisknop op de class `Werknemer`.

- Kies in het vervolghmenu de opdracht *Add* en vervolgens *Property*.
- Tik de naam van de property in: *VerjaarAncien*.
- In het propertiesvenster kies je bij *Access* voor *public*.
- Tik bij *Type* het type van de variabele: *bool*.

Wijzig in de source van de class *Werknemer* de property *VerjaarAncien* als volgt:

```
public bool VerjaarAncien (1)
{
    get (2)
    {
        return inDienstValue.Month == DateTime.Today.Month &&
            inDienstValue.Day == DateTime.Today.Day; (3)
    }
}
```

- (1) Je moet voor deze property geen **private** variabele voorzien. Je kan aan de hand van de **private** variabele *inDienstValue* nagaan of een werknemer vandaag X aantal jaren in dienst is.
- (2) In een *ReadOnly* property schrijf je enkel een **get**, geen **set**.
- (3) Een werknemer is vandaag X aantal jaren in dienst als de maand van zijn indiensttreding dezelfde is als de maand van de systeemdatum én de dag van zijn indiensttreding dezelfde is als de dag van de systeemdatum.

Je kan deze property uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Console.WriteLine(ik.VerjaarAncien);
        }
    }
}
```

17 Methods

17.1.1 Algemeen

Je definieert handelingen die een object kan uitvoeren als methods van de class.

Als eerste voorbeeld voeg je een method *Afbeeld()* toe aan de class Werknemer. Deze method beeldt de properties van een Werknemer object af op het scherm:

```
namespace CSharpPFCursus
{
    public class Werknemer
    {
        ...
        public void Afbeelden()          (1)
        {
            Console.WriteLine("Naam: {0}", Naam);
            Console.WriteLine("Geslacht: {0}", Geslacht);
            Console.WriteLine("In dienst: {0}", InDienst);      (2)
        }
    }
}
```

(1) Je begint de declaratie van een method met een sleutelwoord dat het toegangsniveau van de method aangeeft (**public, private, protected, internal, internal protected**).

Als je geen toegangsniveau opgeeft, past C# **private** toe.

Als de method geen resultaatwaarde teruggeeft, schrijf je daarna het sleutelwoord **void**.

Daarna volgt de naam van de method.

Microsoft stelt voor de naam van een method met een hoofdletter te beginnen. Als de naam van een method uit meerdere woorden bestaat, stelt Microsoft voor de eerste letter van die woorden terug met een hoofdletter te tikken.

Daarna volgen ronde haakjes. Tussen de ronde haakjes kan je parameters voor de method declareren. Dit zie je straks.

(2) De opdrachten van de method omsluit je met accolades.

(3) Dit is het einde van de method.

Je kan deze method toepassen voor elk Werknemer object:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
```

```

        ik.InDienst = new DateTime(2014, 1, 1);
        ik.Afbeelden();
    }
}
}

```

Het geheugengebruik van methods is anders dan dat van properties. Voor een property heeft ieder object zijn eigen geheugenruimte nodig: ieder object heeft een andere waarde voor de property.

Alle objecten van een class delen daarentegen de code van de methods van hun class. Het intern geheugen bevat dus slechts één keer de code van een method.

17.2 Methods met een parameter

Opdat we methods verder zouden kunnen uitleggen, voeg je een class *LijnenTrekker* toe aan de applicatie. Met *LijnenTrekker* objecten teken je lijnen op het scherm.

- 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 bij *Templates* voor *Class*.
- Tik bij *Name* de naam van de nieuwe Class: *LijnenTrekker*.
- Klik op de knop *Add*.

C# maakt een nieuwe source file (*LijnenTrekker.cs*) aan en plaatst de grove structuur van de class in dit bronbestand.

Je voegt aan deze class een method *TrekLijn()* toe. Bij de oproep van deze method geeft de programmeur een parameter mee, die aangeeft hoe lang (hoeveel tekens) de lijn moet zijn.

Je beschrijft het type en de naam van de parameter(s) tussen de ronde haakjes na de naam van de method:

```

using System;
namespace CSharpPFCursus
{
    public class LijnenTrekker
    {
        public void TrekLijn(int lengte) (1)
        {
            for (int teller = 0; teller < lengte; teller++) (2)
                Console.Write('-');
            Console.WriteLine();
        }
    }
}

```

- (1) Je begint de declaratie van een method met het sleutelwoord **public**:
de method is op te roepen vanuit ieder onderdeel van de applicatie.
De method geeft geen resultaatwaarde terug bij de oproep, dus je schrijft het sleutelwoord

`void.`

De naam van de method is *TrekLijn*.

Tussen ronde haakjes declareer je de parameter van de method. De parameter heeft als type `int` en heeft als naam *lengte*.

Microsoft raadt aan de namen van parameters te tikken zoals namen van variabelen: in kleine letters. Als een naam uit meerdere woorden bestaat, begin je de eerste letter van ieder woord na het eerste woord met een hoofdletter (bvb. aantalKinderen). Dit verhoogt de leesbaarheid.

- (2) Je tekent evenveel min-tekens op het scherm als aangeduid door de parameter.

Je kan deze method uitproberen in de method *Main* van de class *Program*:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Werknemer jij = new Werknemer();
            jij.Naam = "Obelix";
            jij.Geslacht = Geslacht.Man;
            jij.InDienst = new DateTime(2014, 1, 2);
            LijnenTrekker trekker = new LijnenTrekker(); (1)
            ik.Afbeelden();
            trekker.TrekLijn(30); (2)
            jij.Afbeelden();
            trekker.TrekLijn(79); (3)
        }
    }
}
```

- (1) Je maakt een nieuw *LijnenTrekker* object en verwijst ernaar met de reference variabele *trekker*.
(2) Je roept de method *TrekLijn()* op en stuurt daarbij de waarde 30 naar de parameter *lengte* van deze method.
(3) Je roept de method *TrekLijn()* op en stuurt deze keer de waarde 79 naar de parameter *lengte* van deze method.

17.3 Methods met meerdere parameters

Je past de method *TrekLijn()* van de class *LijnenTrekker* aan, de method heeft nu 2 parameters:

- De parameter *lengte* bepaalt het aantal te tekenen tekens.
- De parameter *teken* bepaalt het teken waarmee de lijn moet getekend worden.

Wijzig de method als volgt:

```
...
public void TrekLijn(int lengte, char teken)
{
    for (int teller = 0; teller < lengte; teller++)
        Console.Write(teken);
    Console.WriteLine();
}
...
```

Je kan de gewijzigde method uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Werknemer jij = new Werknemer();
            jij.Naam = "Obelix";
            jij.Geslacht = Geslacht.Man;
            jij.InDienst = new DateTime(2014, 1, 2);
            LijnenTrekker trekker = new LijnenTrekker();
            ik.Afbeelden();
            trekker.TrekLijn(30, '-'); (1)
            jij.Afbeelden();
            trekker.TrekLijn(79, '=');
        }
    }
}
```

(1) Je roept de method op met twee parameterwaarden.

C# vult de eerste waarde (30) in in de eerste parameter (*lengte*) en de tweede waarde ('-') in de tweede parameter (*teken*).

17.4 Methods creëren in het Class Diagram

Je kan ook het *Class Diagram* gebruiken om een method aan een class toe te voegen, in plaats van de method in de source zelf in te tikken.

Voeg eerst de class *LijnenTrekker* toe aan het Class Diagram door de class vanuit de Solution Explorer naar het Class Diagram te slepen. Klik vervolgens op de knop  om de bijhorende method te zien.

Verwijder de method *TrekLijn()* handmatig uit de code van de class *LijnenTrekker*. De method is nu ook uit het Class Diagram verdwenen.

Nu maak je de method opnieuw aan, maar in het Class Diagram.

- Klik in het *Class Diagram* met de rechtermuisknop op de class *LijnenTrekker*.
- Kies in het popup menu de opdracht *Add* en vervolgens *Method*.
- Tik de naam van de method in: *TrekLijn*.

De parameters voeg je toe in het venster *Class Details*. Indien dit venster niet zichtbaar is, kies je het menu *VIEW - Other Windows - Class Details*.

- Klik op het *pijltje* voor de method *Treklijn*.
- Klik op *<add parameter>*.
- Tik de naam van de parameter: *lengte* en druk op *Enter*.
- Vervang het type *string* door *int*.
- Klik opnieuw op *<add parameter>*.
- Tik de naam van de parameter: *teken* en druk op *Enter*.
- Vervang het type *string* door *char*.

De method is nu ook toegevoegd in de source. Het throw-statement mag je opnieuw verwijderen.

De code van de method moet je nog zelf intikken in de source (*LijnenTrekker.cs*):

```
public void TrekLijn(int lengte, char teken)
{
    for (int teller = 0; teller < lengte; teller++)
        Console.Write(teken);
    Console.WriteLine();
}
```

17.5 Overloading

Overloading betekent dat je van een method meerdere versies maakt.

C# aanvaardt overloading als:

- een versie van een method een verschillend aantal parameters heeft van de andere versies van deze method.
- een versie van een method een gelijk aantal parameters heeft als een andere versie van deze method, maar één of meerdere types van deze parameters verschillend zijn.

Maak van de method *TrekLijn()* in de class *LijnenTrekker* 3 versies:

- Eén versie (die je al hebt) met twee parameters: de lengte van de lijn en het teken, dat gebruikt wordt om de lijn te trekken.

- Een tweede versie met één parameter: de lengte van de lijn. Deze versie gebruikt het min-teken om de lijn te trekken.
- Een derde versie zonder parameters. Deze versie trekt een lijn van 79 min-tekens.

```
using System;
namespace CSharpPFCursus
{
    public class LijnenTrekker
    {
        public void TrekLijn(int lengte, char teken) (1)
        {
            for (int teller = 0; teller < lengte; teller++)
                Console.Write(teken);
            Console.WriteLine();
        }

        public void TrekLijn(int lengte) (2)
        {
            TrekLijn(lengte, '-'); (3)
        }

        public void TrekLijn() (4)
        {
            TrekLijn(79); (5)
        }
    }
}
```

- (1) De eerste versie van de method (deze versie had je al).
- (2) De tweede versie van de method.
- (3) Binnen de tweede versie roep je de eerste versie op (je recycleert kennis).
- (4) De derde versie van de method.
- (5) Binnen de derde versie roep je de tweede versie op (je recycleert kennis).

Je kan nu de verschillende versies van de method `TrekLijn()` uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Naam = "Asterix";
            ik.Geslacht = Geslacht.Man;
            ik.InDienst = new DateTime(2014, 1, 1);
            Werknemer jij = new Werknemer();
```

```
jij.Naam = "Obelix";
jij.Geslacht = Geslacht.Man;
jij.InDienst = new DateTime(2014, 1, 2);
LijnenTrekker trekker = new LijnenTrekker();
ik.Afbeelden();
trekker.TrekLijn(30, '-');
jij.Afbeelden();
trekker.TrekLijn(79, '=');
trekker.TrekLijn();
trekker.TrekLijn(10);
}
}
```

```
}
```

```
}
```

```
}
```



Merk op dat bij het intypen van `trekker.TrekLijn()` een popup-lijstje verschijnt, waarin je een keuze kan maken uit de drie versies van de method `TrekLijn()`.

`trekker.TrekLijn()`

▲ 1 of 3 ▼ void LijnenTrekker.TrekLijn()

17.6 Named parameters

Bij de oproep van een method geef je de parameters mee *in de volgorde* zoals ze gedefinieerd zijn bij de declaratie van de method.

In de class `LijnenTrekker` heb je bvb. een method `TrekLijn()` als volgt gedefinieerd:

```
public class LijnenTrekker
{
    public void TrekLijn(int lengte, char teken)
    {
        ...
    }
    ...
}
```

Deze method roep je als volgt op:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            LijnenTrekker trekker = new LijnenTrekker();
            trekker.TrekLijn(10, '*');
        }
    }
}
```

waarbij de waarde *10* naar de parameter *lengte* en de waarde *'*'* naar de parameter *teken* doorgegeven wordt.

Je kan bij de *oproep* van een method met parameters ook de *parameternaam* vermelden. Dit zijn zogenaamde “*benoemde parameters*” of “*named parameters*”.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            LijnenTrekker trekker = new LijnenTrekker();
            trekker.TrekLijn(teken: '*', lengte: 10);          (1)
        }
    }
}
```

- (1) Je vermeldt de parameternaam, gevolgd door een dubbele punt, gevolgd door de mee te geven parameterwaarde. Aan de hand van de parameternaam wordt de juiste waarde aan de juiste parameter doorgegeven.

Je hoeft dan geen rekening te houden met de volgorde van de parameters in de definitie van de method.

17.7 Optionele parameters

Je kan bij een method zogenaamde *optionele parameters* definiëren. Hiermee geef je aan dat de programmeur bij de oproep van de method niet verplicht is om voor deze parameters een waarde mee te geven. Je moet een optionele parameter wel een beginwaarde geven. Als een programmeur bij de oproep van de method geen waarde meegeeft aan de parameter, krijgt de parameter deze beginwaarde.

Voorbeeld: maak in de method *TrekLijn(int lengte, char teken)* van de class *LijnenTrekker* van de parameter *teken* een optionele parameter:

```
using System;
namespace CSharpPFCursus
{
    public class LijnenTrekker
    {
        public void TrekLijn(int lengte, char teken = '-')
        {
            for (int i = 0; i < lengte; i++)
            {
                Console.Write(teken);
            }
        }
    }
}
```

(1)

```
        Console.WriteLine();  
    }  
}
```

- (1) Je definieert een parameter als optioneel door er een beginwaarde aan toe te kennen:

```
char teken = '-'
```

De parameter *lengte* is hier niet optioneel, de parameter *teken* wel.

Je kan nu de method *TrekLijn()* op meerdere manieren oproepen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            LijnenTrekker trekker = new LijnenTrekker();
            trekker.TrekLijn(10, '*'); (1)
            trekker.TrekLijn(10); (2)
            trekker.TrekLijn(lengte: 30);
            trekker.TrekLijn(lengte: 40, teken: '='); (3)

        }
    }
}
```

- (1) Je roept de method TrekLijn() op met beide parameters ingevuld.
 - (2) Je roept de method TrekLijn() op met enkel de eerste parameter lengte ingevuld. De tweede parameter krijgt zijn initiële waarde '-'.
 - (3) Je gebruikt named parameters bij de oproep van de method TrekLijn().
Dit is vooral interessant als de method veel optionele parameters bevat: je moet dan bij de oproep de verplichte parameters vermelden + enkel de optionele parameters waaraan je zelf een waarde wil meegeven.



- Bij de *declaratie* van de method moeten eerst alle niet-optionele parameters vermeld worden, daarna alle optionele parameters.
 - Door optionele parameters te gebruiken kan je vermijden dat je meerdere versies van eenzelfde method moet schrijven (zie overloading). Door de method *TrekLijn()* van een optionele parameter te voorzien, is de overloaded method *TrekLijn(int lengte)* overbodig geworden.
Je moet dus minder code schrijven waardoor deze eenvoudiger te onderhouden en beter leesbaar wordt.
 - Bij optionele parameters kunnen de sleutelwoorden *ref* of *out* (zie verder) niet voorkomen.

17.8 Parameters met het sleutelwoord ref

Standaard roept C# methods op "by value".

Dit betekent dat C# kopieën maakt van de waarden die je naar de parameters stuurt bij de oproep van een method. Als je in de method de waarde van de parameters wijzigt, wijzig je de originele waarden niet, maar wijzig je enkel de kopieën.

Als je de declaratie van een parameter in een method laat voorafgaan door het sleutelwoord **ref**, bevat deze parameter geen kopie van de inhoud van de variabele die je bij de oproep van de method meegeeft aan deze parameter.

Deze parameter bevat daarentegen een **verwijzing** naar deze originele variabele. Als je in de method de waarde van deze parameter wijzigt, wijzig je dus ook de waarde van de originele variabele.

Bij de oproep van de method moet je vóór de variabele die je naar de parameter stuurt, ook het sleutelwoord **ref** gebruiken.

Voeg aan het project een class *Verwisselaar* (source file *Verwisselaar.cs*) toe om dit uit te proberen.
Voeg aan deze class een method *Verwissel()* toe, die de inhoud van twee parameters verwisselt.

De eerste versie zal niet werken, want bij de parameters ontbreekt het sleutelwoord **ref**:

```
using System;
namespace CSharpPFCursus
{
    public class Verwisselaar
    {
        public void Verwissel(int getal1, int getal2)
        {
            int tussen = getal1;
            getal1 = getal2;
            getal2 = tussen;
        }
    }
}
```

Het programma om deze method uit te testen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            int eerste = 10, tweede = 20;
            Verwisselaar verwisselaar = new Verwisselaar();
            verwisselaar.Verwissel(eerste, tweede);
            Console.WriteLine(eerste);
            Console.WriteLine(tweede);
        }
    }
}
```

Als je het programma uitvoert, is de inhoud van de variabelen *eerste* en *tweede* niet verwisseld. Dit komt omdat kopieën van de variabelen *eerste* en *tweede* naar de parameters *getal1* en *getal2* gestuurd werden. De method *Verwissel()* heeft enkel deze kopieën verwisseld, niet de originele variabelen (*eerste* en *tweede*).

Opdat de method wel degelijk de inhoud van de variabelen zou verwisselen, voorzie je de parameters van het sleutelwoord **ref**:

```
using System;
namespace CSharpPFCursus
{
    public class Verwisselaar
    {
        public void Verwissel(ref int getal1, ref int getal2)
        {
            int tussen = getal1;
            getal1 = getal2;
            getal2 = tussen;
        }
    }
}
```

Bij het oproepen van de method voorzie je de variabelen die je naar de parameters stuurt, ook van het sleutelwoord **ref**:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            int eerste = 10, tweede = 20;
            Verwisselaar verwisselaar = new Verwisselaar();
            verwisselaar.Verwissel(ref eerste, ref tweede);
            Console.WriteLine(eerste);
            Console.WriteLine(tweede);
        }
    }
}
```

Als je het programma nu uitvoert, is de inhoud van de variabelen *eerste* en *tweede* wel verwisseld. Dit komt omdat referenties van de variabelen *eerste* en *tweede* naar de parameters *getal1* en *getal2* gestuurd werden. De method *Verwissel()* verwisselt via deze referenties de inhoud van de originele variabelen (*eerste* en *tweede*).

17.9 Parameters met het sleutelwoord out

Een parameterdeclaratie met het sleutelwoord **out** heeft ongeveer dezelfde betekenis als een parameterdeclaratie met het sleutelwoord **ref**: de parameter bevat geen kopie van de inhoud van de variabele die je bij de oproep van de method meegeeft aan deze parameter, maar een verwijzing

naar deze originele variabele. Als je in de method deze parameter wijzigt, wijzig je de originele variabele. Bij de oproep van de method moet je voor de variabele die je naar de parameter stuurt, ook het sleutelwoord **out** gebruiken.

Het verschil met het sleutelwoord **ref** is dat je bij het sleutelwoord **out** de variabele die je naar de parameter stuurt geen beginwaarde moet geven (wat je wel moet doen bij **ref**).

Om dit uit te proberen voeg je aan de class *Verwisselaar* een method *VerwisselNaarAndereVariabelen()* toe. Deze method bevat twee gewone parameters waarmee de method twee getallen binnenkrijgt. De method stuurt de verwisselde waarden naar twee **out** parameters:

```
public void VerwisselNaarAndereVariabelen(int getal1, int getal2,
                                             out int verwisseld1, out int verwisseld2)
{
    verwisseld1 = getal2;
    verwisseld2 = getal1;
}
```

Het programma om deze method uit te proberen stuurt de variabelen *resultaat1* en *resultaat2* met het sleutelwoord **out** naar de method. Bemerk dat je deze variabelen (*resultaat1*, *resultaat2*) geen beginwaarde geeft:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            int eerste = 10, tweede = 20;
            int resultaat1, resultaat2;
            Verwisselaar verwisselaar = new Verwisselaar();
            verwisselaar.VerwisselNaarAndereVariabelen(eerste, tweede,
                out resultaat1, out resultaat2);
            Console.WriteLine(resultaat1);
            Console.WriteLine(resultaat2);
        }
    }
}
```

De *TryParse()*method (zie verder) maakt gebruik van **out** parameters.

17.10 Methods met een resultaatwaarde

De methods die je tot nu geschreven hebt, zijn methods zonder resultaatwaarde. Je hebt dit aangegeven door bij de declaratie van de method de naam van de method te laten voorafgaan door het sleutelwoord **void**.

Je kan ook een method definiëren die een resultaatwaarde teruggeeft. De method gedraagt zich dan als een functie: als je de method opropt, krijg je na de oproep de resultaatwaarde van de functie ter beschikking.

Een method kan maximaal één resultaatwaarde hebben. Deze resultaatwaarde kan een Value type zijn ([int](#), [char](#), [bool](#), [DateTime](#), ...) of een reference naar een object ([string](#), [Werknemer](#), ...).

Je geeft aan dat een method een resultaatwaarde teruggeeft door bij de declaratie van de method de naam van de method te laten voorafgaan door het gegevenstype van de waarde die de method teruggeeft.

De resultaatwaarde van de method geef je met het sleutelwoord [return](#) terug aan de code die de method heeft opgeroepen.

Om methods met een resultaatwaarde verder uit te leggen, voeg je aan de applicatie een class *Omzetter* (*Omzetter.cs*) toe.

Met *Omzetter* objecten doe je omzettingen van cm naar inch, van inch naar cm.

Je voegt aan deze class twee methods toe:

- Een method *CmNaarInch()*.
Deze method krijgt als parameter een afstand in cm binnen en geeft deze afstand in inches terug.
- Een method *InchNaarCm()*.
Deze method krijgt als parameter een afstand in inches binnen en geeft deze afstand in cm terug.

```
using System;
namespace CSharpPFCursus
{
    public class Omzetter
    {
        public const double CentimetersPerInch = 2.54d; (1)
        public double CmNaarInch(double cm) (2)
        {
            return cm / CentimetersPerInch; (3)
        }
        public double InchNaarCm(double inch)
        {
            return inch * CentimetersPerInch;
        }
    }
}
```

(1) Een constante met de verhouding tussen centimeters en inches.

(2) Bij de declaratie van de method laat je de naam van de method voorafgaan door het gegevenstype van de resultaatwaarde: [double](#). Verder krijgt de method een parameter van het type [double](#) binnen. Het is toeval dat in dit voorbeeld het gegevenstype van de resultaatwaarde hetzelfde is als dat van de parameter.

- (3) Je geeft de omgezette waarde terug naar de code die de method heeft opgeroepen met het sleutelwoord **return**.

Je kan deze methods uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Omzetter omzetter = new Omzetter();
            Console.Write("Afstand in cm:");
            double cm = double.Parse(Console.ReadLine());
            Console.WriteLine(omzetter.CmNaarInch(cm)); (1)
            Console.WriteLine("inches");
            LijnenTrekker lijnenTrekker = new LijnenTrekker();
            lijnenTrekker.TrekLijn();
            Console.Write("Afstand in inches: ");
            double inches = double.Parse(Console.ReadLine());
            Console.WriteLine(omzetter.InchNaarCm(inches));
            Console.WriteLine("cm");
        }
    }
}
```

- (1) Je roept de method op en geeft als parameter een afstand in cm mee. De resultaatwaarde van de method toon je aan de gebruiker.

17.11 Een toepassing: TryParse()

Bij wijze van toepassing op out-parameters en methods met een resultaatwaarde bekijken we even de method *TryParse()*.

Zoals aangegeven in paragraaf 9.7.2 heeft elk Value type een method *Parse()*, waarmee je een string kan omzetten naar een Value type.

TryParse() is een meer voorzichtige method in die zin dat er een poging wordt gedaan om de string om te zetten naar een Value type. Indien dit lukt, dan is het resultaat van de method *true*. Lukt het niet, dan is het resultaat *false*.

TryParse() heeft twee parameters: een string waarin de te converteren waarde staat, en een out-parameter waarin de omgezette waarde (indien de omzetting lukt) terecht komt.

Een voorbeeld:

```
using System;
namespace CSharpPFCursus
{
```

```
class Program
{
    public static void Main(string[] args)
    {
        Omzetter omzetter = new Omzetter();
        Console.Write("Afstand in cm:");
        double cm;
        if (double.TryParse(Console.ReadLine(), out cm)) (1)
        {
            Console.WriteLine(omzetter.CmNaarInch(cm));
            Console.WriteLine(" inches");
        }
        else
            Console.WriteLine("Geen correct getal");
    }
}
```

- (1) We proberen invoer van het scherm (een string) om te zetten naar een *double*. Indien de omzetting lukt, komt het resultaat in de variabele *cm* terecht. Het resultaat van de method *TryParse()* (een bool) bepaalt of de omzetting wordt uitgevoerd, dan wel een foutbericht wordt afgebeeld.

18 Constructors

18.1 Algemeen

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 tijdelijk gegevens bewaart
- ...

C# roept de constructor automatisch op als je een nieuw object aanmaakt:

```
Werknemer ik = new Werknemer();
```

Je maakt een constructor in een class door in de class een *method* op te nemen met *dezelfde naam als de class*:

```
public class Werknemer
{
    public Werknemer()
    {
    }

    ...
}
```

Je voorziet een constructor van een sleutelwoord dat het toegangsniveau van deze constructor aangeeft (**public** wil bvb. zeggen dat je de constructor vanuit alle code van de applicatie kan oproepen).

Bemerk dat je voor de constructor het sleutelwoord **void** niet schrijft.



Je kan snel een constructor aan een class toevoegen met de code snippet **ctor Tab Tab**.

Standaard maakt Visual Studio geen constructor aan voor een class die je aanmaakt. De C# compiler maakt zelf een constructor aan, die geen code bevat.

Omdat een constructor meestal dient om class variabelen te initialiseren, bekijken we eerst de beginwaarden van class variabelen als je geen constructor voorziet.

In de class *Werknemer* zijn de private variabelen *naamValue*, *inDienstValue* en *geslachtValue* class variabelen.

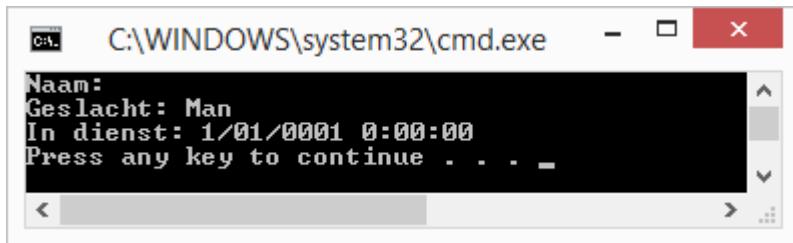
Class variabelen krijgen in C# een beginwaarde volgens hun type:

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

Je kan dit uittesten met het volgende programma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            ik.Afbeelden();
        }
    }
}
```

Op de console zie je het volgende:



Bij `Naam`: zie je niets. De `WriteLine()` method van de class `Console` toont voor een reference variabele die `null` bevat (`naamValue`) niets.

Zonder constructor krijgen properties soms onrealistische waarden (kan bvb. in een hedendaagse firma een werknemer op 1/1/1 in dienst komen?).

Zonder constructor kunnen sommige variabelen ook een beginwaarde krijgen die snel tot fouten leiden. Dit zie je met het volgende programma:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();
            Console.WriteLine(ik.Naam.ToUpper()); (1)
        }
    }
}

```

- (1) Als je dit programma uitvoert, breekt het programma op dit punt af met een fout van het type *NullReferenceException*. De reden is dat de property *Naam* verwijst naar de reference variabele *naamValue* die als beginwaarde **null** gekregen heeft. Op een **null** waarde kan de method *ToUpper()* niet uitgevoerd worden.

18.2 Default constructor

Een default constructor is een constructor die **geen parameter(s)** heeft.

Je voegt een default constructor toe met de volgende code:

```

using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        public Werknemer()
        {
            Naam = "Onbekend"; (1)
            InDienst = DateTime.Today; (2)
        }
        ...
    }
}

```

- (1) Je vult de property *Naam* met de tekst *Onbekend*.
Bemerk dat er niet staat: *naamValue="Onbekend"*; Bij het uitvoeren van *Naam="Onbekend"*; voert C# de **set** van de *Naam* property uit. Deze **set** controleert of we geen lege naam invullen. Het is een goede gewoonte om deze controle altijd te laten uitvoeren.
- (2) Je vult de property *InDienst* in met de systeemdatum.

Als je nu een *Werknemer* object aanmaakt, bevat dit *Werknemer* object reeds meer logische beginwaarden:

```

using System;
namespace CSharpPFCursus

```

```
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            Werknemer ik = new Werknemer();  
            ik.Afbeelden();  
            Console.WriteLine(ik.Naam.ToUpper());  
        }  
    }  
}
```

(1) Het programma breekt niet meer af met een *NullReferenceException*.

18.3 Constructor(s) met parameters

Constructors met parameters laten een programmeur toe bij de aanmaak van een object onmiddellijk beginwaarden voor de object properties mee te geven. Deze beginwaarden komen als parameters van de constructor binnen.

Maak een extra constructor waarmee een programmeur bij de creatie van een Werknemer object reeds *Naam*, *InDienst* en *Geslacht* kan meegeven:

```
Werknemer ik=new Werknemer("Asterix",new DateTime(2014,1,1),Geslacht.Man);
```

Je kan meerdere versies van een constructor maken, zoals je van andere methods meerdere versies kan maken. Dit heb je bij het hoofdstuk METHODS gezien. Deze techniek heet *overloading*.

De tweede versie van de constructor krijgt de beginwaarden voor de properties *Naam*, *InDienst* en *Geslacht* binnen als parameters:

```
using System;  
namespace CSharpPFCursus  
{  
    public class Werknemer  
    {  
        public Werknemer()  
        {  
            Naam = "Onbekend";  
            InDienst = DateTime.Today;  
        }  
  
        public Werknemer(string naam, DateTime inDienst, Geslacht geslacht)  
        {  
            Naam = naam;  
            InDienst = inDienst;  
            Geslacht = geslacht;  
        }  
        ...  
    }  
}
```

- (1) Je neemt de parameterwaarde (*naam*) en stopt deze in de property (*Naam*). C# voert dan de **set** van de property uit. Deze **set** controleert of de waarde niet leeg is. Daarna wordt deze waarde in de **private** variabele *naamValue* ingevuld.

Je kan deze constructor nu gebruiken om een Werknemer object aan te maken:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer("Asterix",
                new DateTime(2014, 1, 1), Geslacht.Man); (1)
            ik.Afbeelden();
            Werknemer jij = new Werknemer(); (2)
            jij.Afbeelden();
        }
    }
}
```

- (1) Je geeft parameters mee bij de aanmaak van het object.
 C# roept de *constructor met parameters* op.
 Deze brengt de parameterwaarden over naar de properties.
- (2) Je geeft geen parameters mee bij de aanmaak van het object.
 C# roept de *default constructor* op.

18.4 this

Het sleutelwoord **this** staat voor een reference naar het *huidige object*. Als je binnen class code **this** gebruikt, verwijst 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.

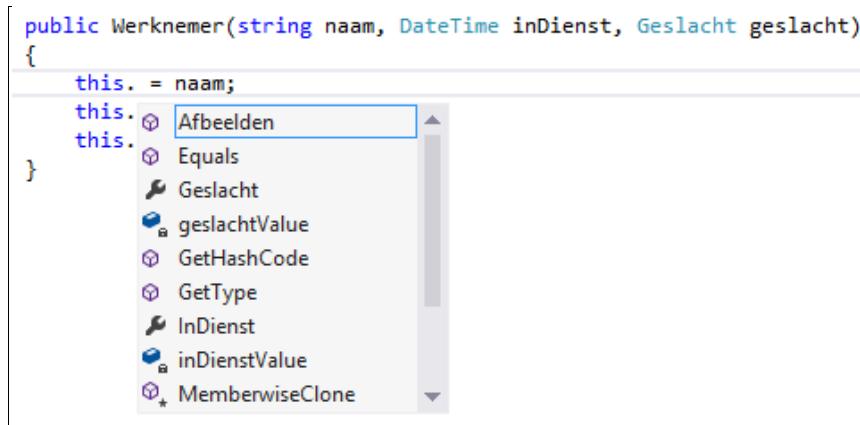
Je kan de constructors dus als volgt herschrijven:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        public Werknemer()
        {
            this.Naam = "Onbekend";
            this.InDienst = DateTime.Today;
        }

        public Werknemer(string naam, DateTime inDienst, Geslacht geslacht)
        {
            this.Naam = naam;
```

```
    this.InDienst = inDienst;
    this.Geslacht = geslacht;
}
...
}
```

Deze manier van werken is handig in VS.NET, omdat na het tikken van **this.** een popup-lijstje met alle properties, methods en variabelen van de class verschijnt, waaruit de programmeur een keuze kan maken:



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.

Als je de parameters van de constructor met hoofdletters zou schrijven (wat ingaat tegen de naamgevingvoorstellingen van Microsoft !), hebben deze parameters dezelfde naam als de properties waarnaar je de parameters doorstuurt.

Als je deze constructor als volgt zou schrijven, stop je de parameters nog eens in zichzelf:

```
public Werknemer(string Naam, DateTime InDienst, Geslacht Geslacht)
{
    Naam = Naam;
    InDienst = InDienst;
    Geslacht = Geslacht;
}
```

(1)

- (1) De verwijzing *Naam* wordt hier beschouwd als een verwijzing naar de parameter *Naam* van de constructor. Je stopt dus de parameter in zichzelf, in plaats van in de property *Naam*.

Om het onderscheid te maken, moet je deze constructor als volgt schrijven:

```
public Werknemer(string Naam, DateTime InDienst, Geslacht Geslacht)
{
    this.Naam = Naam;
    this.InDienst = InDienst;
    this.Geslacht = Geslacht;
```

(1)

}

- (1) De verwijzing *Naam* is een verwijzing naar de parameter *Naam* van de constructor. De verwijzing *this.Naam* is een verwijzing naar de property *Naam* van de class. Nu breng je dus de parameter over naar de property, zoals het hoort.

18.5 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(...)*.

Tussen de ronde haakjes tik je de parameters van de op te roepen constructor (indien nodig).

Deze oproep moet je schrijven vóór de open accolade van de huidige constructor.

Je past deze techniek toe in de class *Werknemer*. Je roept in de *default constructor* de constructor met parameters op:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        public Werknemer(): this("Onbekend", DateTime.Today, Geslacht.Man)           (1)
        {

        }

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

        ...
    }
}
```

- (1) Je roept de constructor met parameters op.

Als *Naam* geef je *Onbekend* mee, voor *InDienst* de systeemdatum en als *Geslacht Man*.



Constructor chaining is een techniek die bij *Inheritance* (zie verder) toegepast wordt.

18.6 Destructor

Een destructor is een method van de class die C# automatisch oproeft 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 opgekuisht wordt.

18.7 Classes zonder default constructor

Een default constructor laat toe een "anoniem" object te maken.

Met de default constructor van Werknemer maak je een "anonieme" werknemer: Onbekend als naam, een man, datumindienst op systeemdatum. De vraag is of deze class een afspiegeling is van de werkelijkheid.

Je kan dit verhinderen door in je class geen default constructor te schrijven, maar enkel één of meerdere constructors mét parameters. Zodra je class een constructor bevat, maakt de compiler geen default constructor meer aan.

Je kan dit uitproberen door de default constructor in de Werknemer class in commentaar te plaatsen en de andere constructor te laten staan.

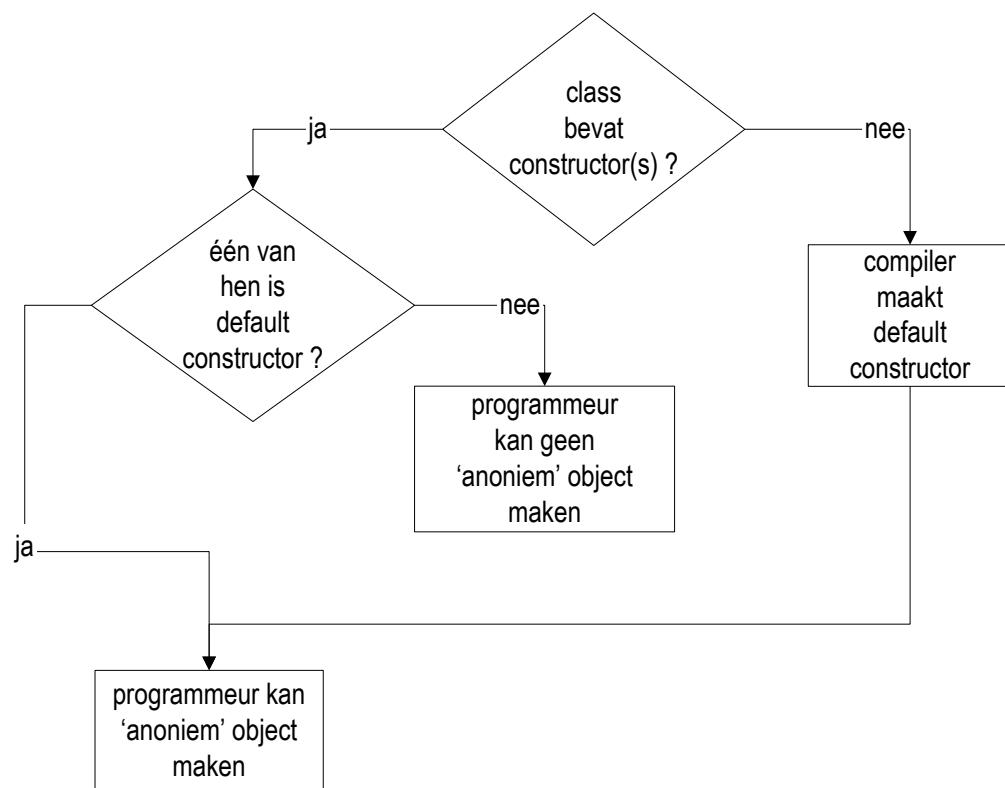
```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        /*
        public Werknemer(): this("Onbekend", DateTime.Today, Geslacht.Man)
        {
        }
        */
        
        public Werknemer(string naam, DateTime inDienst, Geslacht geslacht)
        {
            this.Naam = naam;
            this.InDienst = inDienst;
            this.Geslacht = geslacht;
        }
        ...
    }
}
```

Nu kan je geen "anonieme" werknemer meer creëren:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer("Asterix",
                new DateTime(2014, 1, 1), Geslacht.Man);
            ik.Afbeelden();
            Werknemer jij = new Werknemer(); ——————
            jij.Afbeelden();
        }
    }
}
```



18.8 Overzicht van constructors



oefeningen: Classes en objects

19 Static members

19.1 Static variabelen, properties en methods

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 op 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.

De class Werknemer:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        ...
        private static DateTime personeelsfeestValue; (1)
        public static DateTime Personeelsfeest (2)
        {
            set
            {
                personeelsfeestValue = value;
            }
            get
            {
                return personeelsfeestValue;
            }
        }
    }
}
```

```

public void Afbeelden()
{
    Console.WriteLine("Naam: {0}", Naam);
    Console.WriteLine("Geslacht: {0}", Geslacht);
    Console.WriteLine("In dienst: {0}", InDienst);
    Console.WriteLine("Personeelsfeest: {0}", Personeelsfeest);      (3)
}
...

```

- (1) Alle Werknemer objecten delen de variabele *personeelsfeestValue*.
- (2) Bij de property tik je ook het woord **static**. Zo geef je aan dat je deze property (bvb. in de *Main()*) oproept via de class Werknemer zelf:
Werknemer.Personeelsfeest=DateTime.Today;
- (3) Als je een werknemer afbeeldt, beeld je ook de datum van het personeelsfeest af.

Je kan deze nieuwe property uitproberen:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer("Asterix", DateTime.Today, Geslacht.Man);
            Werknemer jij = new Werknemer("Obelix", DateTime.Today, Geslacht.Man);
            Werknemer.Personeelsfeest = new DateTime(2014, 12, 12);          (1)
            Console.WriteLine(Werknemer.Personeelsfeest);                      (2)
            ik.Afbeelden();                                              (3)
            jij.Afbeelden();
        }
    }
}

```

- (1) Je vult de **static** property *Personeelsfeest* in via de *Werknemer* class. Gezien de **private** variabele *personeelsfeestValue* die bij deze property hoort ook **static** is, delen alle Werknemer objecten deze variabele.
- (2) Je toont de property *Personeelsfeest* via de *Werknemer* class.
- (3) Je ziet bij het afbeelden van een Werknemer object de property *Personeelsfeest*.

19.2 Static constructor

De **static** variabelen van een class krijgen een beginwaarde volgens hun type. De **static** variabele *personeelsfeestValue* krijgt de beginwaarde 1/1/1.

Met een **static** constructor kan je **static** variabelen een andere beginwaarde geven. Je schrijft een **static** constructor als een default constructor voorafgegaan door het woord **static**. Je vermeldt geen **public**.

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

Voorbeeld: Het personeelsfeest valt op de eerste vrijdag van februari:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        ...
        static Werknemer()
        {
            Personeelsfeest = new DateTime(DateTime.Today.Year, 2, 1); (1)
            while (Personeelsfeest.DayOfWeek != DayOfWeek.Friday) (2)
                Personeelsfeest = Personeelsfeest.AddDays(1); (3)
        }
        ...
    }
}
```

- (1) De **static** constructor.
- (2) Je geeft de property *Personeelsfeest* als datum de eerste februari van het huidige jaar.
- (3) Zolang de dag van de week van *Personeelsfeest* geen vrijdag is,
- (4) verhoog je de datum *Personeelsfeest* met één dag.

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

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(Werknemer.Personeelsfeest);
        }
    }
}
```

19.3 Static classes

Je kan een class zelf ook static maken. Je doet dit door het sleutelwoord **static** voor het sleutelwoord **class** te plaatsen bij de definitie van de class.

Een static class kan **enkel static methods** bevatten. Om de static methods van een static class op te roepen hoef je niet eerst een instance van de class te creëren. Meer nog: je kan zelfs geen instance van een static class aanmaken.

In het onderstaande voorbeeld roep je de static method *Kwadraat()* van de static class *Rekenaar* op:

```
using System;
namespace CSharpPFCursus
{
    public static class Rekenaar
    {
```

```
    public static int Kwadraat(int getal) { return getal * getal; }
}
}

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(Rekenaar.Kwadraat(3));
        }
    }
}
```

20 Inheritance

20.1 Algemeen

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:

- aan de nieuwe class properties en methods toevoegen.
- 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 sommige programmeertalen (bvb. C++) kan een derived class meerdere base classes hebben. Dit heet *multiple inheritance*. In die taal kan bvb. een derived class Dolfijn als base class Zoogdier én Vis hebben.

Bij de declaratie van de derived class vermeld je *na de naam van de class een dubbele punt, gevolgd door de naam van de base class* (de class waarvan je erft).

Voorbeeld: maak een nieuwe class *Arbeider* die van de class *Werknemer* erft.

Om compilerfouten te vermijden zorg je er eerst voor dat de base class *Werknemer* terug een *default constructor* heeft, door de commentaar te verwijderen bij de code van de default constructor. Het samenspel tussen inheritance en constructors komt later in dit hoofdstuk aan bod.

Voeg de nieuwe class *Arbeider* (*Arbeider.cs*) toe aan het project *CSharpPFCursus*.

Voeg aan deze class 2 properties toe: *Uurloon* en *Ploegenstelsel*.

```
using System;
namespace CSharpPFCursus
{
    public class Arbeider : Werknemer (1)
    {
        private decimal uurloonValue;
        public decimal Uurloon
        {
            get
            {
                return uurloonValue;
            }
            set
        }
    }
}
```

```

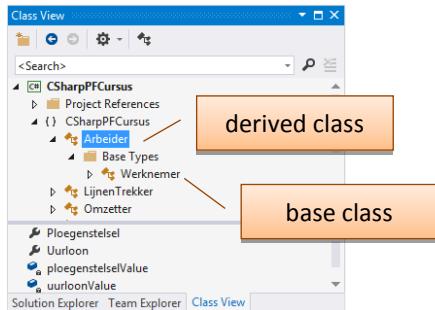
    {
        if (value >= 0m) (2)
            uurloonValue = value;
    }
}

private byte ploegenstelselValue;
public byte Ploegenstelsel
{
    get
    {
        return ploegenstelselValue;
    }
    set
    {
        if (value >= 1 && value <= 3) (3)
            ploegenstelselValue = value;
    }
}
}

```

- (1) De class *Arbeider* erft van de class *Werknemer*.
- (2) De **set** van de property *Uurloon* aanvaardt enkel correcte (positieve) waarden.
- (3) De **set** van de property *Ploegenstelsel* aanvaardt enkel correcte waarden (1, 2 of 3).

De *Class View* toont inheritance op een hiërarchische wijze:



20.2 Overriding

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

Als voorbeeld overschrijf je in de derived class *Arbeider* de method *Afbeelden()* van de base class *Werknemer*.

In de method *Afbeelden()* van de class *Werknemer* beeld je enkel werknemer eigenschappen af: *Naam*, *InDienst* en *Geslacht*. In de method *Afbeelden()* van de class *Arbeider* beeld je ook het *Uurloon* en het *Ploegenstelsel* af.

Eerst moet je in de base class (*Werknemer*) toelaten dat een derived class (*Arbeider*) de method *Afbeeld()* kan overschrijven. Je voegt daartoe het sleutelwoord **virtual** toe aan deze method:

```
using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        ...
        public virtual void Afbeelden() (1)
        {
            Console.WriteLine("Naam: {0}", Naam);
            Console.WriteLine("Geslacht: {0}", Geslacht);
            Console.WriteLine("In dienst: {0}", InDienst);
            Console.WriteLine("Personeelsfeest: {0}", Personeelsfeest);
        }
        ...
    }
}
```

- (1) Met het sleutelwoord **virtual** laat je toe dat derived classes deze method overschrijven.

Voeg een method *Afbeeld()* toe aan de class *Arbeider*. Om de method *Afbeeld()* van de base class te overschrijven, voeg je het sleutelwoord **override** toe aan deze method:

```
using System;
namespace CSharpPFCursus
{
    public class Arbeider : Werknemer
    {
        ...
        public override void Afbeelden() (1)
        {
            base.Afbeelden(); (2)
            Console.WriteLine("Uurloon: {0}", Uurloon); (3)
            Console.WriteLine("Ploegenstelsel: {0}", Ploegenstelsel);
        }
        ...
    }
}
```

- (1) Je overschrijft de method *Afbeeld()* van de base class.

Je geeft dit aan met het sleutelwoord **override**.

- (2) Je wil bij een *Arbeider* ook de properties *Naam*, *Geslacht* en *InDienst* afbeelden. De method *Afbeeld()* van de base class (*Werknemer*) doet dit reeds. Je roept de *Afbeeld()* method van de base class op met het sleutelwoord **base**. Dit sleutelwoord staat voor de base class van de huidige class. Als je bij deze opdracht *Afbeelden()*; schrijft (onder **base**.), dan roep je binnen de method *Afbeeld()* nog eens de method *Afbeeld()* van de class *Arbeider* zelf op, en komt je programma in een eindeloze loop.

Bij objectgeoriënteerd programmeren zal een method die een method van de base class overschrijft, meestal als eerste opdracht de method van de base class oproepen.

- (3) Je toont de extra property *UurLoon* van het *Arbeider* object. Op de volgende lijn toon je ook de extra property *Ploegenstelsel*.

	<p>Een method die een method uit de base class overschrijft, is standaard zelf overschrijfbaar (<i>overrideable</i>) in een afgeleide class. In ons voorbeeld overschrijft de method <i>Afbeeld()</i> van de class <i>Arbeider</i> de method <i>Afbeeld()</i> van de class <i>Werknemer</i>. Een afgeleide class van <i>Arbeider</i> kan op zijn beurt de method <i>Afbeeld()</i> overschrijven. Als je dit wil verhinderen, kan je het sleutelwoord <i>sealed</i> toevoegen aan de method <i>Afbeeld()</i> in de class <i>Arbeider</i>.</p>
---	--

20.3 Inheritance en constructors

20.3.1 Basisregels

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.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Arbeider ik = new Arbeider(); (1)
            ik.Afbeeld(); (2)
        }
    }
}
```

- (1) De derived class (*Arbeider*) heeft geen default constructor. De compiler maakt achter de schermen een default constructor aan, die op zijn beurt de default constructor van de base class (*Werknemer*) oproept. De default constructor van de class *Werknemer* plaatst o.a. de property *InDienst* op de systeemdatum.
- (2) Je ziet dat de property *InDienst* de systeemdatum bevat.

20.3.2 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(...)`
- Je kan een constructor uit de base class oproepen: `base(...)`

	<ul style="list-style-type: none"> • Ook als je de constructor van de base class oproeft (via <code>base(...)</code>), doe je dit vóór de open accolade van de huidige constructor. • Zodra je constructor een andere constructor van de base class oproeft, roept C# niet meer automatisch de default constructor van de base class op. • Als een constructor geen oproep van een andere constructor van de base class bevat, roept de compiler de default constructor van de base class op. Als de base class geen default constructor heeft, krijg je een fout.
---	---

Maak een constructor met parameters in de class *Arbeider*:

```
using System;
namespace CSharpPFCursus
{
    public class Arbeider : Werknemer
    {
        ...
        public Arbeider(string naam, DateTime inDienst, Geslacht geslacht,
            decimal uurloon, byte ploegenstelsel) (1)
            : base(naam, inDienst, geslacht) (2)
        {
            Uurloon = uurloon; (3)
            Ploegenstelsel = ploegenstelsel;
        }
        ...
    }
}
```

- (1) De constructor krijgt de beginwaarden van een nieuw *Arbeider* object binnen als parameters.
- (2) De constructor roept eerst de constructor van de base class (*Werknemer*) op en geeft zijn eigen parameterwaarden door aan die constructor. De oproep van de constructor van de base class doe je met het sleutelwoord `base`. De *Werknemer* constructor controleert deze waarden op geldigheid en neemt ze op in de bijbehorende properties.
- (3) Je neemt de nieuwe waarde van het uurloon binnen. Bij het invullen van de property *UurLoon* voert C# de `set` van deze property uit, die enkel positieve getallen toelaat voor de `private` variabele *uurloonValue*.

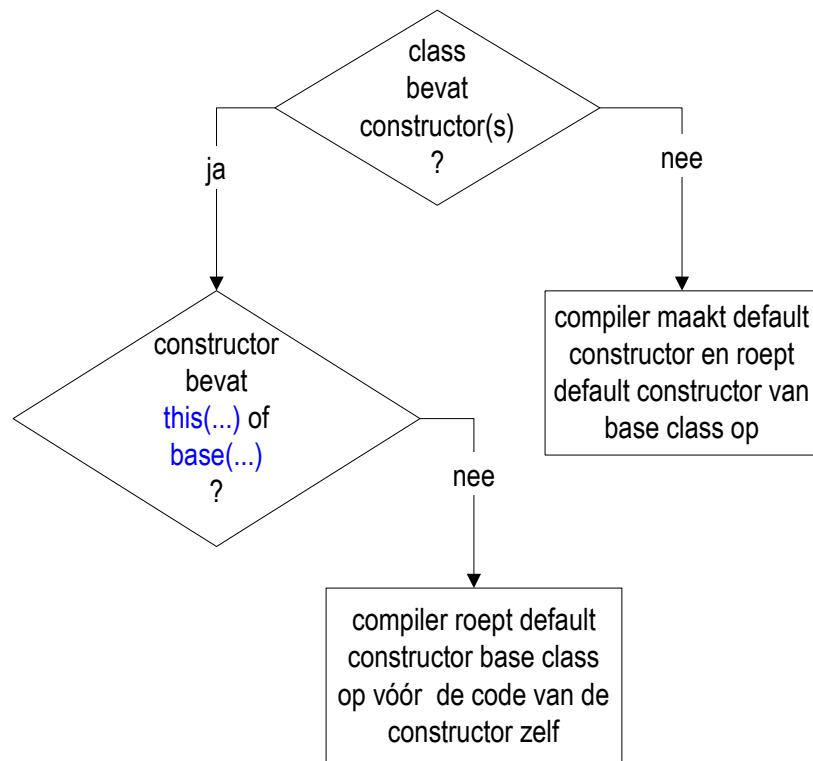
Je kan deze constructor uitproberen:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Arbeider ik = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            ik.Afbeelden();
        }
    }
}

```

20.4 Overzicht van inheritance en constructors



20.5 Andere voorbeelden van Inheritance

Voeg aan het project *CSharpPFCursus* een nieuwe class *Bediende* (*Bediende.cs*) op basis van de class *Werknemer* toe.

In de class *Bediende* doe je het volgende:

- een property *Wedde* toevoegen.
- de method *Afbeelden()* overschrijven.
- een *constructor met parameters* maken.

```

using System;
namespace CSharpPFCursus
{
    public class Bediende : Werknemer
    {
        public Bediende(string naam, DateTime indienst,
            Geslacht geslacht, decimal wedde)
            : base(naam, indienst, geslacht)
        {
            Wedde = wedde;
        }

        private decimal weddeValue;
        public decimal Wedde
        {
            get
            {
                return weddeValue;
            }
            set
            {
                if (value >= 0m)
                    weddeValue = value;
            }
        }

        public override void Afbeelden()
        {
            base.Afbeelden();
            Console.WriteLine("Wedde: {0}", Wedde);
        }
    }
}

```

Je kan deze nieuwe class uitproberen:

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Bediende ik = new Bediende("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m);
            ik.Afbeelden();
        }
    }
}

```

Inheritance is mogelijk op meerdere niveaus: je kan van een derived class terug een class afleiden. Als voorbeeld voeg je aan het project een nieuwe class *Manager* (*Manager.cs*) toe, die gebaseerd is op de class *Bediende* (die reeds gebaseerd was op de class *Werknemer*).

In de class *Manager* doe je het volgende:

- een property *Bonus* toevoegen.
- de method *Afbeeld()* overschrijven.
- een *constructor met parameters* maken.

```
using System;
namespace CSharpPFCursus
{
    public class Manager : Bediende
    {
        public Manager(string naam, DateTime indienst, Geslacht geslacht,
                      decimal wedde, decimal bonus)
            : base(naam, indienst, geslacht, wedde)
        {
            Bonus = bonus;
        }

        private decimal bonusValue;
        public decimal Bonus
        {
            get
            {
                return bonusValue;
            }
            set
            {
                if (value > 0m)
                    bonusValue = value;
            }
        }

        public override void Afbeelden()
        {
            base.Afbeelden();
            Console.WriteLine("Bonus: {0}", Bonus);
        }
    }
}
```

Je kan deze nieuwe class uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
```

```

    {
        Manager ik = new Manager("Asterix", new DateTime(2014, 1, 1),
            Geslacht.Man, 2400.79m, 7000m);
        ik.Afbeelden();
    }
}
}

```

20.6 De class Object

Een class die niet erft van een andere class (in ons voorbeeld *Werknemer*) krijgt toch een base class, namelijk de class *Object*, die in .NET ingebakken is. Je kan deze class beschouwen als “de moeder aller classes”.

Het sleutelwoord **object** (met kleine o) is een synoniem voor de class *Object*.

	De class <i>Object</i> bevat enkele methods die je in je eigen classes beter overschrijft.
---	--

20.6.1 De *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 het project waarin de class gedefinieerd is, terug, gevuld door de naam van de class.

```

using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Manager ik = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m);
            Console.WriteLine(ik.ToString());
        }
    }
}

```

Je ziet *CSharpPFCursus.Manager* op het scherm.

Je kan de method *ToString()* in je eigen classes overschrijven, zodat de method rijkere informatie (bvb. de belangrijkste properties) van het object als een string teruggeeft:

```

using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        ...
    }
}

```

```
    public override string ToString()
    {
        return Naam + '♂' + Geslacht;                                (1)
    }
    ...
}
```

```
using System;
namespace CSharpPFCursus
{
    public class Arbeider : Werknemer
    {
        ...
        public override string ToString()
        {
            return base.ToString() + '♂' + Uurloon + "€uro/uur";      (2)
        }
        ...
    }
}
```

```
using System;
namespace CSharpPFCursus
{
    public class Bediende : Werknemer
    {
        ...
        public override string ToString()
        {
            return base.ToString() + '♂' + Wedde + "€uro/maand";
        }
        ...
    }
}
```

```
using System;
namespace CSharpPFCursus
{
    public class Manager : Bediende
    {
        ...
        public override string ToString()
        {
            return base.ToString() + ", Bonus:" + Bonus;
        }
        ...
    }
}
```

- (1) Je voegt de belangrijkste properties van een Werknemer samen tot één string en je geeft deze string terug als het resultaat van de *ToString()* method.
- (2) Je roept eerst de *ToString()* method van de base class (*Werknemer*) op en voegt aan deze string nog de extra property *Uurloon* van een *Arbeider* toe.

Je kan de eigen *ToString()* method uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Manager ik = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m);
            Console.WriteLine(ik.ToString());
        }
    }
}
```

De *WriteLine()* method van de class *Console* aanvaardt ook een object als parameter (bvb. `Console.WriteLine(ik);`).

De *WriteLine()* method voert dan zelf de *ToString()* method van dat object uit. Je kan het voorbeeld dus ook zo schrijven:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Manager ik = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m);
            Console.WriteLine(ik);
        }
    }
}
```

20.6.2 Equals

Deze method heeft als resultaatwaarde *true* als het huidige object gelijk is aan een ander object en *false* als het huidige object verschillend is van een ander object.

Deze method heeft in de class *Object* een beperkte betekenis: ze vergelijkt enkel de **verwijzingen** van twee reference variabelen (zoals de == vergelijkingsoperator):

```
using System;
namespace CSharpPFCursus
{
```

```

class Program
{
    public static void Main(string[] args)
    {
        Manager ik = new Manager("Asterix", new DateTime(2014, 1, 1),
            Geslacht.Man, 2400.79m, 7000m);
        Manager mezelf = ik;                                (1)
        Manager dezelfde = new Manager("Asterix", new DateTime(2014, 1, 1),
            Geslacht.Man, 2400.79m, 7000m);                  (2)
        Console.WriteLine(ik.Equals(mezelf));                (3)
        Console.WriteLine(ik.Equals(dezelfde));              (4)
    }
}

```

- (1) Je maakt een reference variabele die naar hetzelfde object verwijst als de reference variabele *ik*.
- (2) Je maakt een object met exact dezelfde waarden voor de properties als het object waar de reference variabelen *ik* en *mezelf* naar verwijzen.
- (3) *ik* en *mezelf* verwijzen naar hetzelfde object, het resultaat is **true**.
- (4) *ik* en *dezelfde* verwijzen niet naar hetzelfde object, het resultaat is **false**.

	<p>Je kan in je eigen classes de <i>Equals()</i> method overschrijven, zodat je twee objecten als gelijk kan beschouwen als bepaalde properties van die twee objecten inhoudelijk gelijk zijn. Om over deze inhoudelijke gelijkheid te beslissen vergelijk je één of meerdere properties van deze objecten.</p>
---	---

```

using System;
namespace CSharpPFCursus
{
    public class Werknemer
    {
        public override bool Equals(object obj)                      (1)
        {
            if (obj is Werknemer)                                  (2)
            {
                Werknemer deAndere = (Werknemer)obj;           (3)
                return this.Naam == deAndere.Naam;                 (4)
            }
            else
                return false;                                    (5)
        }
        public override int GetHashCode()                         (6)
        {
            return Naam.GetHashCode();                         (7)
        }
    }
}

```

- (1) De method `Equals()` is in de class `Object` beschreven als een method die een parameter van het type `Object` aanvaardt en een `bool` teruggeeft. De parameter `obj` is het object dat je met het huidig object wil vergelijken. De `Equals()` method geeft `true` terug bij gelijkheid, anders `false`.
- (2) Gezien het te vergelijken object als `Object` binnentkomt, zou een programmeur met de `Equals()` method het huidige object (van het type `Werknemer`) met om het even welk type object (bvb. een `string`) kunnen vergelijken. Dit heeft weinig zin. Daarom controleer je eerst of de parameter `obj` van het type `Werknemer` (of van een afgeleid type van `Werknemer`) is. De expressie `obj is Werknemer` geeft `true` als `obj` van het type `Werknemer` is of van één van de afgeleide types van `Werknemer` (`Arbeider`, `Bediende` of `Manager`). Anders geeft deze expressie `false`. Het sleutelwoord `is` wordt in de volgende paragraaf behandeld.
- (3) Je declareert een reference variabele van het type `Werknemer`. Je doet een expliciete typeconversie van de parameter `obj` naar het type `Werknemer`. Je kan dit door (`Werknemer`) te schrijven vóór de te converteren variabele of parameter (casting).
- (4) Je vergelijkt de naam van het huidige `Werknemer` object met de naam van het `Werknemer` object dat binnentkomt. Als de namen gelijk zijn, besluit je dat het over twee dezelfde werknemers gaat en je geeft als resultaat van `Equals()` `true` terug. Als de namen niet gelijk zijn, geef je `false` terug.
In de praktijk zou het vergelijken van het nummer van twee werknemers een betere indicator voor twee gelijke werknemers zijn, dan het vergelijken van de namen van twee werknemers.
- (5) Als de parameter niet van het type `Werknemer` is, is de parameter inhoudelijk zeker niet gelijk aan het huidige `Werknemer` object. Dan geef je als resultaat van `Equals()` `false` terug.
- (6) Als je de method `Equals()` overschrijft, moet je ook de method `GetHashCode()` overschrijven, ook al gebruik je de method `GetHashCode()` niet altijd in je programma. Anders krijg je compiler warnings. De method `GetHashCode()` wordt gebruikt in hashtables. Deze vallen buiten het bereik van deze cursus.
- (7) De method `GetHashCode()` moet een `int` teruggeven. Deze `int` moet hetzelfde getal bevatten voor twee objecten die gelijk zijn als je er de `Equals()` method op loslaat. In ons geval wil dit zeggen: twee `Werknemer` objecten die volgens de `Equals()` method gelijk zijn, moeten hetzelfde getal teruggeven als je er de `GetHashCode()` method op loslaat. Je hoeft zelf geen algoritme te schrijven om dit te verwezenlijken. In de `Equals()` method heb je de property `Naam` vergeleken om te beslissen of twee werknemers hetzelfde zijn. De property `Naam` heeft als type `string` (de class `String`). In de class `String` is de method `GetHashCode()` al overschreven zodat twee dezelfde `Strings` hetzelfde getal teruggeven als deze twee `Strings` dezelfde tekst bevatten. Je kan dus de `GetHashCode()` van de naam van de werknemer gebruiken om de `GetHashCode()` method van de class `Werknemer` aan te sturen.

Als je het programma nu terug uitvoert, geeft `ik.Equals(mezelf)` `true`, maar ook `ik.Equals(dezelfde)` geeft `true`.

20.7 **is**

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

Syntax:

object **is** class

Het resultaat van deze expressie is **true** als het object tot de class of een afgeleide class behoort, anders **false**.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            object ik = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m); (1)
            Console.WriteLine(ik is Manager); (2)
            Console.WriteLine(ik is Bediende); (3)
            Console.WriteLine(ik is Werknemer); (4)
            Console.WriteLine(ik is Arbeider); (5)
            Console.WriteLine(ik is string); (6)
        }
    }
}
```

- (1) Je maakt een nieuw *Manager* object, en verwijst ernaar met een variabele van het type **object**. Dit mag in C#: je mag met een variabele van een base class verwijzen naar een object van een derived class.
- (2) Dit geeft **true**, want het object waar de variabele *ik* naar verwijst heeft de class *Manager* als type.
- (3) Dit geeft **true**, want het object waar de variabele *ik* naar verwijst heeft de class *Manager* als type, en de class *Manager* heeft als base class *Bediende*.
- (4) Dit geeft **true**, want het object waar de variabele *ik* naar verwijst heeft de class *Manager* als type, en de class *Manager* heeft als base class *Bediende*, die op zich als base class *Werknemer* heeft.
- (5) Dit geeft **false**, want het object waar de variabele *ik* naar verwijst heeft de class *Manager* als type en die heeft de class *Arbeider* niet als base class.
- (6) Dit geeft **false**, want het object waar de variabele *ik* naar verwijst heeft de class *Manager* als type en die heeft de class **string** niet als base class.

20.8 **as**

Met het sleutelwoord **as** kan je eveneens een typeconversie of typecasting doen.

Eerder in dit hoofdstuk heb je gezien dat je het type van een variabele kan wijzigen door gebruik te maken van *ronde haakjes* ():

```
object obj = new Werknemer("Asterix", DateTime.Today, Geslacht.Man);
```

```
Werknemer deAndere = (Werknemer) obj;
```

Hier wordt de variabele `obj`, die van het type `object` is, geconverteerd of gecast naar het type `Werknemer`.

Je kan deze conversie ook schrijven als:

```
Werknemer deAndere = obj as Werknemer;
```

Bij een typeconversie met het sleutelwoord `as` treedt er **geen** fout op wanneer de conversie niet mogelijk is. Als de conversie mislukt wordt `null` teruggeven, wat je als programmeur kan opvangen.

```
using System.Text;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Werknemer ik = new Bediende("Asterix", DateTime.Today,
                Geslacht.Man, 1500m); (1)
            Werknemer jij = new Arbeider("Obelix", DateTime.Today,
                Geslacht.Man, 10m, 1); (2)

            Bediende hij; (3)

            hij = (Bediende) ik; (4)
            hij.Afbeelden(); (5)

            //hij = (Bediende)jij; (6)
            //hij.Afbeelden();

            hij = ik as Bediende; (7)
            if (hij != null) (8)
                hij.Afbeelden();

            hij = jij as Bediende; (7)
            if (hij != null) (8)
                hij.Afbeelden();
        }
    }
}
```

- (1) Je creëert een nieuw `Bediende` object, en verwijst ernaar met een variabele van het type `Werknemer`. Dit mag in C#: je mag met een variabele van een base class verwijzen naar een object van een derived class.
- (2) Je creëert een nieuw `Arbeider` object, en verwijst ernaar met een variabele van het type `Werknemer`.
- (3) Je creëert een variabele `hij` van het type `Bediende`.

- (4) Je laat deze variabele verwijzen naar het *Bediende* object *ik* uit (1), waarbij een typecasting noodzakelijk is omdat de variabele *ik* van type *Werknemer* is. Deze conversie lukt omdat de variabele *ik* naar een *Bediende* object verwijst.
- (5) Het afbeelden van de bedienedegegevens lukt zonder problemen.
- (6) Je laat nu de variabele *hij* verwijzen naar het *Arbeider* object *jij* uit (2), waarbij een typecasting noodzakelijk is omdat de variabele *jij* van type *Werknemer* is. Deze conversie lukt **niet** omdat de variabele *jij* naar een *Arbeider* object verwijst. Als je het programma uitvoert veroorzaakt deze regel code een fout en stopt het programma. De rest van de code wordt niet meer uitgevoerd.
- (7) In plaats van de ronde haakjes te gebruiken voor de typeconversie, gebruik je het sleutelwoord *as*.
- (8) Als de conversie mislukt, krijgt de variabele *hij* de waarde null. Dit kan je nu testen om het verdere verloop van het programma te bepalen.



oefeningen: Inheritance

21 Abstract classes en sealed classes, abstract methods

21.1 Abstract classes

Een *abstract class* is een class waarvan je geen objecten kan aanmaken.

Je kan wel derived classes van een abstract class afleiden.

Van deze derived classes kan je wel objecten aanmaken, tenzij je deze derived classes ook als abstract declareert.

Maak in ons voorbeeld van de class *Werknemer* een abstract class. Op die manier verhinder je dat een programmeur *Werknemer* objecten maakt (die te algemeen zijn) en verplicht je de derived classes (*Arbeider*, *Bediende* of *Manager*) te gebruiken om objecten te creëren.

Je maakt een class abstract met het sleutelwoord **abstract**:

```
using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer
    {
    }
    ...
}
```

Nu kan je geen *Werknemer* objecten meer aanmaken, maar wel objecten van een afgeleide class:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Werknemer();  (1)
            Manager jij = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m); (2)
        }
    }
}
```

(1) Fout: je kan geen objecten aanmaken van een abstracte class.

(2) Juist: je kan wel objecten aanmaken van een afgeleide class van een abstracte class.

Je kan wel reference variabelen aanmaken met als type een abstract class.

Je kan echter met deze reference variabelen niet naar objecten van de abstract class verwijzen, enkel naar objecten van derived classes.

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer ik = new Manager("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 2400.79m, 7000m); (1)
        }
    }
}
```

(1) Het type van de reference variabele is de *abstract class Werknemer*.

Het type van het object waar de reference variabele naar verwijst is de derived class *Manager*.

Dit is handig bij *polymorphism*, wat later in de cursus aan bod komt.

21.2 Sealed classes

Sealed classes zijn classes waarvan je geen andere classes meer kan afleiden.

Je maakt een class *sealed* door het sleutelwoord **sealed** bij de class te vermelden.

In ons voorbeeld kan je de class *Manager* **sealed** maken:

```
using System;
namespace CSharpPFCursus
{
    public sealed class Manager : Bediende
    {
        ...
    }
}
```

Als je een class probeert te baseren op de class *Manager*, krijg je een fout:

```
using System;
namespace CSharpPFCursus
{
    public class Directeur:Manager { ... }
```



21.3 Abstract members

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 schrijft geen code voor de property of method*.

Je verplicht afgeleide classes de methods uit te werken.

Je maakt een abstract property of method met het sleutelwoord **abstract**.

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.

Gezien je geen code mag schrijven in een abstract property of method, mag je ook geen accolades schrijven bij een abstract property of method.

Je sluit een abstract method onmiddellijk na zijn declaratie af met een *puntkomma*. Voorbeeld:

```
public abstract void MijnAbstracteMethod();
```

Je sluit de **get** en **set** van een abstract property onmiddellijk af met een puntkomma. Voorbeeld:

```
public abstract int MijnAbstracteProperty
{
    get;
    set;
}
```



Bij een method met het sleutelwoord **abstract** kan je het sleutelwoord **virtual** niet tikken. Een method met het sleutelwoord **abstract** is automatisch ook **virtual**.

Maak als voorbeeld in de class *Werknemer* een abstracte readonly property *Premie*, die het jaarlijks premiebedrag van een werknemer bevat. Hoe je de premie berekent, hangt af van het soort werknemer: bij een *Arbeider* is de premie het uurloon * 150, bij een *Bediende* de wedde * 2 en bij een *Manager* zijn bonus * 3.

Je kan in de *Werknemer* class zelf nog geen code tikken die de premie berekent.

Definieer in de class *Werknemer* deze readonly property als een abstracte property:

```
using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer
    {
        public abstract decimal Premie
        {
            get;
        }
    }
}
```

```
    }  
}
```

Overschrijf deze property in de derived classes en voorzie ze daar van code:

```
using System;  
namespace CSharpPFCursus  
{  
    public class Arbeider : Werknemer  
    {  
        ...  
        public override decimal Premie  
        {  
            get  
            {  
                return Uurloon * 150m;  
            }  
        }  
    }  
}
```

```
using System;  
namespace CSharpPFCursus  
{  
    public class Bediende : Werknemer  
    {  
        ...  
        public override decimal Premie  
        {  
            get  
            {  
                return Wedde * 2m;  
            }  
        }  
    }  
}
```

```
using System;  
namespace CSharpPFCursus  
{  
    public sealed class Manager : Bediende  
    {  
        ...  
        public override decimal Premie  
        {  
            get  
            {  
                return Bonus * 3m;  
            }  
        }  
    }  
}
```

```
        }
    }
}
```

Je kan nu deze property gebruiken voor objecten van de afgeleide classes:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Arbeider asterix = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            Bediende obelix = new Bediende("Obelix", new DateTime(1995, 1, 1),
                Geslacht.Man, 2400.79m);
            Manager idefix = new Manager("Idefix", new DateTime(1996, 1, 1),
                Geslacht.Man, 2400.79m, 7000m);
            Console.WriteLine(asterix.Premie);
            Console.WriteLine(obelix.Premie);
            Console.WriteLine(idefix.Premie);
        }
    }
}
```



oefeningen: Abstract classes, abstract members, static
members

22 Inheritance polymorphism

22.1 Algemeen

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.

Hieronder verduidelijken we deze zin:

- Een *verzameling* gelijkaardige objecten.
Je maakt een *verzameling* objecten door bvb. een array te creëren waarbij ieder element verwijst naar een object.
- Een *verzameling* gelijkaardige objecten.
Bij inheritance polymorphism zijn objecten gelijkaardig als ze een gemeenschappelijke base class hebben of tot die base class behoren. Later in de cursus zie je ook nog interface polymorphism.

In ons voorbeeld zijn *Arbeider*, *Bediende*, *Manager* en *Werknemer* objecten gelijkaardig: ze hebben eenzelfde base class *Werknemer* of behoren tot die base class.

De array met references naar de gelijkaardige objecten moet als type de gemeenschappelijke base class hebben. In ons voorbeeld moet de array van het type *Werknemer* zijn.

De methods en properties die je voor ieder element kan gebruiken, zijn de methods en properties van de gemeenschappelijke base class (*Werknemer*).

Als je voor een element van de array 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.

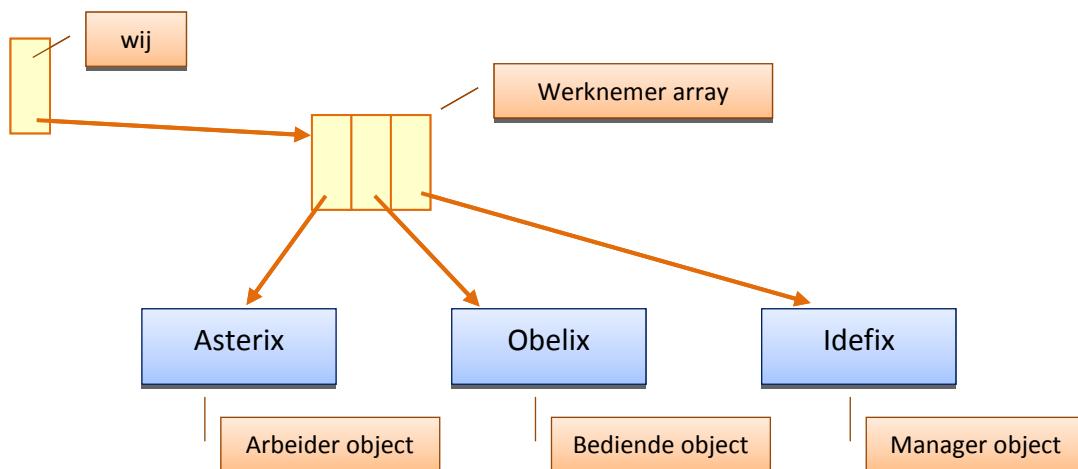
Een voorbeeldprogramma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Werknemer[] wij = new Werknemer[3]; (1)
            wij[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1), (2)
                Geslacht.Man, 24.79m, 3);
            wij[1] = new Bediende("Obelix", new DateTime(1995, 2, 1), (3)
                Geslacht.Man, 2400.79m);
            wij[2] = new Manager("Idefix", new DateTime(1996, 3, 1), (4)
```

```
Geslacht.Man, 2400.79m, 7000m);  
foreach (Werknemer eenWerknemer in wij) (5)  
    eenWerknemer.Afbeelden(); (6)  
}  
}  
}
```

- (1) Je declareert een array van 3 elementen met als type de base class *Werknemer*.
 - (2) Het eerste element verwijst naar een object van het type van de derived class *Arbeider*.
 - (3) Het tweede element verwijst naar een object van het type van de derived class *Bediende*.
 - (4) Het derde element verwijst naar een object van het type van de derived class *Manager*.
 - (5) Met **foreach** doorloop je één voor één de elementen van de array.
 - (6) Je vraagt van ieder element de method *Afbeeld()* uit te voeren.
Het eerste element voert de method *Afbeeld()* van *Arbeider* uit,
het tweede van *Bediende* en het derde van *Manager*.
Elk van deze classes heeft de method *Afbeeld()* opnieuw gedefinieerd.

In het geheugen gebeurt het volgende:



oefeningen: Inheritance polymorfisme

23 Aggregation

23.1 Algemeen

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

Als voorbeeld maak je een class *Afdeling*. Hierin beschrijf je een afdeling waarin werknemers werken. De properties van een *Afdeling* zijn de *afdelingsnaam* en de *verdieping* in het firmagebouw waar deze afdeling zich bevindt.

Je overschrijft ook de method *ToString()* uit de base class (*Object*), die de afdeling als **string** teruggeeft.

Je maakt ook een constructor met als parameterwaarden de *naam* en de *verdieping* van een nieuw *Afdeling* object.

Daarna voeg je aan de *Werknemer* class een property *Afdeling* (type *Afdeling*) toe.

De afgeleide classes (*Arbeider*, *Bediende* en *Manager*) erven deze property.

Maak de class *Afdeling* in een aparte source file:

```
using System;
namespace CSharpPFCursus
{
    public class Afdeling
    {
        public const int Verdiepingen = 10; (1)
        public Afdeling(string naam, int verdieping)
        {
            Naam = naam;
            Verdieping = verdieping;
        }
        private string naamValue;
        public string Naam
        {
            get
            {
                return naamValue;
            }
            set
            {
                naamValue = value;
            }
        }
        private int verdiepingValue;
        public int Verdieping
        {
            get
            {
                return verdiepingValue;
            }
        }
    }
}
```

```

        set
    {
        if (value >= 0 && value <= Verdiepingen) (2)
            verdiepingValue = value;
    }
}
public override string ToString()
{
    return String.Format("Afdeling: {0} op verdieping {1} ",
        Naam, Verdieping);
}
}
}

```

(1) Een public constante met het aantal verdiepingen van het firmagebouw.

(2) Je controleert of de verdieping voor een *Afdeling* object correct is.

En dan nu aggregation zelf.

Je breidt de class *Werknemer* uit met een property *Afdeling*:

```

using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer
    {
        private Afdeling afdelingValue; (1)
        public Afdeling Afdeling (2)
        {
            get
            {
                return afdelingValue;
            }
            set
            {
                afdelingValue = value;
            }
        }
        ...
        public virtual void Afbeelden()
        {
            Console.WriteLine("Naam: {0}", Naam);
            Console.WriteLine("Geslacht: {0}", Geslacht);
            Console.WriteLine("In dienst: {0}", InDienst);
            Console.WriteLine("Personeelsfeest: {0}", Personeelsfeest);
            if (Afdeling != null) (3)
                Console.WriteLine(Afdeling);
        }
        ...
    }
}

```

- (1) Een **private** variabele van het type *Afdeling*.

Het betreft hier een reference variabele: een variabele die op zich geen *Afdeling* object is, maar slechts een verwijzing naar een *Afdeling* object.

Je zal in het hoofdprogramma *Afdeling* objecten aanmaken en *Werknemer* objecten naar die *Afdeling* objecten laten verwijzen. Dit is een voorstelling van de echte wereld: niet iedere werknemer bevat in zich een eigen afdeling. Meerdere werknemers *gebruiken* dezelfde afdeling.

- (2) De property *Afdeling* met zijn **set** en **get** om de reference variabele *afdelingValue* van een *Werknemer* object in te vullen.
- (3) Als de afdeling van een werknemer ingevuld is, toon je de informatie van deze afdeling.

Je kan nu de nieuwe property uitproberen in een testprogramma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Afdeling afdeling1 = new Afdeling("Strijd", 0); (1)
            Afdeling afdeling2 = new Afdeling("Feest", 1); (2)

            Werknemer[] wij = new Werknemer[3];
            wij[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            wij[0].Afdeling = afdeling1; (3)
            wij[1] = new Bediende("Obelix", new DateTime(1995, 2, 1),
                Geslacht.Man, 2400.79m);
            wij[1].Afdeling = afdeling1; (4)
            wij[2] = new Manager("Idefix", new DateTime(1996, 3, 1),
                Geslacht.Man, 2400.79m, 7000m);
            wij[2].Afdeling = afdeling2; (5)
            foreach (Werknemer eenWerknemer in wij)
                eenWerknemer.Afbeeld(en()); (6)
        }
    }
}
```

- (1) Je maakt een *Afdeling* object, initialiseert de properties met de constructor en verwijst met een variabele naar het object.
- (2) Je maakt een tweede *Afdeling* object, initialiseert de properties met de constructor en verwijst met een variabele naar het nieuwe object.
- (3) Je verwijst met de property *Afdeling* van dit *Arbeider* object naar het *eerste Afdeling* object.
- (4) Je verwijst met de property *Afdeling* van dit *Bediende* object naar *hetzelfde Afdeling* object.
- (5) Je verwijst met de property *Afdeling* van dit *Manager* object naar het *tweede Afdeling* object.
- (6) Je toont de gegevens van alle objecten, waarbij ook de informatie van het *Afdeling* object, waarnaar deze objecten verwijzen, getoond wordt.



oefeningen: Aggregation

24 Nested en partial classes

24.1 Nested classes

Een *nested class* – ook wel een *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 een nested class gebruikt men het woord *top-level class*. Dit is een class die een nested class bevat, maar zelf geen nested class is.

Je kan via een sleutelwoord aangeven wat de toegankelijkheid van de nested class is:

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

Maak in het voorbeeldprogramma in de class *Werknemer* een nested class *WerkRegime*. Je beschrijft in deze class de properties en methods van een werknemerregime: *voltijds*, *halftijds*, *viervijfde*:

```
using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer
    {
        public class WerkRegime (1)
        {
            public enum RegimeType (2)
            {
                Voltijds,
                Viervijfde,
                Halftijds
            }
            private RegimeType typeValue; (3)
            public RegimeType Type (4)
            {
                get

```

```

        {
            return typeValue;
        }
        set
        {
            typeValue = value;
        }
    }
    public int Vakantie (5)
    {
        get
        {
            switch (Type)
            {
                case RegimeType.Voltijds:
                    return 25;
                case RegimeType.Viervijfde:
                    return 20;
                case RegimeType.Halftijds:
                    return 12;
                default:
                    return 0;
            }
        }
    }
    public WerkRegime(RegimeType type) (6)
    {
        Type = type;
    }
    public override string ToString() (7)
    {
        return Type.ToString();
    }
}
}
}

```

- (1) De class *WerkRegime* is een *nested class*: ze is gedeclareerd binnen de class *Werknemer*.
- (2) Er bestaan 3 types regime: *voltijds*, *halftijds* en *viervijfde*.
Je beschrijft deze mogelijkheden met een *enum*.
- (3) Een variabele die het type regime bijhoudt.
- (4) Een property voor het type regime.
- (5) Een readonly property met het aantal vakantiedagen van een regime.
- (6) Een constructor voor een *WerkRegime* object.
- (7) Je overschrijft de *ToString()* method uit de base class (*Object*), zodat ze duidelijke informatie over een regime teruggeeft.

Nu kan je per *Werknemer* zijn regime onthouden, door in de *Werknemer* class een variabele en een property van het type *WerkRegime* te voorzien:

```
using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer
    {
        ...
        private WerkRegime regimeValue;
        public WerkRegime Regime
        {
            get
            {
                return regimeValue;
            }
            set
            {
                regimeValue = value;
            }
        }
        ...
    }
}
```

Nu kan je de nested class uitproberen:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Arbeider ik = new Arbeider("Asterix", DateTime.Today,
                Geslacht.Man, 24.79m, 3);
            ik.Regime = new Werknemer.WerkRegime(
                Werknemer.WerkRegime.RegimeType.Voltijds); (1)
            Console.WriteLine(ik.Regime); (2)
        }
    }
}
```

- (1) Je vult de property *Regime* van de *Arbeider* met een nieuw *WerkRegime* object. Omdat de *WerkRegime* class een *nested class* is, moet je de volledige weg beschrijven: *Werknemer.WerkRegime*. Je vermeldt de top-level class, een punt en de nested class.
- (2) Je gebruikt de constructor van de *WerkRegime* class om de property *Type* van het *WerkRegime* object in te vullen. Ook hier moet je de volledige weg aangeven naar één van de *RegimeType enum* mogelijkheden:

`Werknemer.WerkRegime.RegimeType.Voltijds`

Je kan dit interpreteren als: `TopLevelClass.NestedClass.Enum.EnumMogelijkheid`

24.2 Partial classes

Wanneer 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. Die deelbeschrijvingen van de class kan je bovendien desgewenst over meerdere source files spreiden.

Om aan te duiden dat een class uit meerdere delen bestaat, gebruik je het sleutelwoord **partial**.

Voorbeeld:

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

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

Zo zou je de definitie van de inner class `WerkRegime` uit de classbeschrijving van `Werknemer` kunnen halen. Je zou zelfs nog een stapje verder kunnen gaan en de beschrijving van `WerkRegime` in een aparte source file kunnen onderbrengen.

Voor de class `Werknemer` wordt dit dan:

in één source file `Werknemer.cs`:

```
//Eerste deel van de class Werknemer
public abstract partial class Werknemer
{
    ...
}

//Tweede deel van de class Werknemer
public abstract partial class Werknemer
{
    public class WerkRegime
    {
        public enum RegimeType
        {
            Voltijds,
```

```
        Viervijfde,
        Halftijds
    }
    ...
}
}
```

of opgesplitst in twee source files:

Werknemer.cs:

```
//Eerste deel van de class Werknemer
public abstract partial class Werknemer
{
    private string naamValue;
    ...
}
```

en *WerkRegime.cs*:

```
//Tweede deel van de class Werknemer
public abstract partial class Werknemer
{
    public class WerkRegime
    {
        public enum RegimeType
        {
            Voltijds,
            Viervijfde,
            Halftijds
        }
        ...
    }
}
```

Deze twee source files moeten uiteraard een verschillende naam hebben daar in eenzelfde project geen twee source files met dezelfde naam kunnen voorkomen.



`partial class Werknemer`

De naam van de **class Werknemer** bij het woord **partial** geeft aan dat het source files van dezelfde class *Werknemer* zijn.

25 Interfaces

25.1 Algemeen

In een interface definieer je een *contract*.

Dit contract is een verzameling van property declaraties 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 voor de properties en methods. Je beschrijft in een interface dus *wat* er gebeurt, niet *hoe* het gebeurt.

Je werkt de code voor de properties en methods van een interface uit in een class. Als een class de code voor een interface bevat, zegt men dat de class de interface *implementeert*.

De relatie tussen interfaces en classes is breed:

- een class kan meerdere interfaces implementeren.
- meerdere classes kunnen dezelfde interface implementeren.

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

Een interface zelf kan gebaseerd zijn op één of meerdere andere interface(s).

De interface erft dan de method en property declaraties van de base interface(s).

Met interfaces kan je:

- Classes met elkaar in verband brengen die geen gemeenschappelijke base class hebben, maar wel gemeenschappelijk gedrag.
Je declareert het gemeenschappelijk gedrag in een interface.
Je implementeert de interface in de classes en je werkt het gedrag uit door de methods en properties van de interface in de classes code te geven.
De class `Werknemer` en de class `Vrachtwagen` hebben geen gemeenschappelijke base class. Ze hebben echter gemeenschappelijk gedrag: een werknemer is een kost en een vrachtwagen is een kost. De methods en properties die deze kost beschrijven, neem je op in een interface `IKost`. De classes `Werknemer` en `Vrachtwagen` implementeren deze interface.
- Een beperkt beeld geven van een uitgebreide class. Een klant van een bank bekijkt de class `Bank` anders dan een bediende van een bank.
Je maakt een interface `IBankKlant` die enkel de methods en properties bevat vanuit het standpunt van een klant. Je maakt een interface `IBankBediende` waarin je enkel de methods en properties voorziet vanuit het standpunt van een bediende. In een programma voor een klant van de bank benader je een `Bank` object vanuit de interface `IBankKlant`.

Microsoft raadt aan de naam van een interface met **I** (hoofdletter **i**) te beginnen en ieder volgend woord binnen de naam van de interface met een hoofdletter te beginnen.

25.2 Een interface ontwerpen

Je ontwerpt in het voorbeeld een interface *IKost* met een readonly property *Bedrag* (type `decimal`) en een readonly property *Menselijk* (type `bool`). Deze properties geven het bedrag van de kost van een werknemer (arbeider, bediende, manager, ...) of van een voorwerp (fotokopiemachine, gebouw, ...) terug en of het al dan niet om een menselijke kost gaat. De interface beschrijft enkel de declaratie van deze properties *Bedrag* en *Menselijk*.

Je maakt een interface best in een aparte source file:

- Kies in het menu *PROJECT* de opdracht *Add New Item*.
- Kies in het midden de template *Interface*.
- Tik bij *Name*: de naam van de nieuwe interface: *IKost*.
- Druk op de knop *Add*.

Waar een class begint met `class`, begint een interface met `interface`. Een interface plaats je normaal ook in een namespace. Dit heeft als voordeel dat de interface *IKost* uit de namespace CSharpPFCursus niet in naamconflict komt met de interface *IKost* uit een andere namespace. Een interface is standaard *internal*.

Daarna declareer je de methods en properties van de interface tussen accolades:

```
using System;
namespace CSharpPFCursus
{
    interface IKost
    {
        decimal Bedrag
        {
            get;
        }
        bool Menselijk
        {
            get;
        }
    }
}
```

- | | |
|---|--|
|  | <ul style="list-style-type: none">• Bij de properties en methods van een interface kan je geen toegangsniveau (<code>private</code>, <code>public</code>, ...) vermelden. Ze zijn altijd <code>public</code>. Bij de interface zelf kan je wel een toegangsniveau vermelden, bvb. <code>public</code> als je deze interface ook in andere projecten (in een andere assembly) wil gebruiken.• Om een interface te laten erven van één of meerdere andere interfaces, tik je na de naam van de interface de naam of namen van de base interfaces, gescheiden door een komma:
<code>public interface AfgeleideInterface:BaseInterface1, BaseInterface2</code>
{} |
|---|--|

25.3 Een interface implementeren

Je implementeert de interface *IKost* in de class *Werknemer*.

Voor de property *Menselijk* geef je **true** terug.

De property *Bedrag* kan je in de *Werknemer* class nog geen code geven.

Je kan het bedrag van de kost van een werknemer pas berekenen in de afgeleide classes (*Arbeider*, *Bediende*, *Manager*), daar de kost voor elk van deze afgeleide types anders berekend moet worden. Je moet echter alle members van de interface in de class vermelden als een class de interface implementeert.

Je lost dit op door de property *Bedrag* in de *Werknemer* class als **abstract** te beschrijven en deze property concreet uit te werken in de afgeleide classes.

We spreken af dat de kost van een *arbeider* zijn *uurloon* * 2000 is, van een *bediende* zijn *wedde* * 12 en van een *manager* zijn *wedde* * 12 + *bonus*.

Je vermeldt de interface(s) die een class implementeert na de naam van de class en een dubbele punt.

Je implementeert de interface *IKost* in de class *Werknemer*. VS helpt je hierbij:

- Tik ná de naam van de class *Werknemer* :*IKost* en positioneer de cursor in *IKost* → een blauw “handvatje” verschijnt: `public abstract class Werknemer:IKost`
- Positioneer de muisaanwijzer op dit handvatje en open het menu dat nu verschijnt:



- Kies de optie *Implement interface ‘IKost’*. De twee properties *Bedrag* en *Menselijk* worden onderaan in de code van de class toegevoegd:

```
public decimal Bedrag
{
    get { throw new NotImplementedException(); }
}

public bool Menselijk
{
    get { throw new NotImplementedException(); }
}
```

- Vul de code van deze properties in:

```
using System;
namespace CSharpPFCursus
{
    public abstract class Werknemer:IKost
    {
        ...
        public bool Menselijk
    }
}
```

(1)

(2)

```
{  
    get  
    {  
        return true;  
    }  
}  
  
public abstract decimal Bedrag  
(3)  
{  
    get;  
}  
}  
}
```

- (1) Je vermeldt bij de class de interface(s) die deze class implementeert.
- (2) Een property die de property *Menselijk* uit de interface *IKost* implementeert.
- (3) Een property die overeenstemt met de property *Bedrag* uit de interface *IKost*. Gezien je de property hier nog geen code kan geven, vermeld je **abstract**. Zo verplicht je de afgeleide classes deze property code te geven.



Als een class een interface implementeert, dan implementeert een afgeleide class deze interface automatisch ook. Je hoeft dit dan bij de afgeleide class **niet** meer aan te geven door ná de naam van de afgeleide class een dubbele punt en de interface te vermelden.

Je overschrijft de property *Bedrag* in de classes *Arbeider*, *Bediende* en *Manager*:

```
using System;  
namespace CSharpPFCursus  
{  
    public class Arbeider : Werknemer  
    {  
        ...  
        public override decimal Bedrag  
        {  
            get  
            {  
                return Uurloon * 2000m;  
            }  
        }  
    }  
}
```

```
using System;  
namespace CSharpPFCursus  
{  
    public class Bediende : Werknemer  
    {  
        ...  
        public override decimal Bedrag
```

```

        {
            get
            {
                return Wedde * 12m;
            }
        }
    }
}

using System;
namespace CSharpPFCursus
{
    public sealed class Manager : Bediende
    {
        public override decimal Bedrag
        {
            get
            {
                return base.Bedrag + Bonus;
            }
        }
    }
}

```

Voeg aan het project *CSharpPFCursus* een nieuwe class *Fotokopiemachine* (*Fotokopiemachine.cs*) toe, die ook de interface *IKost* implementeert.

Deze class bevat:

- Een property *SerieNr* (type `string`).
- Een property *AantalBlz* (type `int`), die het aantal gekopieerde pagina's voorstelt en enkel positieve waarden aanvaardt.
- Een property *KostPerBlz* (type `decimal`), die de kopieerkost van één pagina voorstelt en enkel positieve waarden aanvaardt, groter dan 0.
- Een geparametriseerde constructor met de parameters *serieNr*, *aantalBlz* en *kostPerBlz*.
- De readonly property *Menselijk*, die `false` teruggeeft.
- De readonly property *Bedrag*: de waarde van dit bedrag wordt berekend als *AantalBlz* * *KostPerBlz*.

```

using System;
namespace CSharpPFCursus
{
    public class Fotokopiemachine : IKost
    {
        private string serieNrValue;
        private int aantalBlzValue;

```

```
private decimal kostPerBlzValue;

public string SerieNr
{
    get
    {
        return serieNrValue;
    }
    set
    {
        serieNrValue = value;
    }
}

public int AantalBlz
{
    get
    {
        return aantalBlzValue;
    }
    set
    {
        if (value >= 0)
            aantalBlzValue = value;
    }
}

public decimal KostPerBlz
{
    get
    {
        return kostPerBlzValue;
    }
    set
    {
        if (value > 0)
            kostPerBlzValue = value;
    }
}

public Fotokopiemachine(string serieNr, int aantalBlz,
    decimal kostPerBlz)
{
    SerieNr = serieNr;
    AantalBlz = aantalBlz;
    KostPerBlz = kostPerBlz;
}

public bool Menselijk
{
    get
    {
```

```
        return false;
    }
}

public decimal Bedrag
{
    get
    {
        return AantalBlz * KostPerBlz;
    }
}
```

In de *Class View* zie je de interface apart vermeld en ook nog eens vermeld bij iedere class die de interface implementeert:



- Een class kan meerdere interfaces implementeren, de verschillende interfaces worden gescheiden door een komma.
`public class Class:Interface1, Interface2`
 - Als een class afgeleid is van een andere class en ook één of meerdere interfaces implementeert, vermeld je eerst de base class en vervolgens de interface(s), gescheiden door een komma.
`public class Class:Base class, Interface1, Interface2`

25.4 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.

Hieronder verduidelijken we deze zin:

- Een **verzameling** gelijkaardige objecten.
Dit kan een array zijn, waarbij ieder element naar een object verwijst.
- Een verzameling **gelijkaardige** objecten.
Bij interface polymorphism zijn objecten gelijkaardig als ze een gemeenschappelijke interface implementeren.

In ons voorbeeld zijn *Fotokopiemachine*, *Arbeider*, *Bediende*, *Manager* en *Werknemer* objecten gelijkaardig: ze implementeren dezelfde interface *IKost*.

De array met references naar de gelijkaardige objecten moet als type *de gemeenschappelijke interface* hebben. In ons voorbeeld moet de array dus van het type *IKost* zijn.

De methods en properties die je voor ieder element kan gebruiken, zijn de methods en properties van de gemeenschappelijke interface (*IKost*).

Als je voor een element van de array een method uitvoert of een property inleest of opvraagt, roept C# de code van de method of property van de class van het object op, waar het arrayelement naar verwijst.

In het voorbeeldprogramma implementeren de classes *Arbeider*, *Bediende*, *Manager* en *Fotokopiemachine* de property *Bedrag* uit de interface *IKost*.

Je kan een *array* van het type *IKost* declareren en met de arrayelementen naar *Arbeider*, *Bediende*, *Manager* of *Fotokopiemachine* objecten verwijzen. Als je voor een arrayelement de property *Bedrag* opvraagt, roept C# de code op van de property *Bedrag* van de class van het object waarnaar het arrayelement verwijst.

In het volgende voorbeeld maak je *Arbeider*, *Bediende*, *Manager* en *FotokopieMachine* objecten. Daarna maak je een array van *IKost* variabelen, waarmee je naar deze objecten verwijst. Je doorloopt de arrayelementen en toont van elk element de properties *Menselijk* en *Bedrag*.

Door het resultaat van de method *Bedrag* van ieder arrayelement te totaliseren, verkrijg je de totale kost van de firma:

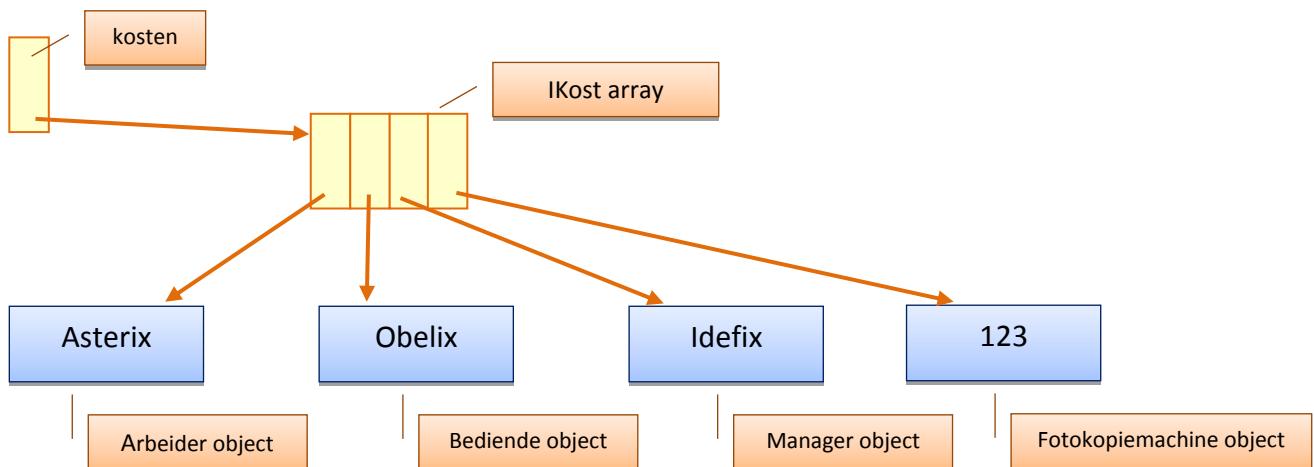
```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            IKost[] kosten = new IKost[4]; (1)

            kosten[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            kosten[1] = new Bediende("Obelix", new DateTime(2014, 2, 1),
                Geslacht.Man, 2400.79m);
            kosten[2] = new Manager("Idefix", new DateTime(2014, 3, 1),
                Geslacht.Man, 2400.79m, 7000m);
```

```
kosten[3] = new Fotokopiemachine("123", 500, 0.025m);  
  
decimal totaleKost = 0m;  
foreach (IKost kost in kosten)  
{  
    Console.WriteLine(kost.Menselijk);  
    Console.WriteLine(kost.Bedrag);  
    totaleKost += kost.Bedrag;  
}  
Console.WriteLine(totaleKost);  
}  
}
```

- (1) Je maakt een array van het type *IKost*.
In de volgende opdrachten vul je deze array met verwijzingen naar objecten van classes die deze interface implementeren.
 - (2) Je voorziet een variabele om de totale kost van de firma te berekenen.
 - (3) Je doorloopt alle elementen van de array en toont de properties *Menselijk* en *Bedrag* van elk element. Merk op dat het *Bedrag* voor elk element op de juiste manier berekend wordt: de code van de property *Bedrag* van de class van het object waarnaar het arrayelement verwijst, wordt uitgevoerd.
 - (4) Je totaliseert de kosten.
 - (5) Je toont de totale kost.

In het geheugen gebeurt het volgende:



	<p>Je kan geen objecten instantiëren van het type van een interface. Dus <code>IKost kost = new IKost();</code> KAN NIET!!!</p> <p>Je kan wel reference variabelen declareren met als type het type van een interface. Deze reference variabelen kan je dan laten verwijzen naar objecten met als type een class (of</p>
---	--

een afgeleide class) die deze interface implementeert.

```
IKost kost = new Fotokopiemachine("123", 1000, 10m);
```

25.5 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.

Syntax:

```
object is interface
```

Het resultaat van deze expressie is **true** als het object tot een class (of een afgeleide class) behoort die de interface (of een afgeleide interface) implementeert, anders **false**.

Voorbeeld:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Object[] dingen = new Object[3]; (1)
            dingen[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            dingen[1] = new Fotokopiemachine("Racekyo", 500, 0.025m);
            dingen[2] = "C#";
            foreach (Object ding in dingen) (2)
                Console.WriteLine(ding is IKost);
        }
    }
}
```

- (1) Je maakt een array van het type *Object* ("de moeder aller classes"). Met een element van deze array kan je dus verwijzen naar alle afgeleide classes van *Object*: *Arbeider*, *Fotokopiemachine*, *string*, ...
- (2) Je voert voor ieder element van de array de **is** operator uit om te controleren of het object waar het element naar verwijst, een class is die de interface *IKost* implementeert. Het eerste element verwijst naar een *Arbeider* object. De class *Arbeider* is afgeleid van de class *Werknemer* die de interface *IKost* implementeert. Daarom krijg je **true**.
Het tweede element verwijst naar een *Fotokopiemachine* object. De class *Fotokopiemachine* implementeert de interface *IKost*. Daarom krijg je **true**.
Het derde element verwijst naar een *string*. De class *String* implementeert de interface *IKost niet*. Daarom krijg je **false**.



oefeningen: Interfaces

26 Namespaces

26.1 Algemeen

Een *namespace* is de naam voor een verzameling van bij elkaar horende types (*classes*, *interfaces*, *enums*).

Dat je types (bvb. *classes*) in een verzameling kan plaatsen en deze verzameling een naam kan geven, is interessant als je types van meerdere programmeurs of softwarefirma's tegelijkertijd wil gebruiken.

Je kan immers een probleem hebben als je een class X gebruikt van de ene softwarefirma en tegelijk een class X gebruikt van een andere softwarefirma. Beide classes hebben dezelfde naam. Als je een object aanmaakt van het type X, krijg je een compilerfout: C# kan niet raden naar welke van de twee classes je verwijst.

Als de ene softwarefirma de class X opgenomen heeft in de namespace A en de andere softwarefirma de class X opgenomen heeft in de namespace B, kan je de twee classes toch onderscheiden: je kan een object maken van het type A.X (een verwijzing naar de *class X* van de *namespace A*) en je kan een object maken van het type B.X (een verwijzing naar de *class X* van de *namespace B*).

Namespaces zijn dus interessant om naamconflicten tussen classes en interfaces op te lossen.

	Namespaces hebben geen verband met de toegankelijkheid van classes (<code>public</code> , <code>private</code> , ...).
---	---

26.2 Geneste namespaces

Binnen een namespace kan je nog eens één of meerdere namespaces voorzien.

Een softwarefirma met 2 afdelingen kan dus een namespace definiëren voor de softwarefirma zelf, en daarbinnen een namespace per afdeling. Zo kan je twee classes met dezelfde naam, van verschillende afdelingen van de softwarefirma, toch onderscheiden.

De volledige verwijzing naar een class is dan:

NaamBuitensteNamespace.NaamGenesteNamespace.NaamClass

Zo'n volledige verwijzing heet een *fully qualified name*.

26.3 Naamafspraken

Microsoft stelt de volgende afspraak voor bij de naamgeving en de structuur van de namespaces die programmeurs creëren:



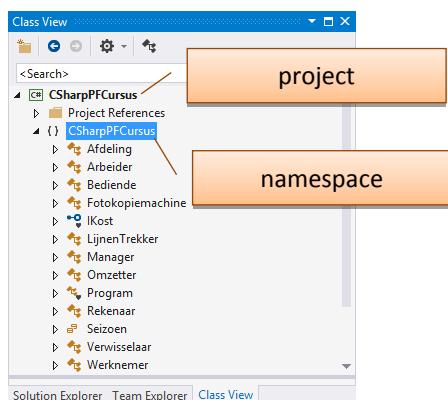
Als je firma bvb. VDAB is, en je maakt classes voor de opleidingen van VDAB, maak je een namespace VDAB en daarin een geneste namespace Opleidingen. Je begint ieder woord binnen de namespacenaam met een hoofdletter.

De volledige verwijzing naar de class *Cursist* is dan:
VDAB.Opleidingen.Cursist

26.4 Standaard namespace

Als je een class aan een project toevoegt, plaatst C# deze class automatisch in een namespace die dezelfde naam heeft als het project.

Je kan dit zien in de *Class View*:



Je kan de naam van deze standaard namespace wijzigen in de properties van het project:

- Klik met de rechtermuisknop op het project *CSharpPFCursus* in de Solution Explorer.
 - Kies de opdracht *Properties*.
 - Kies links in het venster de tab *Application*.
 - Rechts in het venster zie je bij *Default namespace* de standaard namespacesnaam: *CSharpPFCursus*.

Als je deze namespacenaam in de properties van het project wijzigt, past VS.NET deze nieuwe namespace enkel toe in nieuwe sources die je aan het project toevoegt. Bestaande sources behouden de oude namespace.

26.5 Namespaces in de sources

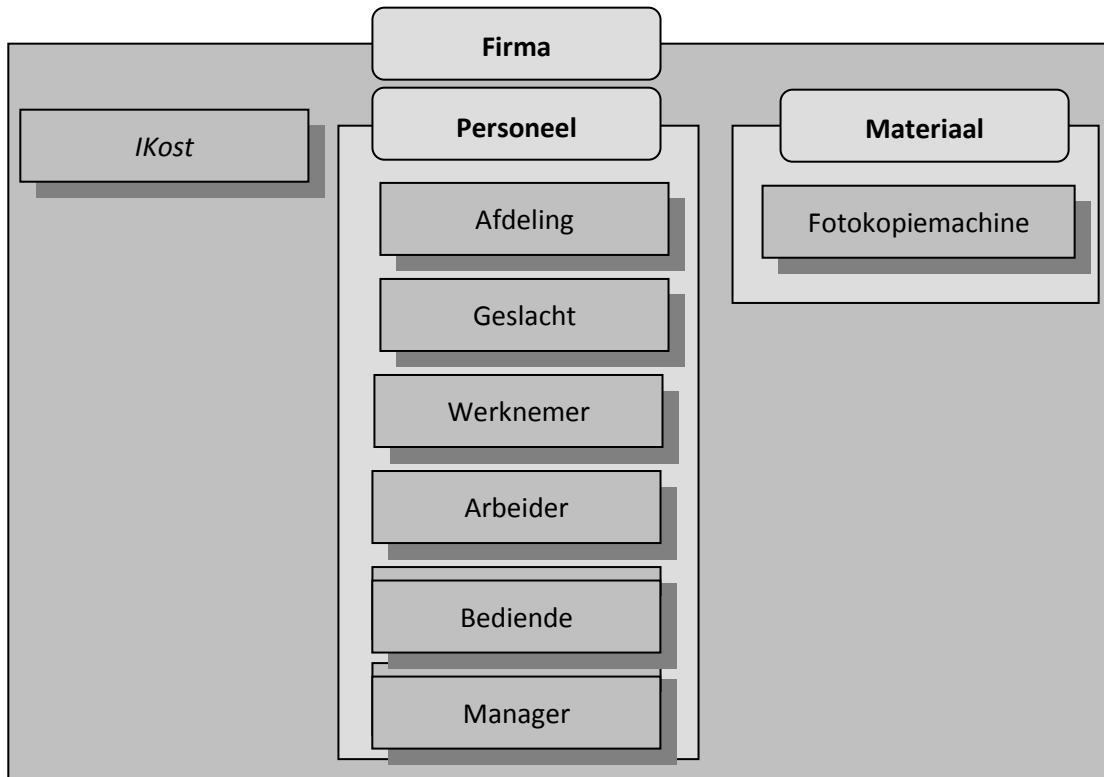
Je kan in de source files afwijken van de standaard namespacenaam. Hierbij kan je ook nested namespaces toepassen.

Je maakt een **namespace Firma**, met een nested **namespace Personeel** en een nested **namespace Materiaal**.

In de **namespace Firma** zelf plaats je de interface **IKost**. Je plaatst deze interface niet in de nested namespaces omdat IKost zowel voor **Personeel** als voor **Materiaal** bruikbaar is.

In de nested **namespace Personeel** verzamel je de classes **Afdeling**, **Werknemer**, **Arbeider**, **Bediende** en **Manager** en de **enum Geslacht**.

In de nested **namespace Materiaal** plaats je de class **Fotokopiemachine**.



Je wijzigt source file per source file:

namespace Firma

{

public interface IKost

source IKost.cs

...

namespace Firma.Personeel

{

class Afdeling

source Afdeling.cs

...

namespace Firma.Personeel

{

enum Geslacht

source Geslacht.cs

...

namespace Firma.Personeel

{

class Werknemer:IKost

source Werknemer.cs

...

namespace Firma.Personeel

{

class Arbeider:Werknemer

source Arbeider.cs

...

```
namespace Firma.Personeel
```

```
{
    class Bediende:Werknemer
    ...
}
```

source Bediende.cs

```
namespace Firma.Personeel
```

```
{
    class Manager:Bediende
    ...
}
```

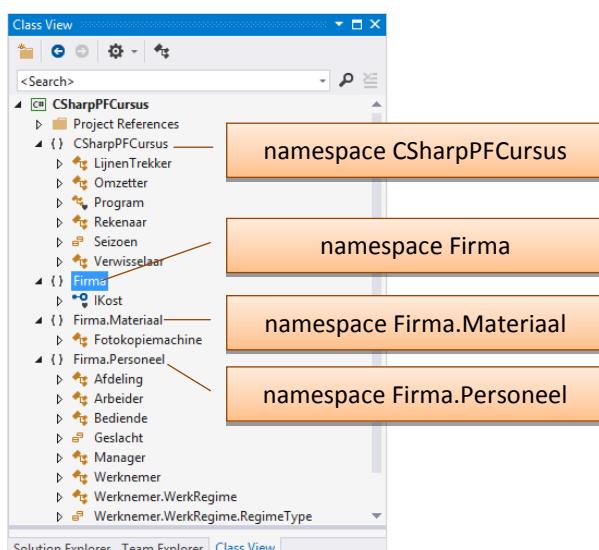
source Manager.cs

```
namespace Firma.Materiaal
```

```
{
    class Fotokopiemachine:IKost
    ...
}
```

source Fotokopiemachine.cs

De nieuwe onderverdeling in namespaces zie je duidelijk in de *Class View*:



Het hoofdprogramma (*Program*) behoort niet tot de **namespace** Firma. Je kan in dit hoofdprogramma niet meer zomaar verwijzen naar de classes uit de namespaces *Firma*, *Firma.Personeel* en *Firma.Materiaal*, je moet volledige verwijzingen gebruiken:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Object[] dingen = new Object[3];
            dingen[0] = new Firma.Personeel.Arbeider(
                "Asterix", new DateTime(2014, 1, 1),
                Firma.Personeel.Geslacht.Man, 24.79m, 3);
            dingen[1] = new Firma.Materiaal.Fotokopiemachine("123", 500, 0.025m);
            dingen[2] = "C#";
            foreach (Object ding in dingen)
                Console.WriteLine(ding is Firma.IKost);
        }
    }
}
```

```
    }  
}
```

26.6 using

Het is vervelend om altijd met volledige verwijzingen te moeten werken als je classes of interfaces gebruikt uit een andere **namespace** dan de huidige.

Wanneer je veel classes en/of interfaces uit een andere **namespace** gebruikt, kan je de **namespace** één keer vermelden (importeren) bovenaan in je source met het sleutelwoord **using**, gevolgd door een namespacenaam, gevolgd door een puntkomma.

In de rest van de source kan je dan naar classes en interfaces uit die namespace verwijzen met hun korte naam.

Je mag meerdere **using** opdrachten onder elkaar vermelden.

Als je een geneste **namespace** wil importeren, moet je die expliciet apart importeren, zelfs als je zijn omsluitende **namespace** reeds geïmporteerd hebt:

```
using System;                                     (1)  
using Firma;                                     (2)  
using Firma.Materiaal;  
using Firma.Personeel;  
namespace CSharpPFCursus  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            Object[] dingen = new Object[3];  
            dingen[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),  
                Geslacht.Man, 24.79m, 3);  
            dingen[1] = new Fotokopiemachine("123", 500, 0.025m);          (3)  
            dingen[2] = "C#";  
            foreach (Object ding in dingen)  
                Console.WriteLine(ding is IKost);  
        }  
    }  
}
```

- (1) Je importeert de **namespace** *Firma*.
- (2) Je importeert de geneste **namespace** *Firma.Materiaal*.
- (3) Je kan de korte namen van de classes (zonder **namespace**) gebruiken.

26.7 Oplossing naamconflicten

In het volgende voorbeeld tonen we aan dat namespaces naamconflicten kunnen oplossen.

Je maakt een **enum Status** die de status van een stuk materiaal beschrijft (*Werkend, Defect*) en je maakt een **enum Status** die de status van een werknemer beschrijft (*HogerKader, LagerKader, Uitvoerend*). Gezien beide enums dezelfde naam hebben, kan je ze standaard niet samen gebruiken.

Door echter de ene `enum` op te nemen in de `namespace Firma.Materiaal` en de andere `enum` op te nemen in de `namespace Firma.Personeel`, kan je ze toch samen gebruiken als je er naar verwijst via hun volledige naam.

Maak in een nieuwe source file `StatusMateriaal.cs` (menu *PROJECT*, opdracht *Add New Item*, template *Code File*) de enum `Status` voor het materiaal:

```
namespace Firma.Materiaal
{
    enum Status
    {
        Werkend,
        Defect
    }
}
```

Maak in een andere nieuwe source file `StatusPersoneel.cs` (menu *Project*, opdracht *Add New Item*, template *Code File*) de enum `Status` voor het personeel:

```
namespace Firma.Personeel
{
    enum Status
    {
        HogerKader,
        LagerKader,
        Uitvoerend
    }
}
```

Wijzig het hoofdprogramma om aan te tonen dat je van beide enums een variabele kan maken via hun volledige naam:

```
using System;
using Firma;
using Firma.Materiaal;
using Firma.Personeel;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Firma.Materiaal.Status statusBoorMachine =
                Firma.Materiaal.Status.Werkend;
            Firma.Personeel.Status statusChef =
                Firma.Personeel.Status.HogerKader;
            Console.WriteLine(statusBoorMachine);
            Console.WriteLine(statusChef);
        }
    }
}
```

26.8 Alias

Als je `using` in de source gebruikt, kan je ook een alias (bijnaam) geven aan de geïmporteerde namespace. Verder in de source kan je dan deze alias gebruiken in plaats van de namespace of class. Als je voor de alias een korte naam kiest, kan je tikwerk besparen.

Syntax om een *namespace* te importeren en een alias te geven:

```
using Alias=NameSpace;
```

Syntax om een *class*, *interface* of *enum* van een namespace te importeren en een alias te geven:

```
using Alias=NameSpace.Class;
using Alias=NameSpace.Interface;
using Alias=NameSpace.Enum;
```

Je kan dit toepassen in het voorbeeldprogramma:

```
using System;
using Firma;
using MateriaalStatus = Firma.Materiaal.Status; (1)
using PersoneelStatus = Firma.Personeel.Status; (2)
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            MateriaalStatus statusBoorMachine = MateriaalStatus.Werkend; (3)
            PersoneelStatus statusChef = PersoneelStatus.HogerKader; (4)
            Console.WriteLine(statusBoorMachine);
            Console.WriteLine(statusChef);
        }
    }
}
```

- (1) Je importeert de `enum Firma.Materiaal.Status` en geeft deze de alias `MateriaalStatus`.
- (2) Je importeert de `enum Firma.Personeel.Status` en geeft deze de alias `PersoneelStatus`.
- (3) Je gebruikt de kortere alias (`MateriaalStatus`).
- (4) Je gebruikt de kortere alias (`PersoneelStatus`).

27 Delegates en events

27.1 Delegates

Tot nu toe heb je reeds gebruikt gemaakt van *reference variabelen* om te verwijzen naar *objecten*.

```
Arbeider ik = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                           Geslacht.Man, 24.79m, 3);
```

Hier is de variabele `ik` een reference variabele van het type `Arbeider`, een *class* die je zelf gedefinieerd hebt. Deze variabele bevat een verwijzing (een reference) naar een `Arbeider` object in het intern geheugen.

Je hebt ook gezien dat je *reference variabelen* kan declareren met als type een *interface*. Deze reference variabelen kan je dan laten verwijzen naar objecten met als type een *class* (of een afgeleide *class*) die deze interface implementeren.

```
IKost kost = new Fotokopiemachine("123", 1000, 10m);
```

Hier is de variabele `kost` een reference variabele van het type `IKost`, een *interface* die je zelf gedefinieerd hebt en door de *class* `Fotokopiemachine` geïmplementeerd wordt.

Deze reference variabele bevat een verwijzing (een reference) naar een `Fotokopiemachine` object in het intern geheugen.

Een reference variabele kan ook van het type **delegate** zijn.

Een *delegate* type 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* (zie verder) of een *lambda expressie* (zie verder).

	<ul style="list-style-type: none"> Net zoals een <i>class</i> type of <i>interface</i> type, moet je ook eerst het <i>delegate</i> type zelf definiëren. Daarna kan je variabelen declareren met als type deze <i>delegate</i>. Met zo een <i>delegate</i> type variabele kan je dan verwijzen naar eender welke method, op voorwaarde dat deze method voldoet aan de definitie van de <i>delegate</i>. Vervolgens kan je de method uitvoeren via deze <i>delegate</i> variabele.
---	--

27.1.1 Werkwijze

Het voorbeeld wordt in de volgende paragraaf uitgewerkt.

- Eerst moet je dus een *delegate* type **definiëren**. Bij de **definitie** van de *delegate* leg je de **signature** 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.

Je doet dit met het sleutelwoord **delegate**, gevolgd door de signature van de method waar je naar wil verwijzen.

- Declaratie van een delegate die verwijst naar een method zonder resultaatwaarde:

```
delegate void DelegateMethod(parameters);
```

- Declaratie van een delegate die verwijst naar een method met resultaatwaarde (functie):

```
delegate returnType DelegateFunction(parameters);
```

- Na de definitie van het delegatetype definieer je een reference variabele met als type deze delegate.
Met deze delegate variabele kan je dan verwijzen naar een method die je via deze delegate variabele wil uitvoeren. Deze method moet dezelfde *signature* hebben als de delegate:
de method moeten dus hetzelfde (of een compatibel) returntype en eenzelfde aantal, type (of compatibel type) en volgorde van de parameters hebben.
- Vervolgens kan de method waar de delegate variabele naar verwijst, uitgevoerd worden via deze delegate variabele. Als deze method parameters verwacht, moet je deze hier meegeven.

Delegates worden o.a. gebruikt om methods als parameters door te geven aan andere methods. De parameters die de methods ontvangen, hebben dan als type een delegate.

Op die manier kan er tijdens de uitvoering van de programmacode nog beslist worden welke method uitgevoerd wordt, bvb. afhankelijk van een keuze die de gebruiker maakt.

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 signature van de delegate op basis waarvan het event gedeclareerd is. Dit wordt verder in dit hoofdstuk uitgelegd.

27.1.2 Voorbeeld

- Een delegate definiëren

Als voorbeeld maak je een delegate *WerknemersLijst*.

Met variabelen van dit type delegate willen we verwijzen naar een method die een array van Werknemers als parameter binnenkrijgt en de werknemers uit de array op het scherm toont (een method zonder returnwaarde, dus van het type **void**):

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

Je zal deze code straks in het programma toevoegen.

Eerst ontwerpen we twee methods die voldoen aan deze signature:

De eerste method *UitgebreideWerknemersLijst()* toont *alle gegevens* van de werknemers van

de array die als parameter meegegeven wordt.

De tweede method *KorteWerknemersLijst()* toont een beperkt aantal gegevens van de werknemers.

Via een delegate variabele van het type *WerknemersLijst* kunnen deze methods dan in het programma uitgevoerd worden.

- De methods toevoegen in de class *Werknemer*
 - Voeg een eerste method *UitgebreideWerknemersLijst()* toe aan de class *Werknemer*. Deze method toont de gegevens van de werknemers van de array, door voor iedere werknemer de method *Afbeeld()* uit te voeren:

```
using System;
namespace Firma.Personeel
{
    public abstract class Werknemer:IKost
    {
        public static void UitgebreideWerknemersLijst(Werknemer[] werknemers)
        {
            Console.WriteLine("Uitgebreide werknemerslijst:");
            foreach (Werknemer werknemer in werknemers)
                werknemer.Afbeeld();
        }
    }
}
```

- Voeg een tweede method *KorteWerknemersLijst()* toe aan de class *Werknemer*. Deze method toont de gegevens van de werknemers, door voor iedere werknemer het resultaat van de method *ToString()* op het scherm te tonen.

```
using System;
namespace Firma.Personeel
{
    public abstract class Werknemer:IKost
    {
        public static void KorteWerknemersLijst(Werknemer[] werknemers)
        {
            Console.WriteLine("Verkorte werknemerslijst:");
            foreach (Werknemer werknemer in werknemers)
                Console.WriteLine(werknemer.ToString());
        }
    }
}
```

Je kan nu een reference variabele van het type *WerknemersLijst* laten verwijzen naar de method *UitgebreideWerknemersLijst()* of naar de method *KorteWerknemersLijst()*.

	De twee methods zijn beschreven als static methods. Dit is omdat een werknemerslijst niet op één werknemer slaat, maar op een verzameling werknemers. Een method met class bereik (static) is een method die je uitvoert op de class (<i>Werknemer</i>) en niet op
---	--

een Werknemer object. Dit is de beste voorstelling van een method die op een verzameling werknemers slaat.
Delegates kan je koppelen aan **static** methods en gewone methods.
Delegates zijn dus niet enkel toepasbaar op **static** methods.

- De delegate *WerknemersLijst* gebruiken:

```
using System;
using Firma;
using Firma.Personeel;
namespace CSharpPFCursus
{
    class Program
    {
        delegate void WerknemersLijst(Werknemer[] werknemers);          (1)
        public static void Main(string[] args)
        {
            Werknemer[] wij = new Werknemer[3];                            (2)
            wij[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            wij[1] = new Bediende("Obelix", new DateTime(1995, 2, 1),
                Geslacht.Man, 2400.79m);
            wij[2] = new Manager("Idefix", new DateTime(1996, 3, 1),
                Geslacht.Man, 2400.79m, 7000m);

            WerknemersLijst lijst;                                         (3)

            lijst = Werknemer.UitgebreideWerknemersLijst;                  (4)
            lijst(wij);                                                 (5)
            Console.WriteLine();

            lijst = Werknemer.KorteWerknemersLijst;                      (6)
            lijst(wij);                                                 (7)
        }
    }
}
```

- (1) Je definieert een **delegate** type *WerknemersLijst*.
Reference variabelen met als type deze **delegate** zullen naar een method kunnen verwijzen die als parameter een *array van Werknemers* binnenkrijgt.
- (2) Je maakt een array van werknemers.
- (3) Je maakt een reference variabele met als type de **delegate** *WerknemersLijst*.
- (4) Je laat deze variabele verwijzen naar de method *UitgebreideWerknemersLijst()*. Daar deze method *static* is, kan je naar de method refereren via de class *Werknemer*.
- (5) Via de delegate variabele wordt de method die op dit ogenblik gekoppeld is, uitgevoerd.
Je moet de parameter die de method verwacht (een array van Werknemers) meegeven.
Het resultaat van de method *UitgebreideWerknemersLijst()* wordt op het scherm getoond.

- (6) Je laat de delegate reference variabele nu verwijzen naar de method *KorteWerknemersLijst()*.
- (7) Via de delegate variabele wordt de gekoppelde method uitgevoerd. Je moet de parameter die de method verwacht (een array van Werknemers) meegeven. Het resultaat van de method *KorteWerknemersLijst()* wordt op het scherm getoond.

	<ul style="list-style-type: none"> Je kan een delegate reference variabele ook op de volgende manier naar een method laten verwijzen: <pre><code>WerknemersLijst lijst; lijst = new WerknemersLijst(Werknemer.UitgebreideWerknemersLijst);</code></pre> <p>Je creëert een delegate object op basis van het delegate type en geeft de method <i>UitgebreideWerknemersLijst()</i> als parameter aan de constructor mee.</p> <ul style="list-style-type: none"> Het delegate type hoeft niet noodzakelijk gedefinieerd te worden in de class waar de delegate gebruikt wordt. Je kan de delegate ook in een aparte source file (bvb. <i>Delegates.cs</i>) definiëren. Hierbij moet je uiteraard wel de nodige namespaces importeren. De signature van de methods moet niet exact overeenkomen met de signature van het delegate type. Methods voldoen aan de signature van de delegate als aan de volgende voorwaarden voldaan is: <ul style="list-style-type: none"> Het <i>type van de parameters</i> van de method mag verschillend zijn van het type van de parameters gedefinieerd door de delegate, op voorwaarde dat conversie tussen de gegevenstypes mogelijk is. Een <i>parameter</i> van een delegate is compatibel met de <i>overeenkomstige parameter van een method</i> als het type van de delegate parameter hetzelfde is of een beperking is (bvb. afgeleid is) van het type van de method parameter. Dit garandeert immers dat de parameter die aan de delegate meegegeven wordt, indien nodig veilig kan gecast worden naar het type van de method parameter (widening conversion). Het <i>type van de returnwaarde</i> 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. Het <i>returntype van de delegate</i> is compatibel met het <i>returntype van de method</i> als het returntype van de method hetzelfde is of een beperking is (bvb. afgeleid is) van het returntype van de delegate. Dit garandeert immers dat het returntype van de method indien nodig veilig kan gecast worden naar het returntype van de delegate (widening conversion).
---	--

27.2 Events

Een **event** is een gebeurtenis die zich kan voordoen in een object: een werknemer neemt ontslag, een fotokopiemachine heeft een onderhoudsbeurt nodig, ...

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, een bediende grijpt in als de fotokopiemachine vastloopt, ...

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 signature van deze method wordt bepaald door het type van het event: een delegate. Het event type bepaalt dus de signature van de methods die kunnen uitgevoerd worden wanneer het event plaatsvindt.

In de volgende modules ga je code schrijven die uitgevoerd moet worden wanneer er tijdens de uitvoering van het programma bepaalde standaard of ingebakken gebeurtenissen of events optreden, bvb. klikken op een button, een keuze maken in een keuzelijst, ...

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.

27.2.1 Werkwijze om dit gedrag in code te modelleren

Als voorbeeld gebruiken we de *Fotokopiemachine* die een event veroorzaakt wanneer de machine een onderhoud nodig heeft. In eerste instantie laten we een *Bediende* object op deze gebeurtenis reageren. Later zal ook een *Manager* object op dezelfde gebeurtenis reageren. Eerst wordt de werkwijze beschreven, in de volgende paragraaf wordt het voorbeeld volledig uitgewerkt in VS.NET.

- Definieer eerst een **delegate** die bepaalt aan welke signature de methods moeten voldoen die op het event reageren:

```
delegate void Onderhoudsbeurt(Fotokopiemachine machine);
```

Deze delegate beschrijft de signature van een method die kan uitgevoerd worden als een fotokopiemachine een onderhoudsbeurt nodig heeft: de vastgelopen fotokopiemachine komt als een parameter binnen.

- Definieer het **event** in de **class waar de gebeurtenis zich kan voordoen** met het sleutelwoord **event**. Gebruik als type van het **event** de delegate die je in het vorige punt gedefinieerd hebt.

In de class *Fotokopiemachine*:

```
public event Onderhoudsbeurt OnderhoudNodig;
```



- Definieer in **de class die op het event reageert**, de bediende, een method die overeenkomt met de **delegate**:

```
public void DoeOnderhoud(Firma.Materiaal.Fotokopiemachine machine){}
```

Bemerk dat de naam van de method niet dezelfde moet zijn als het **delegate** type waarop het **event** gebaseerd is. De signature (returntype **void**, aantal parameters en type parameters **Fotokopiemachine machine**) moet wel gelijk zijn.

- Verbind het **event** met de method van het object dat op het **event** zal reageren:

```
machine.OnderhoudNodig += eenBediende.DoeOnderhoud;
```

Deze opdracht bevat het object waar het **event** zich kan voordoen (een fotokopiemachine **machine**), gevolgd door een punt (.), gevolgd door de naam van het **event** (**OnderhoudNodig**), gevolgd door **+=**, gevolgd door het object dat op het **event** zal reageren (**eenBediende**), gevolgd door een punt (.) en de method waarmee het **Bediende** object op het **event** zal reageren (**DoeOnderhoud**).

Als deze fotokopiemachine onderhoud nodig heeft, zal een bediende reageren door de method **DoeOnderhoud()** uit te voeren.

Indien meerdere objecten op deze gebeurtenis willen reageren, moet je voor elk van deze objecten deze regel code herhalen, uiteraard met een verwijzing naar de juiste method voor elk object.

	<p>Je kan ook op de volgende manier een event koppelen aan een method van het object dat op het event zal reageren:</p> <pre>machine.OnderhoudNodig += new Onderhoudsbeurt(eenBediende.DoeOnderhoud);</pre> <p>Deze opdracht bevat het object waar het event zich kan voordoen (een fotokopiemachine machine), gevolgd door een punt (.), gevolgd door de naam van het event (OnderhoudNodig), gevolgd door +=, gevolgd door de creatie van een delegate object met new. De constructor krijgt als parameter het object dat op het event zal reageren (eenBediende) met de method die op het event zal reageren (DoeOnderhoud).</p>
---	---

- Veroorzaak in het object waarin het event gedeclareerd is (een fotokopiemachine), het **event**. Dit doe je als volgt:

NaamEvent(parametersVoorEvent)

In het voorbeeld van het event **OnderhoudNodig** in de class **Fotokopiemachine**:

```
OnderhoudNodig(this);
```

Hierdoor wordt het event **OnderhoudNodig** veroorzaakt, waarbij het huidige object (de huidige fotokopiemachine = **this**) als parameter meegegeven wordt.

Alle objecten die je met het event verbonden had in de vorige paragraaf

```
machine.OnderhoudNodig += eenBediende.DoeOnderhoud;
```

reageren nu op het event door de method uit te voeren die je met het event associeerde.

Het object *eenBediende* zal nu de method *DoeOnderhoud()* uitvoeren. In de method *DoeOnderhoud()* is een parameter van het type *Fotokopiemachine* gedeclareerd. In deze parameter komt nu de fotokopiemachine binnen die je bij het veroorzaken van het **event** meegeeft.

27.2.2 Het voorbeeld uitwerken in VS.NET

- Definieer de delegate in de source file *Fotokopiemachine.cs*.

```
using System;
namespace Firma.Materiaal
{
    public delegate void Onderhoudsbeurt(Fotokopiemachine machine);
    public class Fotokopiemachine:IKost
    {
        ...
    }
}
```

- Declareer het event *OnderhoudNodig* in de class *Fotokopiemachine*. Dit event heeft als type de delegate *Onderhoudsbeurt* uit de vorige paragraaf.

```
using System;
namespace Firma.Materiaal
{
    public delegate void Onderhoudsbeurt(Fotokopiemachine machine);
    public class Fotokopiemachine:IKost
    {
        public event Onderhoudsbeurt OnderhoudNodig;
        ...
    }
}
```

- Voeg aan de class *Bediende* een method *DoeOnderhoud()* toe, die overeenstemt met de signature van de delegate *Onderhoudsbeurt* en die uitgevoerd zal worden wanneer de gebeurtenis *OnderhoudNodig* plaatsvindt voor een fotokopiemachine.

```
using System;
namespace Firma.Personeel
{
    public class Bediende : Werknemer
    {
        public void DoeOnderhoud(Firma.Materiaal.Fotokopiemachine machine)
        {
            Console.WriteLine("{0} onderhoudt machine {1}", Naam, machine.SerieNr);
```

```

        }
    }
}
```

- Creëer in het hoofdprogramma een *Fotokopiemachine* object en een *Bediende* object. Verbind het event *OnderhoudNodig* van dit *Fotokopiemachine* object met de method *DoeOnderhoud()* van het *Bediende* object, zodat de bediende kan reageren wanneer het event door de fotokopiemachine getriggerd wordt.

```

using System;
using Firma;
using Firma.Personeel;
using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Fotokopiemachine machine = new Fotokopiemachine(
                "123", 0, 2.0m);
            Bediende eenBediende = new Bediende("Asterix",
                DateTime.Today, Geslacht.Man, 2400.79m);
            machine.OnderhoudNodig += eenBediende.DoeOnderhoud;
        }
    }
}
```

- Voeg aan de class *Fotokopiemachine* code toe die het **event** veroorzaakt. Als voorbeeld heeft de machine na 10 blz. gedrukt te hebben een onderhoudsbeurt nodig.

```

using System;
namespace Firma.Materiaal
{
    public delegate void Onderhoudsbeurt(Fotokopiemachine machine);
    public class Fotokopiemachine:IKost
    {
        public event Onderhoudsbeurt OnderhoudNodig;

        //het event OnderhoudNodig veroorzaken indien nodig
        private const int AantalBlzTussen2OnderhoudsBeurten = 10;

        public void Fotokopieer(int aantalBlz)
        {
            for (int blz = 1; blz <= aantalBlz; blz++)
            {
                Console.WriteLine("FotokopieMachine {0} kopieert " +
                    " blz. {1} van {2}.", SerieNr, blz, aantalBlz);
                if (++AantalBlz % AantalBlzTussen2OnderhoudsBeurten == 0)
                    if (OnderhoudNodig != null)
                        OnderhoudNodig(this);
            }
        }
    }
}
```

Het event *OnderhoudNodig* kan de waarde **null** bevatten als geen enkele method met het event zou verbonden zijn. Dan mag het event niet veroorzaakt worden.

- Test het voorbeeld uit door in het hoofdprogramma de method *Fotokopieer()* van het *Fotokopiemachine* object op te roepen, waarbij je het aantal te kopiëren pagina's als parameter meegeeft. Om de 10 bladzijden zal het event *OnderhoudNodig* veroorzaakt worden en bijgevolg de method *DoeOnderhoud()* van het *Bediende* object uitgevoerd worden.

```
using System;
using Firma;
using Firma.Personeel;
using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Fotokopiemachine machine =
                new Fotokopiemachine("123", 0, 2.0m);
            Bediende eenBediende = new Bediende("Asterix",
                DateTime.Today, Geslacht.Man, 2400.79m);
            machine.OnderhoudNodig += eenBediende.DoeOnderhoud;
            machine.Fotokopieer(45);
        }
    }
}
```

Het resultaat ziet er als volgt uit:

```
C:\WINDOWS\system32\cmd.exe - < > x
Fotokopiemachine 123 kopieert blz. 1 van 45.
Fotokopiemachine 123 kopieert blz. 2 van 45.
Fotokopiemachine 123 kopieert blz. 3 van 45.
Fotokopiemachine 123 kopieert blz. 4 van 45.
Fotokopiemachine 123 kopieert blz. 5 van 45.
Fotokopiemachine 123 kopieert blz. 6 van 45.
Fotokopiemachine 123 kopieert blz. 7 van 45.
Fotokopiemachine 123 kopieert blz. 8 van 45.
Fotokopiemachine 123 kopieert blz. 9 van 45.
Fotokopiemachine 123 kopieert blz. 10 van 45.
Asterix onderhoudt machine 123
Fotokopiemachine 123 kopieert blz. 11 van 45.
Fotokopiemachine 123 kopieert blz. 12 van 45.
Fotokopiemachine 123 kopieert blz. 13 van 45.
Fotokopiemachine 123 kopieert blz. 14 van 45.
Fotokopiemachine 123 kopieert blz. 15 van 45.
Fotokopiemachine 123 kopieert blz. 16 van 45.
Fotokopiemachine 123 kopieert blz. 17 van 45.
Fotokopiemachine 123 kopieert blz. 18 van 45.
Fotokopiemachine 123 kopieert blz. 19 van 45.
Fotokopiemachine 123 kopieert blz. 20 van 45.
Asterix onderhoudt machine 123
Fotokopiemachine 123 kopieert blz. 21 van 45.
Fotokopiemachine 123 kopieert blz. 22 van 45.
Fotokopiemachine 123 kopieert blz. 23 van 45.
Fotokopiemachine 123 kopieert blz. 24 van 45.
Fotokopiemachine 123 kopieert blz. 25 van 45.
Fotokopiemachine 123 kopieert blz. 26 van 45.
Fotokopiemachine 123 kopieert blz. 27 van 45.
Fotokopiemachine 123 kopieert blz. 28 van 45.
Fotokopiemachine 123 kopieert blz. 29 van 45.
Fotokopiemachine 123 kopieert blz. 30 van 45.
Asterix onderhoudt machine 123
Fotokopiemachine 123 kopieert blz. 31 van 45.
Fotokopiemachine 123 kopieert blz. 32 van 45.
Fotokopiemachine 123 kopieert blz. 33 van 45.
Fotokopiemachine 123 kopieert blz. 34 van 45.
Fotokopiemachine 123 kopieert blz. 35 van 45.
Fotokopiemachine 123 kopieert blz. 36 van 45.
Fotokopiemachine 123 kopieert blz. 37 van 45.
Fotokopiemachine 123 kopieert blz. 38 van 45.
Fotokopiemachine 123 kopieert blz. 39 van 45.
Fotokopiemachine 123 kopieert blz. 40 van 45.
Asterix onderhoudt machine 123
Fotokopiemachine 123 kopieert blz. 41 van 45.
Fotokopiemachine 123 kopieert blz. 42 van 45.
Fotokopiemachine 123 kopieert blz. 43 van 45.
Fotokopiemachine 123 kopieert blz. 44 van 45.
Fotokopiemachine 123 kopieert blz. 45 van 45.
Press any key to continue . . .
```

Eén object kan reageren op events van meerdere objecten: één bediende kan bvb. reageren op meerdere machines die aan onderhoud toe zijn:

```
using System;
using Firma;
using Firma.Personeel;
using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Fotokopiemachine machine1 = new Fotokopiemachine("123", 0, 2.0m);
            Fotokopiemachine machine2 = new Fotokopiemachine("456", 0, 2.5m);
            Bediende eenBediende =
                new Bediende("Astérix", DateTime.Today, Geslacht.Man, 2400.79m);
            machine1.OnderhoudNodig += eenBediende.DoeOnderhoud;
            machine2.OnderhoudNodig += eenBediende.DoeOnderhoud;
            machine1.Fotokopieer(27);
            machine2.Fotokopieer(42);
        }
    }
}
```

Anderzijds kunnen meerdere objecten reageren op een event van één object.

Als een machine aan onderhoud toe is, kan een bediende reageren met een onderhoudsbeurt en een manager met het registreren in een logboek dat een onderhoud gebeurt.

Breed de class *Manager* uit met een method die met de **delegate** overeenstemt:

```
using System;
namespace Firma.Personeel
{
    public sealed class Manager : Bediende
    {
        public void OnderhoudNoteren(Firma.Materiaal.Fotokopiemachine machine)
        {
            Console.WriteLine("{0} registreert het onderhoud " +
                "van machine {1} in het logboek.", Naam, machine.SerieNr);
        }
    }
}
```

Pas het hoofdprogramma aan:

```
using System;
using Firma;
using Firma.Personeel;
```

```

using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            Fotokopiemachine machine1 = new Fotokopiemachine("123", 0, 2.0m);
            Fotokopiemachine machine2 = new Fotokopiemachine("456", 0, 2.5m);
            Bediende eenBediende =
                new Bediende("Asterix", DateTime.Today, Geslacht.Man, 2400.79m);
            Manager eenManager =
                new Manager("Idefix", DateTime.Today, Geslacht.Man, 4800.4m, 2000m);
            machine1.OnderhoudNodig += eenBediende.DoeOnderhoud;
            machine1.OnderhoudNodig += eenManager.OnderhoudNoteren;
            machine2.OnderhoudNodig += eenBediende.DoeOnderhoud;
            machine2.OnderhoudNodig += eenManager.OnderhoudNoteren;
            machine1.Fotokopieer(49);
            machine2.Fotokopieer(14);
        }
    }
}

```

27.3 Anonymous methods

In de paragraaf over delegates hebben we een korte en een uitgebreide werknemerslijst gegenereerd. We bekijken nog even die code:

```

using System;
using Firma;
using Firma.Personeel;
namespace CSharpPFCursus
{
    class Program
    {
        delegate void WerknemersLijst(Werknemer[] werknemers); (1)
        public static void Main(string[] args)
        {
            Werknemer[] wij = new Werknemer[3]; (2)
            wij[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3); (2)
            wij[1] = new Bediende("Obelix", new DateTime(1995, 2, 1),
                Geslacht.Man, 2400.79m); (2)
            wij[2] = new Manager("Idefix", new DateTime(1996, 3, 1),
                Geslacht.Man, 2400.79m, 7000m); (2)
            WerknemersLijst lijst; (3)
            lijst = Werknemer.UitgebreideWerknemersLijst; (4)
            lijst(wij); (5)
            Console.WriteLine();
            lijst = Werknemer.KorteWerknemersLijst; (6)
            lijst(wij); (7)
        }
    }
}

```

```

        }
    }
}

```

Bovenaan vinden we de definitie van een delegate (1). Vervolgens wordt een array van 3 Werknemers gemaakt (2). Tenslotte creëren we een reference variabele die het type van de delegate heeft (3). Met deze reference variabele verwijzen we naar de method *UitgebreideWerknemersLijst* (4) en voeren deze uit (5), en vervolgens naar de method *KorteWerknemersLijst* (6) en voeren deze eveneens uit (7).

We kunnen aan de delegate reference variabele *lijst* ook een stukje code koppelen dat geen (static) method is in één of andere klasse, maar gewoon ter plaatse genoteerd staat in een blok. Omdat dit stuk code geen naam draagt (in tegenstelling tot *UitgebreideWerknemersLijst* en *KorteWerknemersLijst*) wordt dit een ***anonymous method*** genoemd.

Als voorbeeld laten we de totale kost van een array werknemers berekenen:

```

using System;
using Firma;
using Firma.Personeel;
using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {
        delegate void WerknemersLijst(Werknemer[] werknemers);
        public static void Main(string[] args)
        {
            Werknemer[] wij = new Werknemer[3];
            wij[0] = new Arbeider("Asterix", new DateTime(2014, 1, 1),
                Geslacht.Man, 24.79m, 3);
            wij[1] = new Bediende("Obelix", new DateTime(1995, 2, 1),
                Geslacht.Man, 2400.79m);
            wij[2] = new Manager("Idefix", new DateTime(1996, 3, 1),
                Geslacht.Man, 2400.79m, 7000m);

            WerknemersLijst rapport; (1)

            rapport = delegate(Werknemer[] werknemers) (2)
            {
                decimal totaal = 0m;
                foreach (Werknemer werknemer in werknemers)
                    totaal += werknemer.Bedrag;
                Console.WriteLine("Totale kost is {0}" ,totaal); (3)
            };
            rapport(wij);
        }
    }
}

```

- (1) Omdat we verder niet echt een lijst genereren, maar een totale kost berekenen, noemen we de delegate reference variabele *rapport* i.p.v. lijst.
- (2) We laten *rapport* verwijzen naar een stukje code dat als parameter een werknemerslijst aanvaardt.
- (3) In het onbenoemde stukje code initialiseren we het totaal, lopen we de array af om de totale kost te berekenen en tonen we deze kost op het scherm.
- (4) We voeren de method uit via de delegate reference variabele.

Een anonymous method gebruiken heeft enkel zin als je er zeker van bent dat je de code slechts één keer moet uitvoeren.

Met een *anonymous method* kunnen we ook op een event reageren. In het onderstaande voorbeeld geven we via een anonymous method een melding op het scherm iedere keer als er een onderhoud aangevraagd is. Verder laten we nog steeds een bediende het onderhoud uitvoeren.

```
using System;
using Firma;
using Firma.Personeel;
using Firma.Materiaal;
namespace CSharpPFCursus
{
    class Program
    {

        static void Main(string[] args)
        {
            Fotokopiemachine machine = new Fotokopiemachine("123", 0, 2.0m);           (1)
            Bediende eenBediende =
                new Bediende("Asterix", DateTime.Today, Geslacht.Man, 2400.79m);

            machine.OnderhoudNodig += delegate(Fotokopiemachine apparaat)      (2)
            {
                Console.WriteLine("Onderhoud is aangevraagd voor machine {0}.",
                    apparaat.SerieNr);
            };

            machine.OnderhoudNodig += eenBediende.DoeOnderhoud;                  (3)

            machine.Fotokopieer(49);                                              (4)
        }
    }
}
```

- (1) We maken een fotokopiemachine en een bediende.
- (2) We koppelen een anonymous method, die als parameter een fotokopiemachine aanvaardt, aan het event *OnderhoudNodig*. In de code tonen we op het scherm dat er een onderhoud wordt aangevraagd voor het apparaat.
- (3) We koppelen eveneens een method *DoeOnderhoud* van een bediende, aan het event *OnderhoudNodig*.
- (4) We zorgen ervoor dat het event optreedt.



oefeningen: Delegates en events

28 Exceptions

28.1 Algemeen

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

Voorbeelden van exceptions: je probeert een bestand te openen dat niet bestaat, je probeert data te schrijven naar een schrijfbeveiligde disk, het netwerk valt uit terwijl je data over het netwerk leest, ...

28.2 De class Exception

De exception zelf (het foutsignaal) is een object. Dit object behoort tot de class *Exception* (ingebakken in .NET) 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.

Daarnaast heeft een *Exception* object ook een readonly property *StackTrace* van het type **string**. Met deze property zie je welke methods in uitvoering waren op het moment dat de fout zich voordeed.

28.3 Exceptions opvangen met try - catch

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

Bekijk het volgende programma:

```
using System;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal1, getal2;
            Console.Write("eerste getal:");
            getal1 = decimal.Parse(Console.ReadLine()); (1)
            Console.Write("tweede getal:");
            getal2 = decimal.Parse(Console.ReadLine()); (2)
            Console.WriteLine("deling:" + getal1 / getal2); (3)
        }
    }
}
```

In dit programma kan er op 3 plaatsen een exception optreden:

- (1) Als C# de ingetikte tekst niet naar een getal kan converteren, werpt C# een exception van het type *FormatException* (een afgeleide class van *Exception*).
- (2) Als C# de ingetikte tekst niet naar een getal kan converteren, werpt C# een exception van het type *FormatException*.
- (3) Als het tweede getal nul is (deling door nul), werpt C# een exception van het type *DivideByZeroException* (een afgeleide class van *Exception*).

In dit voorbeeld vang je de fouten niet op. Dus stopt C# het programma bij het optreden van een fout.

Sommige fouten kan je voorkomen met een **if**. Als je fouten kan oppangen met een **if**, is dit de te prefereren manier: het is de meest performante manier.

Zo kan, vooraleer de deling uit te voeren, de fout bij deling door nul opgevangen worden door met een **if** te testen of *getal2* verschillend is van nul:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal1, getal2;
            Console.Write("eerste getal:");
            getal1 = decimal.Parse(Console.ReadLine());
            Console.Write("tweede getal:");
            getal2 = decimal.Parse(Console.ReadLine());
            if (getal2 != 0m)
                Console.WriteLine("deling: " + getal1 / getal2);
            else
                Console.WriteLine("Delen door nul niet toegelaten");
        }
    }
}
```

Je kan echter niet alle mogelijke fouten voorkomen met een **if**.

Om toch te voorkomen dat het programma stopt bij het optreden van een fout, moet je de code die mogelijk fout(en) kan veroorzaken, omsluiten met de opdracht **try** en accolades.

Na de sluitaccoorde 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:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal1, getal2;
            try
            {
                Console.WriteLine("eerste getal:");
                getal1 = decimal.Parse(Console.ReadLine());
                Console.WriteLine("tweede getal:");
                getal2 = decimal.Parse(Console.ReadLine());
                if (getal2 != 0m)
                    Console.WriteLine("deling:" + getal1 / getal2);
                else
                    Console.WriteLine("Delen door nul niet toegelaten");
            }
            catch (1)
            {
                Console.WriteLine("Je tikt geen getal");
            }
        }
    }
}
```

- (1) Deze **catch** opdracht vangt *alle* exceptions op, dus ook exceptions van het type *FormatException* (een afgeleide class van *Exception*).

28.4 Meerdere catch opdrachten

Met meerdere **catch** opdrachten kan je de soorten exceptions (*FormatException*, *DivideByZeroException*, ...) onderscheiden. Je vermeldt dan per **catch** opdracht tussen ronde haakjes het type *Exception* dat die **catch** opdracht verwerkt. Die **catch** opdracht verwerkt dan enkel dat type exception én exceptions die van dat type afgeleid zijn.

Om dit aan te tonen pas je het programma aan, zodat je delen door nul niet opvangt met een **if** (wat je in de praktijk wel doet), maar met een **catch**:

```
namespace CSharpPFCursus
{
    class Program
    {
```

```

static void Main(string[] args)
{
    decimal getal1, getal2;
    try
    {
        Console.WriteLine("eerste getal:");
        getal1 = decimal.Parse(Console.ReadLine());
        Console.WriteLine("tweede getal:");
        getal2 = decimal.Parse(Console.ReadLine());
        Console.WriteLine("deling: " + getal1 / getal2);
    }
    catch (FormatException) (1)
    {
        Console.WriteLine("Je tikt geen getal");
    }
    catch (DivideByZeroException) (2)
    {
        Console.WriteLine("Delen door nul niet toegelaten");
    }
    catch (Exception) (3)
    {
        Console.WriteLine("Een fout heeft zich voorgedaan");
    }
}
}
}

```

- (1) Deze **catch** opdracht vangt fouten op van het type *FormatException*.
- (2) Deze **catch** opdracht vangt fouten op van het type *DivideByZeroException*.
- (3) Deze **catch** opdracht vangt alle andere fouten op.

	<p>Opgepast: de volgorde waarin je de catch opdrachten schrijft is belangrijk. Als een exception optreedt, controleert C# de catch opdrachten in de volgorde zoals jij ze schrijft. Zodra één catch opdracht de exception behandelt, voert C# de daaropvolgende catch opdrachten niet meer uit. Als je de catch opdracht van het type <i>Exception</i> als eerste vermeldt, handelt deze ook de exceptions van het type <i>FormatException</i> en <i>DivideByZeroException</i> af, daar deze types afgeleid zijn van het type <i>Exception</i>. De andere catch opdrachten komen dan nooit aan bod!</p>
---	---

Als je technische informatie over de fout wil weten, vermeld je bij de **catch** opdracht, na het type exception, een variabele. Deze variabele heeft als type de exception class die je bij de **catch** opdracht vermeldt. Tussen de accolades van de **catch** opdracht kan je de properties en methods van deze variabele dan gebruiken.

In het volgende voorbeeld toon je de *Message* property en de *StackTrace* property van een fout die zich voordoet:

```

namespace CSharpPFCursus
{
    class Program

```

```

{
    static void Main(string[] args)
    {
        decimal getal1, getal2;
        try
        {
            Console.WriteLine("eerste getal:");
            getal1 = decimal.Parse(Console.ReadLine());
            Console.WriteLine("tweede getal:");
            getal2 = decimal.Parse(Console.ReadLine());
            if (getal2 != 0m)
                Console.WriteLine("deling:" + getal1 / getal2);
            else
                Console.WriteLine("Delen door nul niet toegelaten");
        }
        catch (FormatException ex) (1)
        {
            Console.WriteLine("Je tikt geen getal");
            Console.WriteLine(ex.Message); (2)
            Console.WriteLine(ex.StackTrace); (3)
        }
    }
}

```

- (1) Je vermeldt bij de **catch** opdracht een variabele (*ex*). Als een *FormatException* optreedt, verwijst deze variabele naar de opgetreden *FormatException*.
- (2) Je toont de *Message* property.
- (3) Je toont de *StackTrace* property.

28.5 Meerdere try opdrachten

Zoals het programma nu geschreven is, kan je in de **catch** opdracht niet weten of de *FormatException* opgetreden is bij het intikken van *getal1* of bij het intikken van *getal2*.

Je kan dit oplossen door meerdere **try** opdrachten te gebruiken, waarbij elke **try** opdracht een bijbehorende **catch** opdracht heeft. Je doet de deling pas als beide **try** opdrachten geen fout veroorzaakten. Daartoe nest je de tweede **try** opdracht in de eerste **try** opdracht:

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal getal1, getal2;
            try (1)
            {
                Console.WriteLine("eerste getal:");
                getal1 = decimal.Parse(Console.ReadLine());

```

```
try (2)
{
    Console.Write("tweede getal:");
    getal2 = decimal.Parse(Console.ReadLine());
    if (getal2 != 0m)
        Console.WriteLine("deling:" + getal1 / getal2);
    else
        Console.WriteLine("Delen door nul niet toegelaten");

}
catch (FormatException) (3)
{
    Console.WriteLine("Je tikt geen getal als tweede getal");
}
catch (FormatException) (4)
{
    Console.WriteLine("Je tikt geen getal als eerste getal");
}
}
```

- (1) Met deze **try** opdracht probeer je *getal1* in te vullen.
 - (2) Met deze (geneste) **try** opdracht probeer je *getal2* in te vullen.
 - (3) Met deze **catch** opdracht vang je de fout op die optreedt bij het verkeerd invullen van *getal2*.
 - (4) Met deze **catch** opdracht vang je de fout op die optreedt bij het verkeerd invullen van *getal1*.

28.6 Een fout melden met throw

Een fout in de code zorgt er niet altijd voor dat het programmaverloop niet kan worden verder gezet. Toch kan de fout ernstig genoeg zijn om een melding te geven aan de gebruiker en bepaalde code niet verder uit te voeren. We nemen het wapen van een exception deze keer zelf in handen. Je doet dit met de [throw](#) opdracht.

Deze opdracht verwacht een *Exception* object: een object van de class *Exception* of van een afgeleide class van *Exception*.

De class `Exception` heeft een constructor waarbij je een `string` kan meegeven, waarin je de fout omschrijft.

De fout die geworpen wordt, kan je opvangen met `try...catch`.

Zodra je een fout werpt, springt C# naar de dichtstbijzijnde `catch` die de fout kan opvangen, of stopt het programma als geen enkele `catch` de fout kan opvangen.

Voorbeeld: wanneer in een *Fotokopiemachine* object de properties *KostPerBlz* of *AantalBlz* verkeerd ingevuld worden met bvb. een negatieve waarde, meld je dat met een exception van het type *Exception*:

```
namespace Firma.Materiaal
{
    public delegate void Onderhoudsbeurt(Fotokopiemachine machine);
    public class Fotokopiemachine : IKost
    {
        ...
        public int AantalBlz
        {
            get
            {
                return aantalBlzValue;
            }
            set
            {
                if (value < 0) (1)
                    throw new Exception("Aantal blz. < 0!");
                aantalBlzValue = value;
            }
        }

        public decimal KostPerBlz
        {
            get
            {
                return kostPerBlzValue;
            }
            set
            {
                if (value <= 0) (2)
                    throw new Exception("Kost per blz. <=0!");
                kostPerBlzValue = value;
            }
        }
        ...
    }
}
```

- (1) Verkeerde waarde voor de property *AantalBlz*. Je maakt een nieuw *Exception* object. Je geeft bij de constructor een foutmelding mee. Je werpt de fout naar degene die de property probeert in te vullen. C# voert de rest van de *set* niet meer uit.
- (2) Verkeerde waarde voor de property *KostPerBlz*. Je maakt een nieuw *Exception* object. Je geeft bij de constructor een foutmelding mee. Je werpt de fout naar degene die de property probeert in te vullen. C# voert de rest van de *set* niet meer uit.

Je kan deze fout opvangen in het hoofdprogramma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Fotokopiemachine machine =
                    new Fotokopiemachine("123", -100, -5.4m);
                Console.WriteLine("Machine goed ingevuld");
            }
            catch (Exception ex)
            {
                Console.WriteLine("Fout:" + ex.Message);
            }
            Console.WriteLine("Einde programma");
        }
    }
}
```

28.7 Een eigen exception class ontwerpen

Je gebruikte in het vorige voorbeeld de class *Exception* om fouten te werpen vanuit de class *Fotokopiemachine*. Dit heeft twee nadelen:

- 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 (bvb. het verkeerde bedrag in *KostPerBlz*), tenzij je deze extra informatie als tekstuele info aan de message toevoegt.

Je kan deze nadelen oplossen door zelf foutclasses te creëren. Eigen foutclasses leid je af van de class *Exception*.

Het eerste nadeel los je op door per fout een aparte class te maken: een class *KostPerBlzException* en een class *AantalBlzException*.

Het tweede nadeel los je op door extra properties met foutinformatie toe te voegen aan deze classes. Aan de class *KostPerBlzException* voeg je een property *VerkeerdeKost* (*decimal*) toe. Aan de class *AantalBlzException* voeg je een property *VerkeerdAantal* (*int*) toe. Je voorziet ook constructors die twee parameters aanvaarden: een omschrijving van de fout (zoals de constructor in de class *Exception*), en de verkeerde waarde voor de nieuwe property.

Je werkt de classes uit als *inner classes* (*nested classes*) van de class *Fotokopiemachine*:

```
namespace Firma.Materiaal
```

```
{  
    public delegate void Onderhoudsbeurt(Fotokopiemachine machine);  
    public class Fotokopiemachine : IKost  
    {  
        public class KostPerBlzException : Exception  
        {  
            private decimal verkeerdeKostValue;  
            public decimal VerkeerdeKost  
            {  
                get  
                {  
                    return verkeerdeKostValue;  
                }  
                set  
                {  
                    verkeerdeKostValue = value;  
                }  
            }  
            public KostPerBlzException(string message,  
                decimal verkeerdeKost)  
                : base(message)  
            {  
                VerkeerdeKost = verkeerdeKost;  
            }  
        }  
  
        public class AantalBlzException : Exception  
        {  
            private int verkeerdAantalBlzValue;  
            public int VerkeerdAantalBlz  
            {  
                get  
                {  
                    return verkeerdAantalBlzValue;  
                }  
                set  
                {  
                    verkeerdAantalBlzValue = value;  
                }  
            }  
            public AantalBlzException(string message,  
                int verkeerdAantalBlz)  
                : base(message)  
            {  
                VerkeerdAantalBlz = verkeerdAantalBlz;  
            }  
        }  
        ...  
  
        public decimal KostPerBlz  
        {  
            get  
        }
```

```

    {
        return kostPerBlzValue;
    }
    set
    {
        if (value <= 0)
            throw new KostPerBlzException("Kost per blz. <=0!", value);      (6)
        kostPerBlzValue = value;
    }
}

public int AantalBlz
{
    get
    {
        return aantalBlzValue;
    }
    set
    {
        if (value < 0)
            throw new AantalBlzException("Aantal blz. < 0!", value);      (7)
        aantalBlzValue = value;
    }
}
...
}

```

- (1) Een *inner class* die een exception voor een foutieve kost per bladzijde definieert. Je leidt eigen exception classes af van de class *Exception*.
- (2) Een **private** member variabele voor de extra foutinformatie.
- (3) Een property voor de extra foutinformatie.
- (4) Een constructor om een fout te maken, met als parameters de omschrijving van de fout en de extra foutinformatie.
- (5) Je geeft de foutomschrijving door aan de constructor van de base class (*Exception*) die deze omschrijving onthoudt.
- (6) Je werpt een fout van het type *KostPerBlzException*.
- (7) Je werpt een fout van het type *AantalBlzException*.

Je kan deze nieuwe classes uitproberen in het hoofdprogramma:

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {

```

```
try
{
    Fotokopiemachine machine =
        new Fotokopiemachine("123", -100, -5.4m);
    Console.WriteLine("Machine goed ingevuld");
}
catch (Fotokopiemachine.KostPerBlzException ex)
{
    Console.WriteLine("Fout:" + ex.Message + ':' +
        ex.VerkeerdeKost);
}
catch (Fotokopiemachine.AantalBlzException ex)
{
    Console.WriteLine("Fout:" + ex.Message + ':' +
        ex.VerkeerdAantalBlz);
}
Console.WriteLine("Einde programma");
}
```

28.8 Gegarandeerde tijdige opkuis van resources

	Om dit onderwerp te kunnen volgen maak je (bvb. met <i>NotePad</i>) een bestand <code>provincies.txt</code> in de map van C:\VS2013. Je tikt in dit bestand de volgende gegevens (provincies en hun oppervlakte in km ²): Antwerpen:2867 Vlaams-Brabant:2106 Waals-Brabant:1091 Henegouwen:3786 Luik:3862 Limburg:2422 Luxemburg:4440 Namen:3666 Oost-Vlaanderen:2942 West-Vlaanderen:3125
---	--

De .NET *garbage collector* neemt het opkuisen van *intern geheugen* dat een programma niet meer nodig heeft voor zijn rekening. Deze opkuis is dus niet de verantwoordelijkheid van de programmeur.

Programma's gebruiken naast intern geheugen ook regelmatig resources. Resources zijn hulpbronnen voor het programma die meer zijn dan enkel intern geheugen.

Voorbeelden van resources:

- een tekstbestand dat het programma leest of schrijft

- een verbinding naar een relationele database
- een netwerkverbinding
- een lettertype dat het programma gebruikt
- ...

De .NET libraries hebben classes om met deze resources te werken. Om te lezen uit een tekstbestand bestaat bijvoorbeeld de class *StreamReader* in de namespace *System.IO*.

Aan de constructor van deze class geef je een string mee met het absolute pad van het bestand dat je wil lezen. De constructor opent dan dit bestand zodat je er later uit kan lezen. Met de volgende opdracht open je bvb. C:\VS2013\provincies.txt:

```
StreamReader lezer=new StreamReader(@"C:\VS2013\provincies.txt");
```



Om in de code de backslash (\) letterlijk als backslash te beschouwen binnen een string, laat je de string voorafgaan door @ (zie verbatim strings).

Daarna kan je het tekstbestand regel per regel lezen met de *ReadLine()* method van je *StreamReader* object. Deze method geeft een **string** met de gelezen regel terug, of **null** als je op het einde van het bestand gekomen bent. Met de volgende opdracht lees je één regel uit het tekstbestand:

```
string regel=lezer.ReadLine();
```

Als je alle regels van het tekstbestand gelezen hebt, sluit je het bestand met de *Close()* method van het *StreamReader* object:

```
lezer.Close();
```

En nu komen we bij de kern van dit onderwerp: als je het bestand vergeet te sluiten, blijft het bestand openstaan terwijl je het in feite niet meer nodig hebt.

Als de variabele *lezer* een lokale variabele van een method is, verwijdert C# de variabele uit het geheugen bij het einde van de method waarin de variabele gedeclareerd is, dus zodra deze method volledig uitgevoerd is.

Het *StreamReader* object waar de variabele naar verwees, blijft echter in het geheugen tot de garbage collector het geheugen opkuist. Op het moment dat de garbage collector een *StreamReader* opkuist, sluit dit object het bestand waarmee het geassocieerd was, tenzij het bestand al gesloten was.

Het *StreamReader* object kan dus korte of lange tijd in het geheugen blijven, de tijd tussen het moment dat jij het niet meer nodig hebt en het moment dat de garbage collector het geheugen opkuist. Gedurende die tijd blijft ook het geassocieerde bestand openstaan. Dit heeft de volgende nadelen:

- Je kan het bestand in die tijd niet verwijderen.

- Je kan het bestand in die tijd niet hernoemen.
- Als je het bestand exclusief opende, kunnen anderen het bestand in die tussentijd niet openen.
- Als je het bestand exclusief opende, kan je het bestand niet meenemen in een backup procedure van de harde schijf.
- ...

Het is dus best om het *StreamReader* object onmiddellijk expliciet te sluiten met de *Close()* method zodra je het niet meer nodig hebt.

Ook andere resources dan *StreamReader* sluit je best onmiddellijk expliciet na gebruik, zodat ook deze niet onnodig in gebruik blijven tot de garbage collector het geheugen opkuist.

De method om een resource te sluiten kan, bij de classes die een resource voorstellen, twee namen hebben: *Close()* of *Dispose()*.

Als je dus een class gebruikt uit de .NET libraries die een *Close()* of *Dispose()* method bevat, pas je best deze method toe van zodra je het object niet meer nodig hebt.

Je maakt als voorbeeld een class *ProvincieInfo*. Deze class bevat een method *ProvincieGrootte()*. Deze method krijgt een *string* parameter binnen met de *naam* van een provincie. Deze method geeft als resultaatwaarde een *int* terug met de grootte van deze provincie. De method leest de gegevens uit C:\VS2013\provinces.txt.

Voeg een nieuwe class *ProvincieInfo* (*ProvincieInfo.cs*) toe aan het project *CSharpPFCursus*.

```
using System;
using System.IO; (1)
namespace CSharpPFCursus
{
    public class ProvincieInfo
    {
        public int ProvincieGrootte(string provincieNaam) (2)
        {
            StreamReader lezer = new StreamReader(@"C:\VS2013\provinces.txt"); (2)
            int oppervlakte = -1;
            string regel;
            while ((regel = lezer.ReadLine()) != null) (3)
            {
                int dubbelPuntPos = regel.IndexOf(':'); (4)
                string provincie = regel.Substring(0, dubbelPuntPos); (5)
                if (provincie == provincieNaam) (6)
                    oppervlakte = int.Parse(regel.Substring(dubbelPuntPos + 1));
            }
            lezer.Close(); (7)
            if (oppervlakte == -1)
                throw new Exception("Onbestaande provincie:" + provincieNaam); (8)
            else
                return oppervlakte; (9)
        }
    }
}
```

```
    }  
}
```

- (1) Je importeert de namespace *System.IO* die de class *StreamReader* bevat.
- (2) Je maakt een *StreamReader* object en geeft aan de constructor het absolute pad mee van het bestand dat je wil lezen.
- (3) Je leest in een iteratie regel per regel uit het bestand.
Met de method *ReadLine()* lees je de volgende regel uit het bestand. Je onthoudt deze regel in de variabele *regel*. Als er geen volgende regel meer is (einde bestand), geeft de method *ReadLine()* de waarde **null** terug.
- (4) Je zoekt op de hoeveelste positie het teken : in de regel voorkomt.
- (5) Alle letters vóór het teken : vormen de provincienaam.
- (6) Als de provincienaam van de gelezen regel gelijk is aan de op te zoeken provincie, onthoud je de oppervlakte van deze provincie. Deze oppervlakte vind je in de cijfers ná het teken :.
- (7) Je sluit het bestand (een resource) explicet zodra je het bestand niet meer nodig hebt. Anders zou het bestand nodoeloos blijven openstaan tot de garbage collector het *StreamReader* object uit het geheugen verwijdert.
- (8) Als de op te zoeken provincie niet voorkomt in het bestand,werp je een exception naar degene die de method *ProvincieGrootte()* oproept.
- (9) Je geeft de oppervlakte van de provincie als resultaatwaarde terug.

Je kan deze nieuwe class uitproberen in het hoofdprogramma:

```
using System;  
namespace CSharpPFCursus  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.Write("Provincie:");  
            string provincie = Console.ReadLine();  
            try  
            {  
                ProvincieInfo info = new ProvincieInfo();  
                Console.WriteLine(info.ProvincieGrootte(provincie));  
            }  
            catch (Exception ex)  
            {  
                Console.WriteLine(ex.Message);  
            }  
        }  
    }  
}
```

Je kan de code van de method *ProvincieGrootte()* optimaliseren. Zodra je in het tekstbestand de provincie gevonden hebt, moet je de volgende regels van het bestand niet meer lezen:

```
using System;
using System.IO;
namespace CSharpPFCursus
{
    public class ProvincieInfo
    {
        public int ProvincieGrootte(string provincieNaam)
        {
            StreamReader lezer = new StreamReader(@"C:\VS2013\provincies.txt");
            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));
            }
            lezer.Close();
            throw new Exception("Onbestaande provincie:" + provincieNaam);
        }
    }
}
```

Zoals de code van de method *ProvincieGrootte()* nu geschreven is, wordt het bestand echter niet meer gesloten als de provincie gevonden wordt! Het **return** statement verlaat de method en voert de code na de **while** niet uit.

Je zou natuurlijk de code kunnen aanpassen: juist vóór het **return** statement schrijf je nog eens de opdracht om het bestand te sluiten: `lezer.Close();`.

Maar ook andere situaties kunnen er voor zorgen dat het bestand niet gesloten wordt. Als het bestand een tikfout (bvb. een letter) bevat in de oppervlakte van de provincie, veroorzaakt de method *Parse()* een exception. Ook bij een exception wordt de method *ProvincieGrootte()* verlaten zonder het bestand te sluiten.

Je zou dit ook kunnen oplossen door de code te omsluiten met **try - catch** opdrachten, en in de **catch** opdracht het bestand sluiten.

Maar uiteindelijk zou je tot code komen waarin de opdracht waarmee je het bestand sluit, meerdere keren voorkomt, wat niet bevorderlijk is voor de leesbaarheid en de onderhoudbaarheid van deze code.

Een betere oplossing is de code die het bestand gebruikt, te omsluiten met een **try** opdracht. Na deze try opdracht schrijf je een **finally** opdracht met accolades. Alle code die je binnen deze accolades schrijft, wordt 100% gegarandeerd uitgevoerd vanaf het moment dat je de bijbehorende **try** binnengekomen bent.

C# voert de **finally** ook uit als er binnen het **try** blok een **return** opdracht uitgevoerd wordt, of als er binnen het **try** blok een exception gebeurt.

```
using System.IO;
namespace CSharpPFCursus
{
    public class ProvincieInfo
    {
        public int ProvincieGrootte(string provincieNaam)
        {
            StreamReader lezer = new StreamReader(@"C:\VS2013\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);
        }
    }
}
```

- (1) Je verlaat de method. Toch wordt eerst de code van het **finally** blok uitgevoerd. Je bekomt dus 100% garantie dat het bestand gesloten wordt.
- (2) De code in dit **finally** blok wordt zeker uitgevoerd zodra je het bijbehorende **try** blok binnentkomt.

Je kan de code nog korter schrijven met de **using** opdracht. Je past deze opdracht toe bij het aanmaken van een object dat je achteraf wil sluiten. In ons geval is dit het *StreamReader* object. Na deze opdracht vermeld je accolades. Tussen de accolades gebruik je het object:

```
using System;
using System.IO;
namespace CSharpPFCursus
{
    public class ProvincieInfo
    {
        public int ProvincieGrootte(string provincieNaam)
        {
            using (StreamReader lezer = new
                StreamReader(@"C:\VS2013\provincies.txt"))
```

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

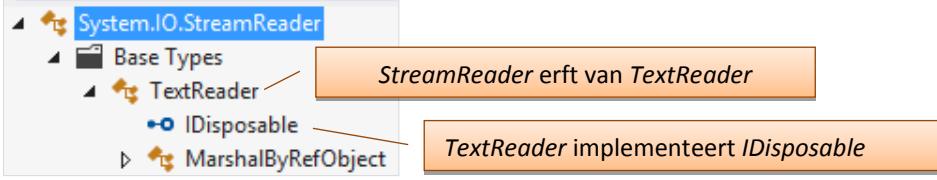
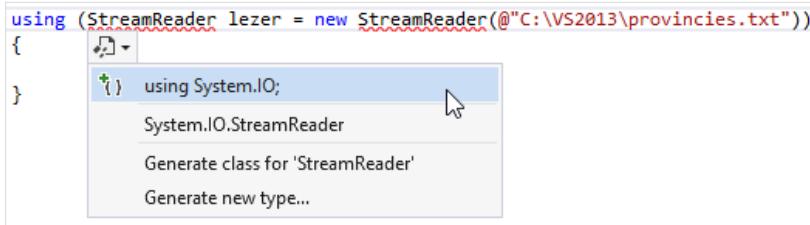
Zodra je het te sluiten object op deze manier in een `using` opdracht geplaatst hebt, sluit deze `using` opdracht zelf het object. Jij hoeft het object niet meer zelf te sluiten in de code.

De `using` opdracht sluit het bijbehorende object bij de sluitaccoorde van de `using` opdracht, maar ook als je binnen de `using` accolades de `return` opdracht toepast, of als er binnen de `using` opdracht exceptions optreden.

De **using** opdracht is dus een korte manier om gegarandeerde tijdige opkuis van resources te bekomen.

Opmerkingen:

	<p>using is een verkorte schrijfwijze van try - finally. Je kan met using geen catch verbinden. Als je foutopvang met catch én finally nodig hebt, zal je dus geen using kunnen gebruiken. Dan zal je toch try - catch - finally moeten toepassen.</p>
	<p>Je kan using enkel gebruiken voor objecten die de interface <i>IDisposable</i> implementeren. Je kan met de volgende stappen nagaan of de class <i>StreamReader</i> de interface <i>IDisposable</i> implementeert:</p> <ul style="list-style-type: none">• Kies in het menu <i>VIEW - Object Browser</i>.• Je ziet een venster met de titel <i>Object Browser</i>.• Tik bovenaan in het tekstvak <i><Search></i> het woord <i>StreamReader</i> en druk op <i>Enter</i>.• Klik op het pijltje vóór <i>System.IO.StreamReader</i>.• Klik op het pijltje vóór <i>Base Types</i>. Je ziet dat de class <i>StreamReader</i> van de class <i>TextReader</i> erft.• Klik op het pijltje vóór <i>TextReader</i>. Je ziet dat de class <i>TextReader</i> de interface <i>IDisposable</i> implementeert. Gezien de class <i>StreamReader</i> erft van <i>TextReader</i>, implementeert <i>StreamReader</i> ook de interface <i>IDisposable</i>.

	 <pre> graph TD StreamReader[System.IO.StreamReader] --> BaseTypes[Base Types] BaseTypes --> TextReader[TextReader] TextReader --> IDisposable[IDisposable] TextReader --> StreamReader </pre> <p><i>StreamReader erft van TextReader</i></p> <p><i>TextReader implementeert IDisposable</i></p>
	<p>Je gebruikt in C# het sleutelwoord using in twee totaal verschillende situaties:</p> <ul style="list-style-type: none"> • gegarandeerde tijdige opkuis van resources. • importeren van een namespace.
	<p>Je kan in VS op een eenvoudige manier een namespace importeren als dit nog niet gebeurd is. Om dit uit te proberen verwijder je eerst de coderegel using System.IO;.</p> <p>Je ziet nu dat StreamReader rood onderlijnd is en een foutmelding toont:</p> <pre> using (StreamReader lezer = new StreamReader(@"C:\VS2013\provincies.txt")) { The type or namespace name 'StreamReader' could not be found (are you missing a using directive or an assembly reference?) } </pre> <p>Je kan op twee manieren de namespace importeren:</p> <ul style="list-style-type: none"> • Klik met de muis in het woord StreamReader en vervolgens op het blauwe handvatje om het menu te openen en kies <i>using System.IO;</i>  <p>of</p> <ul style="list-style-type: none"> • Klik met de muis in het woord StreamReader en druk vervolgens de toetsencombinatie Ctrl ; (<i>control puntkomma</i>) en daarna de <i>Enter</i>-toets.



oefeningen: Exceptions

29 Operator overloading

29.1 Algemeen

Met operator overloading kan je een eigen betekenis geven aan de *operatoren* van C# (+, -, ==, >, >=, ...) wanneer je deze operatoren wil toepassen voor objecten van een eigen geschreven class.

Als voorbeeld ontwerp je een class *Breuk* die een wiskundige breuk voorstelt, met een teller en een noemer.

In deze class geef je onder andere betekenis aan de *productoperator* (*).

Als je * toepast op twee *Breuk* objecten, geeft dit een nieuw *Breuk* object terug, dat het product is van de twee *Breuk* objecten waarop de * operator is toegepast.

Het gebruik van deze operator is als volgt:

```
Breuk breuk1=new Breuk(1,2); // teller=1, noemer=2, dus 1/2
```

```
Breuk breuk2=new Breuk(1,3); // teller=1, noemer=3, dus 1/3
```

```
Breuk breuk3=breuk1*breuk2; // breuk3 wordt 1/2*1/3=1/6
```

	<ul style="list-style-type: none">Operator overloading is niet bij iedere class even interessant. Welke betekenis kan je bijvoorbeeld geven aan het optellen van twee werknemers?De lijst met overloadbare operatoren is: +, -, *, /, ~, ^, &, , %, !, <<, >>, ==, !=, >, <, >=, <=, ++, --, false en true.
---	---

29.2 De class Breuk

Voeg aan het project *CSharpPFCursus* de class *Breuk* (*Breuk.cs*) toe.

Eerst voeg je basis properties en basis methods toe aan deze class.

Daarna werk je operator overloading in deze class uit.

```
using System;
namespace CSharpPFCursus
{
    public class Breuk
    {
        int tellerValue;
        int noemerValue;

        public int Teller
        {
            get
                (1)
```

```

        {
            return tellerValue;
        }
        set
        {
            tellerValue = value;
        }
    }
    public int Noemer
    {
        get
        {
            return noemerValue;
        }
        set
        {
            if (value == 0)
                throw new Exception("Noemer mag niet nul zijn.");
            noemerValue = value;
        }
    }
}

public Breuk(int teller, int noemer)
{
    Teller = teller;
    Noemer = noemer;
}

public override string ToString()
{
    return Teller + "/" + Noemer;
}

public override bool Equals(object obj)
{
    if (obj is Breuk)
    {
        Breuk andereBreuk = (Breuk)obj;
        return (decimal)Teller / Noemer ==
               (decimal)andereBreuk.Teller / andereBreuk.Noemer;
    }
    else
        return false;
}

public override int GetHashCode()
{
    return Teller + Noemer;
}
}

```

(2)

(3)

(4)

(5)

(6)

- (1) Een property voor de teller van de breuk.
- (2) Een property voor de noemer van de breuk.
- (3) Een constructor met parameters voor de teller en de noemer van de breuk.
- (4) Je overschrijft de *ToString()* method van de base class (*Object*).
Je geeft een **string** terug met de tekstuele voorstelling van een breuk.
- (5) Je overschrijft de *Equals()* method van de base class (*Object*).
In deze method geef je **true** terug als de huidige breuk gelijk is aan het object dat als parameter *obj* binnentkomt. Anders geef je **false** terug. Twee breuken zijn gelijk als de deling (met decimalen) van teller door noemer van de ene breuk gelijk is aan de deling van teller door noemer van de andere breuk.
- (6) Als je de *Equals()* method overschrijft, moet je ook de *GetHashCode()* method overschrijven. De *GetHashCode()* verwacht dat je een **int** teruggeeft. Deze **int** moet hetzelfde zijn voor twee *Breuk* objecten met dezelfde inhoud. Dit wil zeggen: twee *Breuk* objecten met dezelfde teller en dezelfde noemer moeten dezelfde **int** teruggeven. Daartoe tel je de Teller op bij de Noemer. Een andere oplossing is ook mogelijk.

Je kan deze basisfunctionaliteit uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Breuk breuk2 = new Breuk(1, 3);
            Breuk breuk3 = new Breuk(2, 6);
            Console.WriteLine(breuk1.ToString());
            Console.WriteLine(breuk2.Equals(breuk3));
        }
    }
}
```

29.3 De vergelijkingsoperatoren

Je werkt de vergelijkingsoperatoren (==, !=, >, >=, <, <=) uit als methods binnen de class *Breuk*. Elk van deze methods moet de volgende eigenschappen hebben:

- De method moet **static** zijn (method met class bereik).
- De naam van de method moet het sleutelwoord **operator** zijn, gevolgd door de *vergelijkingsoperator* die deze method definiert.
- De method krijgt twee parameters binnen. De eerste parameter stelt de waarde voor die vóór de vergelijkingsoperator staat. De tweede parameter stelt de waarde voor die na de vergelijkingsoperator staat. Als je de operator **>** definieert, stelt de eerste parameter de waarde vóór **>** voor. De tweede parameter stelt de waarde na **>** voor.

- De method moeten een **bool** waarde teruggeven. Deze waarde moet de **bool** waarde voorstellen die de operator teruggeeft als je hem toepast op de twee parameters. Als je de operator **>** definieert, moet de method **true** teruggeven als de eerste parameter groter is dan de tweede parameter, anders moet de method **false** teruggeven.

Als voorbeeld werk je de operatoren **==** en **!=** uit in de class **Breuk**:

```
namespace CSharpPFCursus
{
    public class Breuk
    {
        ...
        public static bool operator ==(Breuk eerste, Breuk tweede)
        {
            return eerste.Equals(tweede); (1)
        }
        public static bool operator !=(Breuk eerste, Breuk tweede)
        {
            return !eerste.Equals(tweede); (2)
        }
    }
}
```

- (1) Om de **==** operator uit te werken kan je de **Equals()** method gebruiken. Deze heeft dezelfde betekenis als de **==** operator: twee objecten testen op *gelijkheid*.
- (2) Om de **!=** operator uit te werken pas je de **!** (*not*) operator toe op de **Equals()** method. Je draait de uitkomst van de **Equals()** method om: **true** wordt **false**, **false** wordt **true**.

Je kan bvb. de **==** operator uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Breuk breuk2 = new Breuk(1, 3);
            Breuk breuk3 = new Breuk(2, 6);
            Console.WriteLine(breuk1 == breuk2);
            Console.WriteLine(breuk2 == breuk3);
        }
    }
}
```

29.4 De wiskundige operatoren met twee waarden

Je werkt de wiskundige operatoren met twee waarden ($a+b$, $a-b$, $a*b$, a/b) uit als methods binnen de class *Breuk*. Elk van deze methods moet de volgende eigenschappen hebben:

- De method moet **static** zijn (method met class bereik).
- De naam van de method moet het sleutelwoord **operator** zijn, gevolgd door de wiskundige operator die deze method definieert.
- De method krijgt twee parameters binnen. De eerste parameter stelt de waarde voor die vóór de wiskundige operator staat. De tweede parameter stelt de waarde voor die na de wiskundige operator staat. Als je de operator * definieert, stelt de eerste parameter de waarde vóór * voor. De tweede parameter stelt de waarde na * voor.
- De method moeten een waarde teruggeven met als type de class waarin je de method schrijft. Deze waarde moet de waarde voorstellen die de operator teruggeeft als je hem toepast op de twee parameters. Als je de operator * definieert, moet de method een waarde teruggeven die het product voorstelt van de eerste en tweede parameter.

Als voorbeeld werk je de operator * uit in de class *Breuk*:

```
namespace CSharpPFCursus
{
    public class Breuk
    {
        ...
        public static Breuk operator *(Breuk eerste, Breuk tweede)
        {
            return new Breuk(
                eerste.Teller * tweede.Teller, eerste.Noemer * tweede.Noemer);      (1)
        }
    }
}
```

- (1) Je maakt een nieuw *Breuk* object dat het product voorstelt van het *Breuk* object *eerste* en het *Breuk* object *tweede*. De teller van het nieuwe *Breuk* object is de vermenigvuldiging van de *tellers* van de *Breuk* objecten *eerste* en *tweede*. De noemer van het nieuwe *Breuk* object is de vermenigvuldiging van de *noemers* van de *Breuk* objecten *eerste* en *tweede*. Je geeft dit nieuwe *Breuk* object terug als resultaatwaarde van de method.

Je kan de * operator uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Breuk breuk2 = new Breuk(1, 3);
            Console.WriteLine(breuk1 * breuk2);
        }
    }
}
```

}

Eén van de twee parameters van de method die een wiskundige operator met twee parameters overloadt, mag een ander type hebben dan de class waarin je de method uitschrijft.

Als voorbeeld maak je naast de eerste versie van de operator *, een tweede versie die als tweede parameter een `int` binnenkrijgt. De bedoeling is dat je met deze overloaded method een *Breuk* object kan vermenigvuldigen met een `int` en dat deze bewerking een nieuw *Breuk* object teruggeeft:

```
namespace CSharpPFCursus
{
    public class Breuk
    {
        ...
        public static Breuk operator *(Breuk breuk, int waarde)
        {
            return new Breuk(breuk.Teller * waarde, breuk.Noemer);           (1)
        }
    }
}
```

- (1) Je maakt een nieuw *Breuk* object dat het product voorstelt van het *Breuk* object *breuk* en het getal *waarde*. De teller van het nieuwe *Breuk* object is de vermenigvuldiging van de teller van het *Breuk* object *breuk* en *waarde*. De noemer van het nieuwe *Breuk* object is de *noemer* van het *Breuk* object *breuk*. Je geeft dit nieuwe *Breuk* object terug als resultaatwaarde van de method.

Je kan deze * operator uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Console.WriteLine(breuk1 * 3);
        }
    }
}
```

29.5 De operatoren die met één verhogen of verlagen (++,-)

Je werkt de operatoren ++ en -- uit als methods binnen de class *Breuk*. Elk van deze methods moet de volgende eigenschappen hebben:

- De method moet `static` zijn (method met class bereik).
- De naam van de method moet het sleutelwoord `operator` zijn, gevolgd door de operator die deze method defineert.

- De method krijgt één parameter binnen. Deze parameter heeft als type de class waarin je de method schrijft. Deze parameter stelt het object voor waarop je ++ of -- toepast.
- De method moet een waarde teruggeven met als type de class waarin je de method schrijft. Deze waarde moet de waarde voorstellen die de operator teruggeeft als je hem toepast op de parameter. Als je de operator ++ definieert, moet de method een waarde teruggeven die het object voorstelt dat met één is verhoogd.

Als voorbeeld werk je de operator ++ uit in de class Breuk:

```
using System;
namespace CSharpPFCursus
{
    public class Breuk
    {
        public static Breuk operator ++(Breuk breuk)
        {
            return new Breuk(breuk.Teller + breuk.Noemer, breuk.Noemer);      (1)
        }
    }
}
```

- (1) Je maakt een nieuw *Breuk* object dat het *Breuk* object *breuk* voorstelt dat met één is verhoogd. De *teller* van het nieuwe *Breuk* object is de som van de *teller* van het *Breuk* object *breuk* en de *noemer* van het *Breuk* object *breuk*. De *noemer* van het nieuwe *Breuk* object is de *noemer* van het *Breuk* object *breuk*. Je geeft dit nieuwe *Breuk* object terug als resultaatwaarde van de method.

Je kan de ++ operator uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Breuk breuk2 = breuk1++;                                (1)
            Console.WriteLine(breuk1);
            Console.WriteLine(breuk2);
        }
    }
}
```

- (1) Gezien je de ++ operator toepast ná de variabele *breuk1*, kent C# eerst de oorspronkelijke waarde van *breuk1* toe aan *breuk2*. Pas daarna verhoogt C# *breuk1* met één.

29.6 De verkorte operatoren (`+=`, `-=`, `*=`, `/=`, `%=`)

Je kan de verkorte operatoren niet overloaden, maar dit is ook niet nodig.

Als je overloading hebt toegepast van de bijbehorende wiskundige operatoren, kan je ook de verkorte operatoren voor je class objecten toepassen. Als je bijvoorbeeld operator overloading van de `*` operator hebt toegepast, kan je ook de `*=` operator gebruiken voor je class objecten.

Je kan dit uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(1, 2);
            Breuk breuk2 = new Breuk(1, 3);
            breuk1 *= breuk2;                                (1)
            Console.WriteLine(breuk1);
        }
    }
}
```

- (1) Gezien je overloading van de `*` operator hebt toegepast, kan je ook de `*=` operator toepassen voor `Breuk` objecten.

29.7 Conversie operatoren

Conversie operatoren zijn operatoren waarmee je objecten van je class naar een ander type kan omzetten. Je kan bijvoorbeeld een conversie operator schrijven om een breuk naar een `double` (decimaal getal) om te zetten.

Hierbij moet je onderscheid maken tussen `explicit` en `implicit` conversie operatoren.

Je voorziet een conversie operator van het sleutelwoord `explicit` als de conversie kan leiden tot een exception of informatieverlies. Als deze problemen zich niet kunnen voordoen, voorzie je een conversie operator van het sleutelwoord `implicit`.

Als je bvb. een breuk omzet naar een `double`, veroorzaakt dit geen exception of informatieverlies (% als `double` is 0,5). Deze conversie voorzie je van het sleutelwoord `implicit`.

Als je echter een breuk omzet naar een `int`, veroorzaakt dit mogelijk informatieverlies (% als `int` is 0). Deze conversie voorzie je van het sleutelwoord `explicit`.

Je werkt conversie operatoren uit als methods binnen de class `Breuk`. Elk van deze methods moet de volgende eigenschappen hebben:

- De method moet `static` zijn (method met class bereik).
- De method moet voorzien zijn van het sleutelwoord `explicit` of `implicit`.

- De naam van de method moet het sleutelwoord **operator** zijn, gevolgd door het type waarnaar je wil converteren.
- De method krijgt één parameter binnen. Deze parameter heeft als type de class waarin je de method schrijft. Deze parameter stelt het object voor dat je gaat converteren.
- De method moet een waarde teruggeven met als type het type waarnaar je het object, dat je als parameter binnenkrijgt, gaat converteren.

Je werkt de conversie operatoren naar **double** en naar **int** uit in de class Breuk:

```
using System;
namespace CSharpPFCursus
{
    public class Breuk
    {
        ...
        public static implicit operator double(Breuk breuk) (1)
        {
            return (double)breuk.Teller / (double)breuk.Noemer; (2)
        }
        public static explicit operator int(Breuk breuk) (3)
        {
            return breuk.Teller / breuk.Noemer; (4)
        }
    }
}
```

- (1) De conversie naar **double** veroorzaakt geen exception of informatieverlies. Daarom voorzie je deze conversie operator van het sleutelwoord **implicit**.
- (2) Je zet de teller en de noemer van de breuk om naar **double** en deelt daarna deze *doubles*. Het resultaat van deze deling (een **double**) is de conversie van de breuk naar **double**.
- (3) De conversie naar **int** veroorzaakt mogelijk informatieverlies. Daarom voorzie je deze conversie operator van het sleutelwoord **explicit**.
- (4) Je deelt de teller door de noemer van de breuk. Gezien deze beide van het type **int** zijn, is het resultaat ook een **int**, waarin cijfers na de komma zijn afgekapt. Het resultaat van deze deling is de conversie van de breuk naar **int**.

Je kan deze conversie operatoren uittesten met het volgende programma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(5, 2); (1)
            double dWaarde = breuk1;
            int iWaarde = (int)breuk1; (2)
            Console.WriteLine(dWaarde);
            Console.WriteLine(iWaarde);
        }
    }
}
```

```

        }
    }
}

```

- (1) Je converteert een breuk naar een **double**. Hierbij pas je een **implicit** conversie toe.
- (2) Je converteert een breuk naar een **int**. Hierbij pas je een **explicit** conversie toe. Bij een **explicit** conversie moet je bij het gebruik van de conversie het type waarnaar je converteert (**int**) tussen ronde haakjes vermelden vóór het object dat je converteert.

	<p>Opmerking: als je <i>nu</i> in een testprogramma een breuk object aanmaakt</p> <pre>Breuk breuk1=new Breuk(1,2);</pre> <p>en als volgt afbeeldt op de console:</p> <pre>Console.WriteLine(breuk1);</pre> <p>krijg je 0,5 en niet 1/2 .</p> <p>De <i>WriteLine()</i> method van <i>Console</i> geeft de voorkeur aan de conversie van het <i>Breuk</i> object naar double en het afbeelden van deze double t.o.v. het toepassen van de <i>ToString()</i> method van de class <i>Breuk</i>.</p> <p>Je kan de <i>Breuk</i> nog op de klassieke manier als volgt afbeelden:</p> <pre>Console.WriteLine(breuk1.ToString());</pre>
---	---

29.8 De operatoren true en false

De operatoren *true* en *false* zijn twee vrij ongewone operatoren. Ze kunnen namelijk niet expliciet gebruikt worden in code. De operatoren kunnen wel door de compiler worden opgeroepen als je een object gebruikt waar een boolean waarde verwacht wordt.

Een voorbeeld: stel, je moet in je programma dikwijls het onderscheid maken tussen echte breuken (waar de teller kleiner is dan de noemer en dus de waarde van de breuk tussen 0 en 1 ligt) en onechte breuken. Het zou dus handig zijn indien je kon schrijven:

```

if (breuk1)
    ...code voor echte breuk
else
    ...code voor onechte breuk

```

De compiler zal dan op zoek gaan naar een operator *true* van het object *Breuk*. Indien deze niet gedefinieerd is, krijg je een compileerfout.

Laten we deze operator *true* nu definiëren in de class *Breuk*:

```

public static bool operator true(Breuk breuk)                                (1)
{
    return breuk.Teller < breuk.Noemer;                                         (2)
}

```

- (1) Ook hier is de method *static*. Het resultaat van de operator *true* is een *bool*. De operator aanvaardt een *Breuk* object als parameter.

- (2) Je geeft *true* terug als de teller kleiner is dan de noemer en *false* indien niet.

De operatoren *true* en *false* vormen echter een onafscheidelijk duo. Ook deze laatste operator, *false*, moet je definiëren:

```
public static bool operator false(Breuk breuk) (1)
```

```
{ (2)
```

```
    return breuk.Teller >= breuk.Noemer;
```

```
}
```

- (1) De method is *static*. Het resultaat van de operator *false* is een *bool*. De operator aanvaardt een *Breuk* object als parameter.
(2) Je geeft *false* terug als de teller groter is dan of gelijk is aan de noemer en *true* indien niet.

Je kan nu terecht opmerken dat hier min of meer overbodige code staat. In ons voorbeeld is de test nog relatief eenvoudig maar deze kan ongetwijfeld in sommige gevallen veel ingewikkelder zijn. Daarom kan je ook het volgende schrijven:

```
public static bool operator false(Breuk breuk)
{
    return !breuk; (1)
}
```

- (1) Aangezien de operator *false* net het tegenovergestelde moet teruggeven van de operator *true*, gebruik je gewoon de *!* (*not*) operator.

Je moet in dit geval de *!* (*not*) operator eveneens overladen voor de class *Breuk*.

```
public static bool operator !(Breuk breuk)
{
    if (breuk)
        return false; (1)
    else
        return true;
}
```

- (1) We bepalen implicit met de operator *true* of het om een echte breuk gaat of niet en we retourneren het tegenovergestelde.

Je kan dit nu testen in het hoofdprogramma:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Breuk breuk1 = new Breuk(5, 2);
            if (breuk1)
```

```
        Console.WriteLine("Echte breuk");
    else
        Console.WriteLine("Onechte breuk");
    }
}
}
```

30 Indexers

30.1 Algemeen

Indexers in een class laten toe dat je data van objecten van deze class gebruikt zoals data van een array: deze data gedragen zich als een verzameling waarvan je één element kan opvragen.

Indexers zijn krachtiger dan gewone arrays.

Bij arrays spreken we één element uit de verzameling aan door het volgnummer (een **int**) van het element mee te geven.

Bij indexers kan je één element uit de verzameling ook aanspreken door een sleutelwaarde (bv. een **string**) mee te geven, die het element uniek identificeert in de verzameling.

Als voorbeeld maak je een class *Overuren*. Een object van het type *Overuren* houdt per maand bij hoeveel overuren je gepresteerd hebt.

Deze class bevat dus een verzameling van 12 elementen (overuren per maand).

Je zal één element op de klassieke manier kunnen aanspreken via zijn volgnummer:

```
Overuren mijnOveruren=new Overuren();
mijnOveruren[0]=4; // 1° maand 4 overuren gewerkt.
```

Daarnaast zal je één element kunnen aanspreken via de maandafkorting:

```
Overuren mijnOveruren=new Overuren();
mijnOveruren["jan"]=4; // 1° maand 4 overuren gewerkt.
```

Voeg de class *Overuren* (*Overuren.cs*) toe aan het project *CSharpPFCursus*.

De class ziet er als volgt uit:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSharpPFCursus
{
    public class Overuren
    {
        private int[] overurenValue = new int[12]; (1)
        private static readonly string[] maanden =
            {"jan", "feb", "maa", "apr", "mei", "jun",
             "jul", "aug", "sep", "okt", "nov", "dec"}; (2)

        public int this[int maand] (3)
        {
            get (4)
            {
                return overurenValue[maand]; (5)
            }
            set (6)
        }
    }
}
```

```

        {
            overurenValue[maand] = value; (7)
        }
    }

    public int this[string maand] (8)
    {
        get (9)
        {
            return overurenValue[WelkeMaand(maand)]; (10)
        }
        set (11)
        {
            overurenValue[WelkeMaand(maand)] = value; (12)
        }
    }

    private int WelkeMaand(string maand) (13)
    {
        int maandNr = Array.IndexOf(maanden, maand); (14)
        if (maandNr == -1)
            throw new IndexOutOfRangeException("Ongeldige maand:" + maand); (15)
        return maandNr; (16)
    }

    public int Totaal (17)
    {
        get
        {
            int totaal = 0;
            foreach (int overuur in overurenValue)
                totaal += overuur;
            return totaal;
        }
    }
}

```

- (1) Je maakt een **int** array om de overuren per maand bij te houden.
- (2) Je maakt een **string** array met de afkortingen van de maanden.
- (3) Met deze *indexer* zal je vanuit een testprogramma naar het aantal overuren van één maand refereren via het *volgnummer* van de maand (tussen 0 en 11).
- Je begint een *indexer* met het toegangsbereik (**public**, **private**, ...).
 - Daarna volgt het *type* van het element dat je met de *indexer* zal bereiken. Gezien het aantal overuren van één maand een **int** is, vermeld je hier **int**.
 - Daarna volgt het sleutelwoord **this**.
 - Daarna volgt tussen vierkante haakjes het type waarmee je het element identificeert in de verzameling. Hier vermeld je **int**. Daarmee bedoel je dat je één maand in de verzameling identificeert via een **int** (het volgnummer van de maand).

- Daarna vermeld je de naam van een parameter. Als het element geïdentificeerd wordt, bevat deze parameter de identificatie. Als in dit voorbeeld de maand met volgnummer 11 aangesproken wordt, bevat de parameter maand de waarde 11.
- (4) Het **get** gedeelte van een indexer dient om één element van de verzameling op te vragen. Dit is vergelijkbaar met het **get** gedeelte van een property.
- (5) Je haalt de waarde met het opgegeven volgnummer op uit de array met overuren en je geeft deze waarde terug als resultaatwaarde.
- (6) Het **set** gedeelte van een indexer dient om één element van de verzameling in te vullen. Dit is vergelijkbaar met het **set** gedeelte van een property.
- (7) Je neemt de in te vullen waarde (die je binnenkrijgt via het sleutelwoord **value**) over in het element met het opgegeven volgnummer.
- (8) Met deze indexer zal je vanuit een testprogramma het aantal overuren van één maand bereiken via de *afkorting van de maand* (vb. "jan"). Gezien deze afkorting een **string** is, zie je tussen de vierkante haakjes **string** maand.
- (9) Het **get** gedeelte van de indexer dient om één element van de verzameling op te vragen. Dit is vergelijkbaar met het **get** gedeelte van een property.
- (10) Eerst wordt het volgnummer van de opgegeven maandafkorting (de parameter *maand*) bepaald via de method *WelkeMaand(string maand)* (zie 13 t.e.m.16). De resultaatwaarde van deze method (een **int**) wordt dan als volgnummer gebruikt om het element met dit volgnummer op te zoeken in de array met overuren. Je geeft de waarde van dit element – de overuren van de opgegeven maand – als resultaatwaarde terug.
- (11) Het **set** gedeelte van deze indexer dient om één element van de verzameling in te vullen. Dit is vergelijkbaar met het **set** gedeelte van een property.
- (12) Je gebruikt dezelfde method *WelkeMaand(string maand)* (zie 13 t.e.m.16) om het volgnummer van de opgegeven maandafkorting (de parameter *maand*) te bepalen. Je plaatst de in te vullen waarde (die je binnenkrijgt via het sleutelwoord **value**) in de array met overuren in het element met het volgnummer dat je gevonden hebt via de method *WelkeMaand()*.
- (13) Met deze method wordt het volgnummer van de opgegeven maandafkorting (de parameter *maand*) bepaald en als een **int** teruggegeven.
- (14) Je zoekt het volgnummer van de opgegeven maandafkorting op in de array met alle maandafkortingen.
- (15) Als de opgegeven maandafkorting niet voorkomt in de array met maandafkortingen, w提醒p je een exception.
- (16) Als de opgegeven maandafkorting in de array met maandafkortingen gevonden wordt, geef je het volgnummer van de gevonden maandafkorting als resultaatwaarde terug.
- (17) Naast indexer(s) mag een class ook klassieke onderdelen (properties, methods, constructors, events, ...) bevatten.
Als voorbeeld maak je een readonly property *Totaal* waarin je het totaal aantal overuren (van alle maanden samen) teruggeeft.

Je kan deze class uittesten met het volgende programma:

```
using System;
namespace CSharpPFCursus
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Overuren mijnOveruren = new Overuren();  
            mijnOveruren[0] = 4;                                     (1)  
            mijnOveruren["apr"] = 2;                                 (2)  
            Console.WriteLine(mijnOveruren["jan"]);  
            Console.WriteLine(mijnOveruren[3]);  
            Console.WriteLine(mijnOveruren.Totaal);  
        }  
    }  
}
```

- (1) Je gebruikt de eerste *indexer* (met een **int** parameter) om in de eerste maand 4 overuren in te vullen. Je gebruikt een indexer zoals een array: je vermeldt tussen vierkante haakjes welk element uit de verzameling je wil aanspreken.
- (2) Je gebruikt de tweede *indexer* (met een **string** parameter) om in de vierde maand 2 overuren in te vullen.

31 Collections en Generics

31.1 Collections

31.1.1 Wat zijn collections?

Eerder in deze cursus werd het gebruik van arrays behandeld. Arrays mogen dan wel een nuttig instrument zijn, toch hebben ze hun beperkingen. Het vergt namelijk nogal wat kunst- en vliegwerk om een array te vergroten, elementen in te voegen of elementen te schrappen.

Er is echter een oplossing in de vorm van *Collections*. *Collections* is een verzamelnaam voor een aantal verschillende soorten rekbaar lijsten die wél een oplossing bieden voor de beperkingen van een array. *Collections* vinden we terug in de namespace *System.Collections*. Vergeet deze dus niet te importeren.

31.1.2 Soorten collections

Als we eens een kijkje nemen in de *System.Collections* namespace, vinden we een aantal verschillende types van collections: *ArrayList*, *BitArray*, *Hashtable*, *Queue*, *SortedList* en *Stack*. Een beschrijving van deze collections vind je in de onderstaande tabel:

ArrayList	Een rekbaar 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.

31.1.3 Collections gebruiken

Bij wijze van voorbeeld demonstreren we in deze paragraaf het gebruik van een *ArrayList*. Alle bovenstaande types collection uitgebreid bespreken zou ons te ver leiden.

De voornaamste methods en properties van een *ArrayList* zijn:

Count	Levert het aantal elementen van de lijst.
Add()	Voegt een element achteraan de lijst toe.
Clear()	Verwijdt 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 welbepaalte plaats.
LastIndexOf()	Zoekt het laatste voorkomen van een element in de lijst.
Remove()	Verwijderd een welbepaald element uit de lijst.
RemoveAt()	Verwijderd 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.

```

using Firma;
using Firma.Personeel;
using Firma.Materiaal;
using System;
using System.Collections;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Arbeider asterix = new Arbeider("Asterix",
                new DateTime(2014, 1, 1), Geslacht.Man, 24.79m, 3);
            Bediende obelix = new Bediende("Obelix",
                new DateTime(2014, 1, 1), Geslacht.Man, 2400.79m);
            Manager idefix = new Manager("Idefix",
                new DateTime(2014, 1, 1), Geslacht.Man, 2400.79m, 7000m); (1)

            ArrayList personeel = new ArrayList(); (2)
            personeel.Add(asterix);
            personeel.Add(obelix); (3)
            personeel.Insert(1, idefix); (4)

            Console.WriteLine(((Werknemer)personeel[0]).Naam +
                " is de 1ste van " + personeel.Count +
                " personeelsleden."); (5)

            foreach (Werknemer personeelslid in personeel) (6)
                Console.WriteLine(personeelslid.Naam);

            Afdeling eenAfdeling = new Afdeling("Verzending", 0); (7)
            personeel.Add(eenAfdeling);
            foreach (Werknemer personeelslid in personeel)
                Console.WriteLine(personeelslid.Naam);
        }
    }
}

```

}

- (1) Je maakt een arbeider, een bediende en een manager aan.
- (2) Je maakt een nieuwe *ArrayList personeel*.
- (3) Je voegt de arbeider en de bediende toe aan de arraylist.
- (4) Je voegt een manager in op positie 1 (Asterix blijft op positie 0, Idefix komt op positie 1 en Obelix schuift door naar positie 2).
- (5) We beelden Asterix af die op positie 0 staat. *ArrayList* ondersteunt directe toegang via een volgnummer. We moeten het object wel casten naar een *Werknemer*. Elementen van een *ArrayList* worden altijd opgeslagen als gewone objecten (van het type *Object*). Via de property *Count* tonen we het aantal elementen.
- (6) Met een foreach doorlopen we de *ArrayList*.
- (7) We maken een nieuwe afdeling aan en stoppen deze “per ongeluk” in de *ArrayList*. We proberen nadien de *ArrayList* te overlopen als een lijst van werknemers maar we krijgen een runtimefout.

Besluit:

- Een *ArrayList* is een zeer flexibele array met heel veel mogelijkheden. Nadeel is dat alle elementen in de *ArrayList* van het type *Object* zijn. Haal je een element uit de *ArrayList*, dan moet je dit eerst casten naar zijn echte type vooraleer je er de properties en methods van kan gebruiken.
- Je kan om het even wat in de *ArrayList* stoppen. De *ArrayList* controleert niet wat er ingestopt wordt. Dit kan fouten opleveren bij het casten, die pas optreden at runtime.

31.2 Generic lists

Het .NET Framework biedt (vanaf versie 2.0) een oplossing voor het probleem uit de vorige paragraaf, in de vorm van **generic lists**. De namespace *System.Collections.Generic* bevat een aantal classes zoals *List*, *Queue*, *SortedList* en *Stack*. In tegenstelling tot de gelijknamige classes in de namespace *System.Collections* bevatten deze collections elementen van een welbepaald type en niet van het algemene type *Object*.

Als voorbeeld passen we de code uit de vorige paragraaf aan. In plaats van een *ArrayList* gebruiken we deze keer een *List*. De syntax verschilt in die zin dat we deze keer opgeven welk type object er in de list terecht komt.

Verander de lijn code

```
ArrayList personeel = new ArrayList();
```

in

```
List<Werknemer> personeel = new List<Werknemer>();
```

Van zodra je deze lijn veranderd hebt, merk je dat de volgende lijn code een fout bevat:

```
personeel.Add(eenAfdeling);
```

De compiler merkt dat je probeert een afdeling toe te voegen aan de list, terwijl het gewenste type een Werknemer is. Verwijder de laatste 4 regels code waarin een afdeling wordt aangemaakt en toegevoegd (poging) en waar de lijst wordt overlopen.

Aangezien elk element in onze lijst van het type *Werknemer* is, is ook de cast-operatie niet langer nodig. Verander de lijn

```
Console.WriteLine(((Werknemer)personeel[0]).Naam +  
    " is de 1ste van " + personeel.Count + " personeelsleden.");
```

in

```
Console.WriteLine(personeel[0].Naam +  
    " is de 1ste van " + personeel.Count + " personeelsleden.");
```

32 Nullable types

32.1 Definitie

Als je in een programma een *lokale Value type* variabele, die je geen beginwaarde hebt gegeven, probeert te gebruiken, dan krijg je een compileerfout.

Value types zoals ints, bytes, longs, ..., chars, bools en DateTimes krijgen bij de declaratie wel een default waarde indien het membervariabelen zijn. Voor getallen is dit *0*, voor een bool *false* en voor een DateTime is dit *1/1/0001 00:00*.

Indien je nu bijvoorbeeld het aantal kinderen van een werknemer niet kent en dit aantal dus niet expliciet invult, dan zal de membervariabele *aantalKinderen* van een class Werknemer toch op de defaultwaarde 0 gezet worden. Wie deze waarde leest, kan dus geen onderscheid maken of het aantal kinderen onbekend of werkelijk 0 is.

In de databasewereld kan men de waarde van een veld op ‘onbekend’ of ‘niet ingevuld’ zetten. Ook in .NET kunnen we een Value type de waarde ‘onbekend’ geven. Je gebruikt hiervoor een zogenaamd *nullable type*.

Nullable types zijn extensies van de gewone Value types en je definieert ze als volgt:

```
byte? aantalKinderen;
```

```
bool? gehuwd;
```

Je kan niet alleen variabelen van nullable types maken, je kan ook arrays van nullable types maken, procedures schrijven die parameters hebben van een nullable type en functies maken die een resultaat hebben dat van een nullable type is.

32.2 Een nullable type gebruiken

Als je een *lokale* variabele van een nullable type aanmaakt en er geen beginwaarde aan toekent, dan krijg je een compileerfout wanneer je deze variabele verder in je programma probeert te gebruiken. Een *member* variabele van een nullable type krijgt default de waarde *null*.

Je kan variabelen van een nullable type op dezelfde manier een waarde geven als variabelen van een Value type:

```
aantalKinderen = 7;
```

Je kan de waarde van een variabele van een nullable type ook op *onbekend* zetten. Dit doe je met het sleutelwoord *null*:

```
aantalKinderen = null;
```

Om te controleren of een variabele van een nullable type een waarde bevat, gebruik je de property *HasValue*:

```

if (aantalKinderen.HasValue) // of if (aantalKinderen != null)
    Console.WriteLine("Er zijn {0} kinderen", aantalKinderen);
else
    Console.WriteLine("Het aantal kinderen is onbekend");

```

Je kan de waarde van een nullable type ook opvragen via de *Value* property:

```
Console.WriteLine("Er zijn {0} kinderen", aantalKinderen.Value);
```

32.3 De operator ??

Je kan de waarde van een *nullable* type ook toekennen aan een *Value type*. Maar wat als het nullable type de waarde *null* heeft?

Met de operator ?? kan je aangeven welke waarde aan het *Value type* wordt toegekend wanneer het nullable type op *null* staat:

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            byte? aantalKinderen = null;
            byte aantalKamers;
            aantalKamers = aantalKinderen ?? 0; (1)
            Console.WriteLine("Er zijn {0} kinderkamers nodig", aantalKamers);
        }
    }
}

```

- (1) Het *Value type* *aantalKamers* krijgt de waarde van het nullable type *aantalKinderen*, tenzij deze de waarde *null* heeft, dan wordt de waarde van het *Value type* *aantalKamers* 0.

33 Extra taalelementen

In dit hoofdstuk leer je een aantal taalelementen van C# kennen die vanaf versie 3.0 aan de programmeertaal toegevoegd werden. Deze extra taalelementen verhogen de productiviteit van de programmeur: met minder source code meer bereiken.

Niet alle nieuwe taalelementen voegen nieuwe functionaliteit toe aan C#, maar m.b.v. deze nieuwe features kunnen er korte en krachtige C# statements geschreven worden, waardoor de code beter leesbaar en eenvoudiger te onderhouden is.

De taalelementen zijn o.a. toegevoegd om het gebruik van *LINQ* (Language **I**ntegrated **Q**uery) binnen een .NET applicatie mogelijk te maken.

LINQ is een SQL-achtige technologie die je kan gebruiken wanneer je in een applicatie met (verzamelingen) *gegevens* wil werken. LINQ laat je toe om op een zeer krachtige en uniforme manier gegevens te gebruiken, ongeacht de bron van deze gegevens: een array, een List, een XML document, een relationele database, LINQ komt verder in het opleidingstraject aan bod.

33.1 type inference - Implicitly typed local variables

In het hoofdstuk LOKALE VARIABELEN heb je geleerd dat je lokale variabelen in C# op de volgende manier declareert en er een waarde aan toekent:

```
int aantalKinderen = 3;  
decimal wedde = 1500m;  
of  
int aantalKinderen;  
aantalKinderen = 3;  
decimal wedde;  
wedde = 1500m;
```

Je declareert dus een variabele door eerst het *type* van de variabele op te geven, gevolgd door een spatie, gevolgd door de *naam* van de variabele. Je kan meteen bij de declaratie of verder in de code een waarde toekennen aan deze variabele.

Het is echter ook mogelijk om een lokale variabele te declareren en er een initiële waarde aan toe te kennen **zonder een type op te geven** voor de variabele. Je doet dit met het sleutelwoord **var**. De *compiler* bepaalt het type van deze variabele op basis van de initiële waarde of expressie die aan de variabele toegekend werd.

Voorbeeld:

```
namespace CSharpPFCursus  
{  
    class Program  
    {  
        static void Main(string[] args)
```

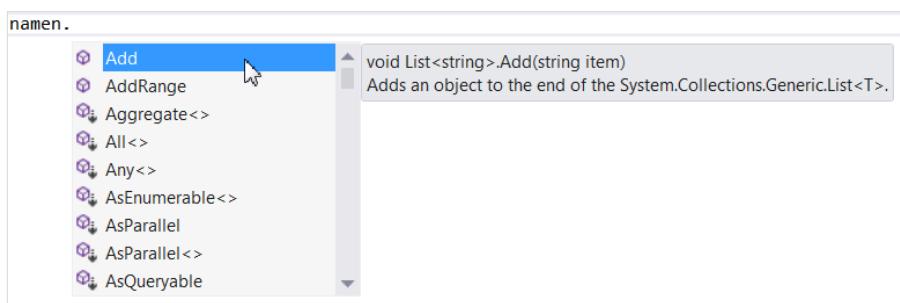
```

    {
        var aantalKinderen = 3; (1)
        var wedde = 1500m; (2)
        var namen = new List<string>(); (3)

        Console.WriteLine(aantalKinderen.GetType());
        Console.WriteLine(wedde.GetType());
        Console.WriteLine(namen.GetType());
    }
}
}

```

- (1) De compiler vervangt `var` door het juiste type op basis van de waarde 3: de variabele `aantalKinderen` krijgt het type `int` (`System.Int32`).
- (2) De compiler vervangt `var` door het juiste type op basis van de waarde `1500m`: de variabele `wedde` krijgt het type `decimal` (`System.Decimal`).
- (3) De compiler vervangt `var` door het type `System.Collections.Generic.List<T>`, waarbij T van het type `string` is. Als je nu de variabele `namen` verder in de code gebruikt, zijn alle properties en methods van de class `List` beschikbaar:



- Eénmaal de compiler aan een variabele een bepaald type toegekend heeft, kan dit type niet meer gewijzigd worden. Voorbeeld:

`var aantalKinderen = 3;`
`aantalKinderen = "drie";`

class System.String
Represents text as a series of Unicode characters.

Error:
Cannot implicitly convert type 'string' to 'int'

- Het sleutelwoord `var` kan enkel gebruikt worden bij de declaratie van **lokale** variabelen (vandaar *local variable type inference*). `var` 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 voor de parameters van een method, ...
- De initiële waarde bij *local type inference* mag niet `null` zijn:
`var woord = null; //geeft een compilerfout`
- Een variabele gedeclareerd met `var` moet een initiële waarde bevatten:
`var woord; //geeft een compilerfout`

Het sleutelwoord `var` kan ook gebruikt worden bij de initialisatie in een *for* constructie.

```
for(var i=1;i<11;i++)           de variabele i is van het type int
    Console.WriteLine(i);
```

```
for(var j=1.0;j<11;j++)         de variabele j is van het type double
    Console.WriteLine(j);
```

Het sleutelwoord `var` kan ook gebruikt worden bij de initialisatie in een *foreach* constructie.

```
var maanden = new string[] {"januari", "februari", "maart", "april",           (1)
    "mei", "juni", "juli", "augustus",
    "september", "oktober", "november", "december"};
foreach (var maand in maanden)          (2)
    Console.WriteLine(maand.ToUpper());
```

(1) De variabele *maanden* heeft als type een array van strings: `System.String[]`.

(2) Hierdoor krijgt de variabele *maand*, die gebruikt wordt om deze array te doorlopen, ook impliciet het type `string`. M.a.w. de variabele die gebruikt wordt om een array of een collectie van elementen te doorlopen, krijgt automatisch het type van de elementen van de array of collectie, als je deze variabele declareert met `var`.

	<p>Met het sleutelwoord <code>var</code> kan je ook het type van een <i>array implicit</i> declareren. Hierbij wordt het type van de array bepaald op basis van de elementen die opgegeven worden bij de initialisatie van de array. Er wordt dus geen type vermeld na het sleutelwoord <code>new</code>, enkel <code>[]</code>.</p> <p>Dit worden <i>Implicitly Typed Arrays</i> genoemd.</p> <pre>var getallen = new[] { 10, 20, 30 }; Console.WriteLine(getallen.GetType());</pre> <p>De variabele <i>getallen</i> is van het type <code>Int32[]</code>, een <i>array van integers</i>.</p> <p>Een voorwaarde is wel dat alle elementen tussen de <code>{ }</code> hetzelfde type hebben of dat er bij deze elementen minstens één type is waarnaar het type van alle andere elementen impliciet kan geconverteerd worden.</p> <pre>var werknemers = new[] {new Arbeider("Asterix", new DateTime(2009,1,1),Geslacht.Man,10m,1), new Bediende("Obelix",new DateTime(2009,2,1),Geslacht.Man,1500m)};</pre> <p>Dit voorbeeld geeft een compilerfout daar een <i>Arbeider</i> object niet impliciet kan geconverteerd worden naar een <i>Bediende</i> object of omgekeerd.</p> <pre>var werknemers = new[] {new Bediende("Asterix",new DateTime(2009,1,1),Geslacht.Man,1500m), new Manager("Obelix",new</pre>
---	---

```

        DateTime(2009,2,1),
        Geslacht.Man,1500m,1000m});
Console.WriteLine(woorknemers.GetType());

```

Deze code is correct: een *Manager* object kan impliciet geconverteerd worden naar een *Bediende* object daar de class *Manager* een afgeleide class is van de class *Bediende*. Het type van de array *woorknemers* is dan ook *Bediende[]*, m.a.w. *woorknemers* is een array van referenties naar *Bediende* objecten.

Het sleutelwoord **var** kan ook gebruikt worden in een *using* constructie (*using*: zie hoofdstuk EXCEPTIONS paragraaf GEGARANDEerde TIJDige OPKUIS VAN RESOURCES).

```

using (var lezer = new StreamReader(@"C:\VS2013\provinces.txt"))
{
    class System.IO.StreamReader
    Implements a System.IO.TextReader that reads characters from a byte stream in a particular encoding.
}

```

Hier krijgt de variabele *lezer* impliciet het type *System.IO.StreamReader*, daar een *StreamReader* object als initiële waarde toegekend wordt aan de variabele *lezer*.

33.2 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 om nieuwe methods aan een bestaand type toe te voegen wanneer het niet mogelijk is of niet de bedoeling is dat je de source code van de bestaande class wijzigt (bvb. een class van het .NET Framework of een class waarvan je zelf niet de eigenaar bent). Anderzijds kunnen er van een aantal classes, de zogenaamde *sealed* classes, geen nieuwe classes afgeleid worden. Indien je aan een dergelijke class nieuwe functionaliteit wil toevoegen, kan je hiervoor *extension methods* definiëren.

Extension methods zijn altijd *static* methods die in een *static* class gedefinieerd worden.

33.2.1 Extension methods definiëren

Als voorbeeld voegen we twee extension methods toe aan de class *String* van het .NET Framework. De class *String* is een *sealed* class, zodat we er geen nieuwe class van kunnen afleiden (zie documentatie).

- een method *ToUpperFirst()*
Deze method converteert de eerste letter van een opgegeven tekst naar een hoofdletter en de rest van de letters naar kleine letters.
- Een method *Right(n)*
Deze method geeft een deel van de string, nl. *n* aantal tekens, te beginnen vanaf rechts.

We definiëren deze methods in een nieuwe class *MyStringExtensions*.

Creëer een *static* class *MyStringExtensions* (*MyStringExtensions.cs*) in het project *CSharpPFCursus*. Voeg de static methods *ToUpperFirst()* en *Right()* toe aan deze class.

```
namespace CSharpPFCursus
{
    public static class MyStringExtensions
    {
        public static string ToUpperFirst(this string s)
        {
            return char.ToUpper(s[0]) + s.Substring(1);
        }

        public static string Right(this string s, int aantal)
        {
            if (s.Length <= aantal)
                return s;
            return s.Substring(s.Length - aantal);
        }
    }
}
```

(1) Extension methods worden gedefinieerd in een *static* class.

(2) Extension methods zijn *static*.

De parameterlijst van een extension method begint met het sleutelwoord *this*. Dit sleutelwoord komt nog vóór de eerste parameter, hier de parameter *string s*. Met het sleutelwoord *this* wordt aangegeven dat de extension method van toepassing is op het type dat er juist achter staat. De method *ToUpperFirst(this string s)* is dus een extension method voor de class *String* (*string* is een alias naam voor de class *String*). M.a.w., deze method kan uitgevoerd worden voor elk object van het type *string*.

De resultaatwaarde van deze method is in dit geval ook van het type *string*, maar dit is geen noodzakelijke voorwaarde.

(3) De extension method *Right(this string s, int aantal)* kan eveneens uitgevoerd worden voor elk object van het type *string* (*this string s*). Via een tweede parameter (*int aantal*) kan je opgeven hoeveel tekens uit de string teruggegeven worden.

De resultaatwaarde van deze method is ook van het type *string*.

33.2.2 Extension methods gebruiken

Je kan deze extension methods uitproberen in de class *Program.cs*:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            var land = "belgië";
            Console.WriteLine(land.ToUpperFirst());          (1)
            Console.WriteLine(land.Right(3));                (2)
            Console.WriteLine(land.Right(3));                (3)
        }
    }
}
```

```

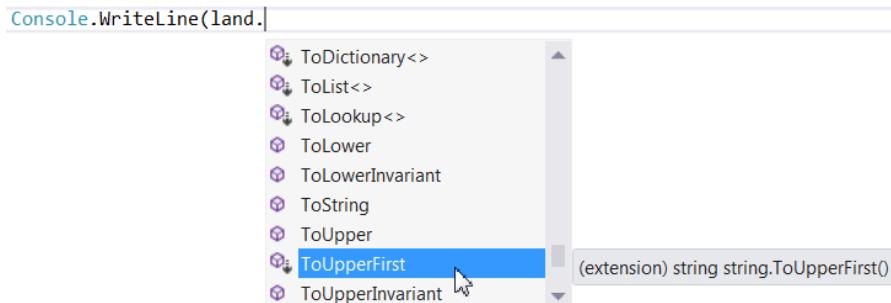
    }
}

}

```

- (1) De variabele *land* is van het type **string** (local variable type inference).
- (2) Hierdoor kan de extension method *ToUpperFirst()* toegepast worden op dit string object. Merk op dat je geen parameterwaarde hoeft mee te geven voor deze method. De extension method wordt opgeroepen d.m.v. de variabele *Land* die naar een string object verwijst. De method gebruikt dit string object als eerste parameter en wordt dus uitgevoerd voor dit string object.
- (3) Hierdoor kan de extension method *Right()* eveneens toegepast worden op dit string object. Voor de tweede parameter (*aantal*) moet er uiteraard wel een waarde voorzien worden.

- Bij het intikken van *land.* verschijnt er een popup venster met alle beschikbare members van de class *String*. Merk op dat de extension methods eveneens beschikbaar zijn. Je herkent ze aan het pijltje bij het symbool:



- In het hoofdstuk STATIC MEMBERS heb je geleerd dat je *static* methods van een class kan oproepen en uitvoeren zonder eerst een object van deze class te creëren. Je roept de static methods op via de class zelf. In het geval van een static class kan je zelfs geen instantie van deze class creëren. Extension methods daarentegen roep je wel op d.m.v. een object van het type waarvoor de extension method gedefinieerd is. Achter de schermen genereert de compiler echter een static method aanroep.
- Het is verstandig om extension methods logisch te groeperen in een aparte namespace zodat je weet waar ze te vinden zijn en er geen conflicten ontstaan door dubbele naamgeving of functionaliteit. Uiteraard moet je dan rekening houden met de bereikbaarheid van de extension methods. Als de static class, die de extension methods bevat, in een andere namespace gedefinieerd is dan de namespace waarin je de extension methods wil gebruiken, moet je deze namespace d.m.v. het sleutelwoord **using** importeren. Hierdoor worden alle extension methods van alle static classes van de geïmporteerde namespace beschikbaar.
- Als je een method oproeft via een object variabele, wordt er wel eerst nagegaan of er voor dit type object een instance method met dezelfde naam en signature is. Zo ja, dan wordt deze method uitgevoerd. Zo neen, dan wordt de overeenkomende extension method uitgevoerd. M.a.w. de instance methods krijgen voorrang op de extension methods.

Extensions methods worden bij LINQ gebruikt. Microsoft heeft in de namespace *System.Linq* een framework van extension methods voorzien, die overal door de CLR kunnen gebruikt worden. De methods die in deze namespace gedefinieerd zijn, zijn bedoeld om met (verzamelingen) gegevens te werken. Je kan deze methods toepassen op XML objecten, relationele database objecten en .NET objecten die de interface *IEnumerable* of de generic versie ervan *IEnumerable<T>* implementeren.

33.3 Automatic properties – Auto-Implemented properties

Belangrijke principes bij objectgeoriënteerd programmeren zijn *data hiding* en *encapsulation*. Bij het ontwerp van een class kan de programmeur deze principes nastreven door de data van de class in te kapselen (private te houden) en enkel toegankelijk te maken voor de buitenwereld via public methods.

In .NET worden de data van de class gedefinieerd d.m.v. properties. Door properties te gebruiken in plaats van public variabelen, kan je onbeveiligde, ongecontroleerde en niet-geverifieerde toegang tot de objectgegevens voorkomen.

In het hoofdstuk CLASSES, OBJECTS EN OBJECT VARIABELEN heb je geleerd dat iedere property uit drie onderdelen bestaat:

- Een **private** variabele waarin je de waarde van de property bishoudt. Enkel methods binnen de class kunnen deze variabele aanspreken.
- Een beveiligde toegangsweg **set** waarmee een programmeur de waarde van de property kan invullen. Deze set routine bevat vaak code om de waarde, die de programmeur wil invullen, te valideren. C# roept de **set** routine automatisch op als een programmeur de waarde van de property probeert te wijzigen.
- Een toegangsweg **get** waarmee een programmeur de waarde van de property (dus van de **private** variabele) kan opvragen. C# roept de **get** routine automatisch op als de programmeur de waarde van de property opvraagt.

De **set** en de **get** zijn dus beveiligde toegangswegen naar de **private** variabele die de waarde van de property bishoudt.

Vanaf C# 3.0 kan je met een minimum aan tikwerk properties definiëren, die een minimale functionaliteit bevatten. Deze properties bevatten enkel een **get** en een **set**, zonder code. Dit zijn zogenaamde **automatic properties** of **auto-implemented properties**.

D.m.v. automatic properties kan je op een zeer beknopte manier properties voorzien in een class, wanneer de **get** en de **set** van deze properties geen extra code moeten bevatten.

Voorbeeld:

```
namespace CSharpPFCursus
{
    public class Klant
    {
        public string Naam { get; set; } (1)
```

```
public int Klantnummer { get; private set; } (2)
}
}
```

- (1) Een *automatic property Naam*. De *private variabele* die bij deze property hoort, zie je niet, maar wordt door de *compiler* automatisch aangemaakt bij het gebruik van deze property via de *get* en de *set* accessors. Deze private variabele kan ook alleen maar via deze accessors aangesproken worden en is ontoegankelijk voor de programmeur.
- (2) Een *readonly automatic property Klantnummer*. Automatic properties moeten per definitie een *set* en een *get* accessor bevatten. Dus een readonly automatic property kan je niet definiëren door enkel een *get* routine te voorzien (cfr. een gewone readonly property). Je kan een automatic property wel readonly maken door de *set* te voorzien van het sleutelwoord *private*.

De programmeur kan een automatic property op dezelfde manier aanspreken als een gewone property:

```
Klant eenKlant = new Klant();
eenKlant.Naam = "Jan";
Console.WriteLine(eenKlant.Naam);
```

	<p>De VS.NET ontwikkelomgeving helpt je bij het schrijven van van automatic properties. Zo kan je bvb. op een snelle manier een automatic property <i>Naam</i> aan een class toevoegen:</p> <ul style="list-style-type: none"> • positioneer de cursor in de code van de class waar je de property wil toevoegen en tik de shortcut <i>prop</i> en druk vervolgens tweemaal op de Tab-toets of • klik met de rechtermuisknop in de code van de class waar je de property wil toevoegen en kies <i>Insert Snippet...</i> kies vervolgens <i>Visual C#</i> en verder <i>prop</i> <p>Je ziet nu de volgende code:</p> <pre>public int MyProperty { get; set; }</pre> <ul style="list-style-type: none"> • tik het juiste type (<i>string</i>) van de property en druk op de Tab-toets • tik de naam (Naam) van de property en druk op de Tab-toets
---	---

	<ul style="list-style-type: none"> • Je kan een automatic property later herschrijven als een gewone property, door bvb. validatiecode toe te voegen aan het <i>set</i> gedeelte en een private variabele te voorzien. Het achteraf wijzigen van deze property heeft echter geen gevolgen voor de programmeur die deze class property reeds gebruikt heeft. Deze programmeur hoeft in zijn code niets te wijzigen.
---	---

- Het is een goed idee om public data variabelen in een class te vervangen door automatic properties. Het is immers zo dat als je beslist om een public variabele later te herschrijven als een gewone property, de programmeur die deze public variabele reeds gebruikt heeft, waarschijnlijk zijn code ook moet wijzigen.

33.4 Object initializers

In de praktijk is het vaak zo dat de properties van een object reeds bij de creatie van dit nieuwe object een beginwaarde krijgen, m.a.w. dat objecten bij de creatie geïnitialiseerd worden.

In het hoofdstuk CONSTRUCTORS heb je geleerd dat je een object bij de creatie ervan kan initialiseren door gebruik te maken van geparametriseerde constructors. De beginwaarden van de properties komen als parameters van de constructor binnen. In een class kan je meerdere geparametriseerde constructors voorzien. Zij verschillen in het aantal en/of type van de parameters.

```
namespace CSharpPFCursus
{
    public class Klant
    {
        public Klant(int klantnummer, string naam)
        {
            this.Klantnummer = klantnummer;
            this.Naam = naam;
        }

        public int Klantnummer { get; private set; }
        public string Naam { get; set; }
    }
}
```

De programmeur kan bij de creatie van een *Klant* object het *klantnummer* en de *naam* meegeven:

```
Klant eenKlant = new Klant(1, "Jan");
```

D.m.v. **object initializers** is het vanaf C# 3.0 mogelijk om op een beknopte manier objecten bij hun creatie te initialiseren, zonder geparametriseerde constructors te voorzien. Een *object initializer* specificeert een waarde voor één of meerdere properties (of public variabelen) van het object dat gecreëerd wordt.

Hierdoor krijg je compacte en beter leesbare code.

Het is ook mogelijk om *geparametriseerde constructors* en *object initializers* te combineren.

Voorbeeld class *Persoon*:

```
namespace CSharpPFCursus
{
    public class Persoon
    {
```

```

public string Naam { get; set; } (1)
public int AantalKinderen { get; set; } (1)

public Persoon()
{
}

public Persoon(string naam) (3)

{
    Naam = naam;
}

public Persoon(string naam, int aantalKinderen) (3)
{
    Naam = naam;
    AantalKinderen = aantalKinderen;
}
}
```

- (1) Een automatic property.
- (2) De default constructor.
- (3) Een geparameerde constructor.

Objecten van deze class *Persoon* kunnen op verschillende manieren gecreëerd worden:

```

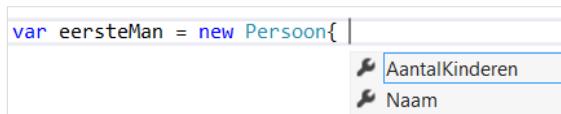
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            var eersteMan = new Persoon { Naam = "Adam",
                                         AantalKinderen = 2 }; (1)

            var eersteVrouw = new Persoon("Eva") { AantalKinderen = 2 }; (2)

            var eersteZoon = new Persoon { Naam = "Kaïn" }; (3)
            eersteZoon.AantalKinderen = 0;

        }
    }
}
```

- (1) Hier worden enkel *object initializers* gebruikt om een nieuw *Persoon* object te initialiseren: **ná** *new Persoon* **tik je een open accolade { en een spatie**. VS toont je alle properties die je op dit ogenblik kan initialiseren:



Geef een waarde op voor elke property die je wil initialiseren via een object initializer. De verschillende *object initializers* worden gescheiden door een *komma* en *afgesloten* met een *accolade* }.

Wanneer je enkel *object initializers* wil gebruiken om nieuwe objecten te initialiseren, hoeft er in de class geen constructor gedefinieerd te worden. De default constructor wordt wel uitgevoerd maar deze moet niet expliciet gedefinieerd zijn in de class, daar de compiler deze automatisch aanmaakt indien nodig.

Je kan dit uittesten door de drie constructors in de class *Persoon* even in commentaar te plaatsen.

Let op: indien de class een constructor met parameters bevat, moet de default constructor wel expliciet gedefinieerd zijn in deze class.

- (2) Hier wordt een combinatie gebruikt van een *geparametriseerde constructor* (voor de initialisatie van de property *Naam*) en een *object initializer* (voor de initialisatie van de property *AantalKinderen*).
De geparametriseerde constructor en ook een default constructor moeten gedefinieerd zijn in de class *Persoon*.
- (3) Hier wordt enkel de property *Naam* via een object initializer geïnitialiseerd bij de creatie van het *Persoon* object. Je bent dus niet verplicht om alle properties (op dezelfde manier) te initialiseren.

	<ul style="list-style-type: none">• Indien een combinatie van een constructor en object initializers gebruikt wordt voor de initialisatie van een object, wordt eerst de constructor uitgevoerd, daarna de object initializer.• Object initializers spelen een rol bij lambda expressions (zie verder).
--	--

33.5 Collection initializers

Bij de declaratie van een array kan je beginwaarden meegeven om de elementen van de array te initialiseren (zie hoofdstuk ARRAYS).

Het is ook mogelijk om bij de *creatie* van een nieuw *collection* object (uit de namespace *System.Collections* of *System.Collections.Generic*) beginwaarden mee te geven om de elementen van de nieuwe collection te initialiseren. Hiervoor gebruik je zogenaamde **collection initializers**. Deze beginwaarden kunnen een *waarde*, een *expressie* of een *object initializer* zijn. *Collection initializers* maken de code meer compact door de *creatie* en het *opvullen* van een collectie in één instructie te combineren. De programmeur hoeft de verschillende *Add* instructies niet meer te schrijven, de compiler neemt dit voor zijn rekening.

Voorbeeld:

```
namespace CSharpPFCursus
{
    class Program
    {
```

```

    static void Main(string[] args)
    {
        var getallen = new ArrayList {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; (1)
        var maanden = new List<string>{"januari", "februari", "maart", "april",
            "mei", "juni", "juli", "augustus", "september", "oktober", "november", "december"}; (1)
        var personen = new List<Persoon> {
            new Persoon{Naam="Adam", AantalKinderen=2},
            new Persoon {Naam="Eva", AantalKinderen=2}}; (2)
        }
    }
}

```

- (1) Je plaatst de beginwaarden voor de collectie tussen accolades {}, de verschillende elementen worden door een komma gescheiden.
- (2) Hier wordt een combinatie van een *collection initializer* en *object initializers* gebruikt. De *List personen* wordt via een *collection initializer* geïnitialiseerd met *Persoon* objecten ({}) na `new List<Persoon>`). Ieder nieuw *Persoon* object zelf wordt d.m.v. *object initializers* geïnitialiseerd ({} telkens na `new Persoon`).

33.6 Anonymous types

Tot nu toe heb je steeds objecten gecreëerd op basis van types of classes die ergens gedefinieerd zijn: een eigen gedefinieerde class of een class uit het .NET Framework.

Vanaf C# 3.0 kan je objecten creëren op basis van classes die niet expliciet gedefinieerd zijn. Deze niet expliciet gedefinieerde types hebben geen naam en worden daarom ***anonieme types*** of ***anonymous types*** genoemd.

Een *anonymous type* wordt implicit gedefinieerd tijdens de creatie van een object op basis van een verzameling properties, die in de vorm van object initializers bij de creatie van het object voorzien worden.

In de instructie

```
var persoon = new { Nr = 1, Naam = "Adam", AantalKinderen = 2};
```

verwijst de reference variabele persoon naar een object waarvan het type on-the-fly, tijdens de creatie van het object, gedefinieerd wordt.

Vermits er na het sleutelwoord `new` geen type of class vermeld is, heeft het type van het gecreëerde object geen naam. Eigenlijk wordt de naam van het anonieme type door de compiler zelf bepaald en kan/mag deze naam niet gebruikt worden in je code. Het anonieme type wordt afgeleid van de object initializers die bij de creatie van het object vermeld zijn: { `Nr = 1, Voornaam = "Adam", AantalKinderen = 2` }.

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)

```

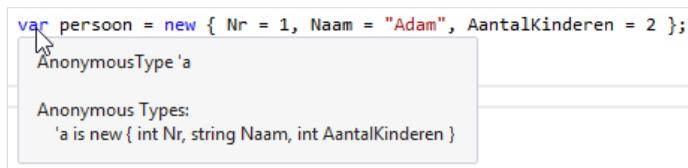
```

    {
        var persoon = new { Nr = 1, Naam = "Adam", AantalKinderen = 2};      (1)
        Console.WriteLine(persoon.GetType().ToString());
        Console.WriteLine(persoon.Naam);
    }
}
}

```

- (1) De variabele persoon heeft als type een *anonieme class* met de properties *Nr*, *Naam* en *AantalKinderen*.

Merk op dat het sleutelwoord **var** hier interessant is: bij de declaratie van een variabele met **var** hoef je immers geen type op te geven, het type wordt afgeleid van de initiële waarde die aan de variabele toegekend wordt (*local type inference*):



- Anonieme types erven rechtstreeks van de class *Object*.
- In C# zijn de properties van een object van een anoniem type *readonly*. `persoon.Naam = "Eva";` geeft een compilerfout.
- Twee anonymous types zijn hetzelfde als hun properties dezelfde naam en hetzelfde type hebben, én als de properties in dezelfde volgorde staan.
- Twee objecten van een anonymous type zijn gelijk als de waarden van al hun properties hetzelfde zijn.
- Een anonymous type bevat geen methods, enkel properties waardoor de mogelijkheden van dit type erg beperkt zijn. Daar 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 spelen een rol bij LINQ.

33.7 Partial methods

In het hoofdstuk NESTED EN PARTIAL CLASSES heb je gezien dat je de source code van een class kan opsplitsen over meerdere source files. Door de source code te verdelen over meerdere sources, houd je enerzijds de code overzichtelijk en gestructureerd. Anderzijds kan de ene source file de code bevatten die VS automatisch genereert, terwijl de andere source file de code bevat die door de programmeur zelf geschreven wordt. Partial classes herken je aan het sleutelwoord **partial**, dat bij de definitie van de class vermeld wordt bvb. `public partial class Werknemer`.

Tijdens het compileren wordt de code van de partial classes samengevoegd.

Naast partial classes kunnen ook **partial methods** gedefinieerd worden. Een partial method is een **lege private** method in de source van een **partial class**. Deze partial class source is meestal

gegenerereerd door VS.NET. De partial method is van het type **void**. De method kan enkel opgeroepen worden door een andere method van de *gegenerereerde* source.

Wanneer de programmeur dit wenst, kan hij de generereerde partial method in een andere source file van de betreffende partial class van code voorzien. Technisch gezien is het mogelijk om de implementatie van de partial method in dezelfde partial source file te schrijven waarin de partial method gedefinieerd is. Doorgaans bevindt de code van de partial method zich echter in een andere source file van de partial class. Partial methods worden vooral toegepast wanneer men wil vermijden dat de code, die door een tool gegenereerd wordt, aangepast wordt. De wijzigingen kunnen immers verloren gaan wanneer deze code opnieuw gegenereerd wordt.

Wanneer de programmeur de partial method implementeert, moet de signature uiteraard overeenkomen met deze van de generereerde partial method.

Als de partial method code bevat, zal deze code uitgevoerd worden wanneer de generereerde code deze method aanroept. Als de partial method nergens geïmplementeerd is door de programmeur, verwijdert de compiler automatisch de method-oproep naar deze partial method, zodat er geen completime of runtime fouten optreden (de code wordt geoptimaliseerd).

Veronderstel dat de partial class *Klant* door VS.NET als volgt gegenereerd is:

```
namespace CSharpPFCursus
{
    public partial class Klant
    {
        //code gegenereerd door Visual Studio...
        private string naam;
        public string Naam
        {
            get { return naam; }
            set { naam = value; NaamChanged(naam); } (1)
        }

        partial void NaamChanged(string naam); (2)
    }
}
```

- (1) Wanneer er een waarde toegekend wordt aan de property *Naam* van een *Klant* object of wanneer deze *Naam* property gewijzigd wordt, wordt de **set** routine van deze property uitgevoerd. Binnen deze **set** wordt de method *NaamChanged()* opgeroepen.
- (2) *NaamChanged()* is een partial method. De definitie van een partial method bevat het sleutelwoord **partial**. Het sleutelwoord **private** moet niet vermeld worden: partial methods zijn per definitie private. De definitie van de partial method bevat geen code.

Een programmeur kan nu bepalen wat er moet gebeuren als de property *Naam* van een *Klant* object ingevuld of gewijzigd wordt. In een andere source file van de partial class *Klant* kan hij code voorzien voor de partial method *NaamChanged()*.

```
namespace CSharpPFCursus
```

```
{
    public partial class Klant
    {
        //code geschreven door de programmeur
        partial void NaamChanged(string naam) (1)
        {
            Console.WriteLine("Naam gewijzigd naar {0}.", naam); (2)
        }
    }
}
```

- (1) De signature van de partial method moet behouden blijven. Je voorziet ook hier het sleutelwoord **partial**.
- (2) De implementatie van de partial method.

Je kan nu de partial method op de volgende manier uittesten:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            var klant = new Klant { Naam = "Jan" }; (1)
            klant.Naam = "Piet"; (2)
            Console.WriteLine(klant.Naam);
        }
    }
}
```

- (1) Bij de creatie van het *Klant* object wordt de property *Naam* ingevuld via een *object initializer*. Hierdoor wordt de **set** van de property *Naam* uitgevoerd waardoor dus ook de method *NaamChanged()* uitgevoerd wordt.
- (2) Wanneer de property *Naam* van het *Klant* object gewijzigd wordt, wordt de method *NaamChanged()* uitgevoerd. Je ziet het volgende op het scherm:

Indien de partial method *NaamChanged()* nergens geïmplementeerd is, wordt deze method-oproep in het **set** gedeelte van de *Naam* property gewoon genegeerd.
Je kan dit uittesten door de code van de method *NaamChanged()* in commentaar te plaatsen en het programma opnieuw uit te voeren. Je krijgt dan het volgende:



Hier zou je hetzelfde resultaat kunnen bereiken door gebruik te maken van events, maar partial methods zijn performanter.

33.8 Lambda (λ) expressions

33.8.1 Inleiding

In het hoofdstuk DELEGATES EN EVENTS heb je geleerd dat je met een delegate reference variabele kan verwijzen naar

- een method die elders gedefinieerd is én voldoet aan de signature van de delegate. Via de delegate reference variabele kan de gekoppelde method uitgevoerd worden.
- een *anonymous method*, een stukje code dat niet elders gedefinieerd is, maar gewoon ter plaatse (in-line) genoteerd staat in een blok. Een anonymous method wordt gebruikt als deze code slechts één keer gebruikt wordt.

Een voorbeeld ter herhaling:

```
namespace CSharpPFCursus

{
    class Program
    {
        delegate bool Filter(int getal); (1)

        private static bool IsEvengetal(int getal) (2)
        {
            return (getal % 2 == 0);
        }

        private static bool IsOnevenGetal(int getal) (3)
        {
            return getal % 2 == 1;
        }

        public static void Main(string[] args)
        {
            Filter evenGetal = IsEvengetal; (4)
            Console.WriteLine("Even getallen:");
            ToonGetallen(evenGetal); (5)

            Filter onevenGetal = IsOnevenGetal; (6)
        }
    }
}
```

```
Console.WriteLine("Oneven getallen");
ToonGetallen(onevenGetal); (7)

Filter getalDeelbaarDoorVijf = delegate(int getal) (8)
{
    return (getal % 5 == 0);
};
Console.WriteLine("Getallen deelbaar door 5:");
ToonGetallen(getalDeelbaarDoorVijf); (9)
}

private static void ToonGetallen(Filter filter) (10)
{
    var getallen = new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    foreach (var getal in getallen)
        if (filter(getal))
            Console.WriteLine(getal);
}
}
```

- (1) Definitie van een delegate *Filter*: met een reference variabele van het type *Filter* kan je verwijzen naar elke method met een *parameter* van het type *int* en een *returnwaarde* van het type *bool*.
- (2) Deze method voldoet aan de signature van de delegate *Filter*: de method controleert of het meegegeven getal *even* is of niet.



Deze method is een voorbeeld van wat men een **predicate method** noemt. Een *predicate method* krijgt een parameter binnen en geeft *true* of *false* terug, op basis van die parameter.

- (3) Deze method voldoet eveneens aan de signature van de delegate *Filter*: de method controleert of het meegegeven getal *oneven* is of niet.
Deze method is eveneens een voorbeeld van een **predicate method**.
- (4) Een *Filter* delegate variabele *evenGetal* verwijst naar de method *IsEvenGetal()*:
`Filter evenGetal = IsEvengetal;`
- (5) De method *IsEvenGetal()* wordt via de *Filter* delegate variabele *evenGetal* als parameter meegegeven aan de method *ToonGetallen()*.
De method *ToonGetallen()*, die de method *IsEvenGetal()* in een parameter van het type *Filter* ontvangt, – zie punt (10) – wordt uitgevoerd: van een reeks getallen worden enkel die getallen getoond waarvoor de method *IsEvenGetal()* de waarde *true* teruggeeft.
- (6) Een *Filter* delegate variabele *onevenGetal* verwijst naar de method *IsOnevenGetal()*:
`Filter onevenGetal = IsOnevenGetal;`

- (7) De method *IsOnevenGetal()* wordt via de *Filter* delegate variabele als parameter meegegeven aan de method *ToonGetallen()*.
 De method *ToonGetallen()* – zie punt (10) – wordt uitgevoerd: van een reeks getallen worden enkel die getallen getoond waarvoor de method *IsOnevenGetal()* de waarde true teruggeeft.
- (8) Een *Filter* delegate variabele *getalDeelbaarDoorVijf* verwijst nu naar een *anonymous method* die nagaat of het meegegeven getal deelbaar is door 5.
- (9) Deze anonymous method wordt via de *Filter* delegate variabele als parameter meegegeven aan de method *ToonGetallen()*.
 De method *ToonGetallen()* – zie punt (10) – wordt uitgevoerd: van een reeks getallen worden enkel die getallen getoond die deelbaar zijn door 5.
- (10) Definitie van de method *ToonGetallen()*: de method verwacht een *Filter* delegate object als parameter.
- (11) Voor elk getal wordt de method die op dit ogenblik gekoppeld is aan het meegegeven delegate object, uitgevoerd. Enkel de getallen waarvoor de method de waarde true teruggeeft, worden op het scherm getoond.

Een **lambda expressie** is een andere schrijfwijze voor een anonymous method. Een lambda expressie is een beknopte subroutine/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.

33.8.2 Een lambda expressie definiëren

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. Je kan bvb. met één lambda expressie een verzameling gegevens sorteren. Bij het imperatief programmeren is er veel meer code nodig daar het sorteringsgoritme volledig moet uitgeschreven worden. D.m.v. lambda expressies bekom je een grote flexibiliteit met weinig code.

Een lambda expressie bestaat uit 3 onderdelen:

- Een parameterlijst met nul, één of meerdere parameters die als argument(en) voor de lambda expressie gebruikt worden.
- Het symbool =>
- 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.

We verduidelijken dit met een aantal voorbeelden.

- **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; }
```

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

Dit is een alternatieve schrijfwijze voor de expression lambda

```
getal => getal % 2 == 0
```

```
tekst => {
    var letters = tekst.ToCharArray();
    Array.Reverse(letters);
    return new String(letters);
};
```

Deze statement lambda keert de volgorde van de letters van een opgegeven tekst om en geeft de omgekeerde tekst als resultaat terug.

Je kan deze statement lambda ook als volgt schrijven:

```
tekst => {
    var letters = tekst.ToCharArray();
    Array.Reverse(letters);
    Console.WriteLine(new String(letters));
}
```

Hier wordt de lambda als een method zonder returnwaarde gedefinieerd.

- Let op: om de lambda expressie's uit de bovenstaande voorbeelden te kunnen uitvoeren, moet je de expressies toewijzen aan een variabele met als type een delegate. Dit delegate type moet je eerst definiëren en de signature van de lambda expressie moet voldoen aan de signature van het delegate type.
Via een *delegate reference variabele* kan de gekoppelde lambda expressie verder in de code uitgevoerd worden.

Zo kan de lambda expressie

getal => getal * getal

op de volgende manier uitgevoerd worden via de delegate reference variabele kwadraat:

```
namespace CSharpPFCursus
{
    class Program
    {
        delegate int DelegateType (int getal); (1)
        static void Main(string[] args)
        {
            DelegateType kwadraat = getal => getal * getal; (2)
            Console.WriteLine(kwadraat(10)); (3)
        }
    }
}
```

(1) Definitie van het delegate type.

(2) De lambda expressie **getal => getal * getal**
toewijzen aan een variabele **kwadraat** met als type de gedefinieerde delegate.

(3) De lambda expressie uitvoeren via de delegate reference variabele
kwadraat.

- Het type van de input parameters van een lambda expressie wordt afgeleid uit de lambda expressie zelf en hoef je dus niet op te geven. Je mag het type van de parameters wel vermelden bvb.
(int getal) => getal * getal

33.8.3 Lambda expressies gebruiken

Een lambda expressie kan je op verschillende manieren in de code gebruiken en uitvoeren. Hier volgen enkele voorbeelden.

33.8.3.1 Een lambda expressie uitvoeren via een delegate reference variabele

Een lambda expressie kan aan een delegate reference variabele toegekend worden en via deze delegate variabele uitgevoerd worden:

```
namespace CSharpPFCursus
{
    class Program
    {
        delegate int FunctieMetTweeParameters(int getal1,int getal2); (1)
        delegate int FunctieMetEenParameter(int getal); (2)
        delegate int FunctieZonderParameters(); (3)

        public static void Main(string[] args)
        {
            FunctieMetTweeParameters som = (getal1,getal2)=>getal1+getal2; (4)
            Console.WriteLine(som(3,7)); (5)
            Console.WriteLine(som(10, 6)); (5)

            FunctieMetEenParameter kwadraat = getal => getal*getal; (6)
            Console.WriteLine(kwadraat(5)); (7)

            FunctieZonderParameters willekeurigGetal = () => new Random().Next(10); (8)
            Console.WriteLine(willekeurigGetal()); (9)
        }
    }
}
```

- (1) Definitie van een delegate met twee parameters van het type `int` en een returnwaarde van het type `int`.
- (2) Definitie van een delegate met één parameter van het type `int` en een returnwaarde van het type `int`.
- (3) Definitie van een delegate zonder parameters en een returnwaarde van het type `int`.
- (4) De lambda expressie `(getal1,getal2)=>getal1+getal2` wordt gedefinieerd en toegekend aan een delegate reference variabele `som` van het type `FunctieMetTweeParameters`. Een lambda expressie kan toegekend worden aan een variabele van het type delegate, mits de lambda expressie voldoet aan de signature van de delegate: twee inputparameters van het type `int` en een resultaatwaarde van het type `int`.
- (5) De lambda expressie wordt uitgevoerd via de delegate variabele `som`: het resultaat van de lambda expressie – de som van de twee meegegeven getallen – wordt op het scherm getoond.
- (6) De lambda expressie `getal => getal*getal` wordt gedefinieerd en toegekend aan een delegate reference variabele `kwadraat` van het type `FunctieMetEenParameter`.
- (7) De lambda expressie wordt uitgevoerd via de delegate variabele `kwadraat`: het resultaat van de lambda expressie – het kwadraat van het meegegeven getal – wordt op het scherm getoond.

- (8) De lambda expressie `() => new Random().Next(10)` wordt gedefinieerd en toegekend aan een delegate reference variabele `willekeurigGetal` van het type `FunctieZonderParameters`.
- (9) De lambda expressie wordt uitgevoerd via de delegate variabele `willekeurigGetal`: een willekeurig positief getal kleiner dan 10 wordt gegenereerd en op het scherm getoond.

33.8.3.2 Een lambda expressie als parameter van een method

Een lambda expressie kan eveneens als parameter meegegeven worden aan een method. De method ontvangt de lambda expressie in een parameter met als type een delegate. De lambda expressie moet voldoen aan de signature van deze delegate.

We werken het voorbeeld van paragraaf 33.8.1 uit met lambda expressies.

We kunnen de functies `IsEvenGetal(int getal)` en `IsOnevenGetal(int getal)`, en de anonymous functie vervangen door lambda expressies:

- lambda expressie voor de method `IsEvenGetal(int getal)`:

getal => getal % 2 == 0

- lambda expressie voor de method `IsOnevenGetal(int getal)`:

getal => getal % 2 == 1

- lambda expressie voor de anonymous method:

getal => getal % 5 == 0

In de volgende code worden deze lambda expressies als parameter doorgegeven aan een method:

```
namespace CSharpPFCursus
{
    class Program
    {
        delegate bool Filter(int getal); (1)
        public static void Main(string[] args)
        {
            Filter filterEvenGetallen = getal => getal % 2 == 0; (2)
            Console.WriteLine("Even getallen:");
            ToonGetallen(filterEvenGetallen); (3)

            Console.WriteLine("Oneven getallen:");
            ToonGetallen(getal => getal % 2 == 1); (4)

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

        private static void ToonGetallen(Filter filter) (6)
    }
}
```

```

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

```

(7)

- (1) Definitie van een delegate *Filter*: met een delegate reference variabele van het type *Filter* kan je verwijzen naar elke method met een *parameter* van het type `int` en een *returnwaarde* van het type `bool`, maar ook naar een lambda expressie.
- (2) De lambda expressie `getal => getal % 2 == 0` wordt gedefinieerd en toegekend aan een delegate reference variabele `filterEvenGetallen` van het type `Filter`. Deze lambda expressie voldoet aan de signature van de delegate: een expressie met één parameter van het type `int` en als returnwaarde het type `bool`.
Merk op dat je het type van de parameter(s) in een lambda expressie niet hoeft op te geven, het type wordt afgeleid uit de context van de lambda expressie, in dit geval bepaald door de delegate waarmee de lambda expressie verbonden wordt.
Je mag het type van de parameter(s) wel vermelden, de parameter moet je dan tussen ronde haakjes schrijven:
`Filter filterEvenGetallen = (int getal) => getal % 2 == 0;`
- (3) De lambda expressie wordt via de *Filter* delegate variabele `filterEvenGetallen` als parameter meegegeven aan de method `ToonGetallen()`.
De method `ToonGetallen()` – zie punt (6) – wordt uitgevoerd.
- (4) Je kan een lambda expressie ook rechtstreeks als parameter meegeven zonder deze eerst toe te kennen aan een delegate variabele. De method `ToonGetallen()` ontvangt de lambda expressie in een parameter met als type een delegate: de meegegeven lambda expressie moet uiteraard voldoen aan de signature van deze delegate.
De method `ToonGetallen()` – zie punt (6) – wordt uitgevoerd
- (5) Ook hier wordt een lambda expressie als parameter doorgegeven aan de method `ToonGetallen()`, die vervolgens uitgevoerd wordt.
- (6) Definitie van de method `ToonGetallen()`: de method verwacht een *Filter* delegate object – een method of een lambda expressie – als parameter.
- (7) Voor elk getal wordt de method of lambda expressie die op dit ogenblik gekoppeld is aan het delegate object, uitgevoerd. Enkel de getallen waarvoor de method de waarde `true` teruggeeft, worden op het scherm getoond.

33.8.3.3 Een lambda expressie als returnwaarde van een method

Een method met als returntype een delegate, kan eveneens als resultaatwaarde een lambda expressie bevatten.

Voorbeeld:

```

namespace CSharpPFCursus
{
    class Program

```

```

{
    delegate bool Filter(int getal);                                (1)

    static Filter MaakLambda()                                     (2)
    {
        Console.Write("Geef een getal: ");
        var deelbaarDoor = int.Parse(Console.ReadLine());          (3)
        return getal => getal % deelbaarDoor == 0;                  (4)
    }

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

public static void Main(string[] args)
{
    ToonGetallen(MaakLambda());                                    (7)
}
}

```

- (1) Definitie van een delegate *Filter*: met een delegate reference variabele van het type *Filter* kan je verwijzen naar elke method of lambda expressie met een *parameter* van het type *int* en een *returnwaarde* van het type *bool*.
- (2) Definitie van een method met als returnwaarde een delegate van het type *Filter*.
- (3) De gebruiker geeft een willekeurig getal in dat onthouden wordt in de lokale variabele *deelbaarDoor*.
- (4) De returnwaarde van deze method is de lambda expressie *getal => getal % deelbaarDoor == 0*. Deze lambda expressie maakt gebruik van deze lokale variabele *deelbaarDoor* en geeft true terug als een meegegeven getal deelbaar is door het getal dat door de gebruiker ingegeven werd, en anders false. De lambda expressie voldoet dus aan de signature van de delegate *Filter*.
- (5) Definitie van een method met als parameter een delegate van het type *Filter*.
- (6) Voor elk getal wordt de lambda expressie die op dit ogenblik gekoppeld is aan de delegate parameter, uitgevoerd. Enkel de getallen waarvoor de lambda expressie de waarde true teruggeeft, worden op het scherm getoond.
- (7) Bij de oproep van de method *ToonGetallen(MaakLambda())* wordt eerst de method *MaakLambda()* uitgevoerd. De lambda expressie die hier gecreëerd wordt door de functie *MaakLambda()*, wordt dan als parameter meegegeven aan de method *ToonGetallen()*, die verder deze lambda expressie zal gebruiken en toepassen voor elk getal van de array *getallen*. Stel vast dat de method *ToonGetallen()* nog steeds de juiste waarde van de lokale variabele *deelbaarDoor* kent.

	<p>De levensduur van een <i>lokale</i> variabele is normaal gezien beperkt vanaf het punt waarop de variabele gedeclareerd is tot het einde van de method of de structuur (if, while, ...) waarin de variabele gedeclareerd is. In principe is de levensduur van de variabele <i>deelbaarDoor</i> dus beperkt tot het einde van de method MaakLambda(). Wanneer je deze code uitvoert stel je echter vast dat de waarde van de variabele <i>deelbaarDoor</i> nog steeds gekend is in de method ToonGetallen(). Deze method gebruikt immers de juiste waarde, die door de gebruiker ingegeven werd. De lambda expressie die in de method MaakLambda() gecreëerd werd, maakt gebruik van de lokale variabele <i>deelbaarDoor</i>. Als nu de lambda expressie een langere levensduur heeft dan de lokale variabele, gebeurt zogenaamde variable lifting: de <i>lokale variabele</i> leeft even lang als de lambda expressie en behoudt zijn waarde wanneer de lambda expressie uitgevoerd wordt.</p>
---	--

33.9 Action en Func delegates

In plaats van 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. Zo kan je een method of lambda expressie toekennen aan een variabele van het type *Func* of *Action* en via deze variabelen de method of lambda expressie gebruiken. Ook hier geldt dat de signature van de method of lambda expressie moet voldoen aan de signature van de Action/Func delegate.

33.9.1 Action

Met een **Action delegate type** kan je verwijzen naar een method of lambda expressie **zonder returnwaarde** (dus van het type void), die aan de signature van de *Action delegate* voldoet. Deze sub of lambda expressie kan 0 tot 16 parameters bevatten.

Action werkt met generics: de parameters kunnen van elke welk type zijn.

Syntax:

Action<type parameter1, type parameter2, ...> variabelenaam = lambda expressie

Voorbeeld:

```
Action<int> kwadraat = getal => Console.WriteLine(getal * getal);
```

Je kan het volgende voorbeeld uittesten:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
```

```

    {
        Action<int> kwadraat = getal => Console.WriteLine(getal * getal); (1)
        kwadraat(10); (2)

        Action<string,int> tekstDeel = (tekst,vanaf) =>
            Console.WriteLine(tekst.Substring(vanaf)); (3)
        tekstDeel("VDAB", 2); (4)
    }
}

(1) Definitie van een Action delegate variabele kwadraat met één parameter van het type int, waaraan een lambda expressie toegekend wordt. De lambda expressie voldoet aan de signature van de Action delegate en toont het kwadraat van het opgegeven getal op het scherm.

(2) De lambda expressie wordt uitgevoerd via de Action delegate variabele kwadraat.

(3) Definitie van een Action delegate variabele tekstDeel met twee parameters waaraan een lambda expressie toegekend wordt. De lambda expressie voldoet aan de signature van de Action delegate en toont het deel tekst uit de string tekst vanaf de opgegeven positie vanaf op het scherm.

(4) De lambda expressie wordt uitgevoerd via de Action delegate variabele tekstDeel.

```

33.9.2 Func

Met een **Func delegate type** kan je verwijzen naar een method of lambda expressie **met een returnwaarde**, die aan de signature van de *Func delegate* voldoet. Deze functie of lambda expressie kan 0 tot 16 parameters bevatten.

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*.

Syntax:

Func<type parameter1, ..., type returnwaarde> variabelenaam = lambda expressie

Voorbeeld:

```
Func<int, int, int> som = (getal1, getal2) => getal1 + getal2;
```

De *laatste parameter* bepaalt het type van de *returnwaarde* van de method of lambda expressie. De overige parameters bepalen het type van de parameters van de function.

Je kan het volgende voorbeeld uittesten:

```
namespace CSharpPFCursus
{
    class Program
    {

        static void Main(string[] args)
```

```

    {
        Func<int, int, int> som = (getal1, getal2) => getal1 + getal2;      (1)
        Console.WriteLine(som(10,5));                                         (2)

        Func<string, int, string> tekstDeel = (tekst, vanaf) =>
            tekst.Substring(vanaf);                                         (3)
        Console.WriteLine(tekstDeel("VDAB", 2));                           (4)
    }
}
}

```

- (1) Definitie van een *Func delegate variabele* som met drie parameters van het type `int`, waaraan een *lambda expressie* toegekend wordt. De eerste twee parameters bepalen het type van de *parameters* van de *Func delegate*. De laatste parameter bepaalt het *returntype* van de *Func delegate*.
De lambda expressie voldoet aan de signature van de *Func delegate* en geeft de som terug van twee opgegeven getallen.
- (2) De lambda expressie wordt uitgevoerd via de *Func delegate variabele* som voor de meegegeven getallen.
- (3) Definitie van een *Func delegate variabele* tekstDeel met drie parameters waaraan een *lambda expressie* toegekend wordt. De eerste twee parameters bepalen het type van de *parameters* van de *Func delegate*. De laatste parameter bepaalt het *returntype* van de *Func delegate*.
De lambda expressie voldoet aan de signature van de *Func delegate* en geeft als resultaat het deel tekst uit de string `tekst` terug vanaf de opgegeven positie `vanaf`.
- (4) De lambda expressie wordt uitgevoerd via de *Func delegate variabele* tekstDeel voor de meegegeven tekst en positie.

- Met de sleutelwoorden *Func* en *Action* kan je ook verwijzen naar een method die elders gedefinieerd is.

Voorbeeld: koppeling van een method aan een *Action delegate*

```

namespace CSharpPFCursus
{
    class Program
    {
        static void UitbreidewerknemersLijst(
            Werknemer[] werknemers)
        {
            foreach (Werknemer eenWerknemer in werknemers)
                eenWerknemer.Afbeelden();
        }

        public static void Main(string[] args)
        {
            Werknemer[] werknemers = new Werknemer[2];
            werknemers[0] = new Arbeider("Asterix",
                new DateTime(2014, 1, 1), Geslacht.Man, 24.79m, 3);
            werknemers[1] = new Bediende("Obelix",
                new DateTime(2014, 1, 1), Geslacht.Man, 1500m);
            Action<Werknemer[]> toonWerknemers =
                UitbreidewerknemersLijst;
        }
    }
}

```

```
        toonWerknemers (werknemers);  
    }  
}
```

De method `UitgebreideWerknemersLijst` wordt hier aan een *Action* delegate variabele gekoppeld. De *Action* delegate verwacht een method *zonder returnwaarde (void)* met als parameter een *array van Werknemer objecten*. Via de *Action* delegate variabele wordt de gekoppelde method uitgevoerd, waarbij een *array van Werknemer objecten* als parameter meegegeven wordt.

- *Action* en *Func* delegates kunnen ook als type gebruikt worden voor de returnwaarde van een functie of voor de parameters van een functie/sub



oefeningen: Lambda expressies

34 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...).

D.m.v. LINQ query's kan je bewerkingen op gegevensverzamelingen uitvoeren: gegevens selecteren en filteren, gegevens sorteren, (statistische) berekeningen op gegevens uitvoeren,

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, XPath 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,

Daarnaast maakt LINQ gebruik van de nieuwe taalelementen die in het vorige hoofdstuk aan bod kwamen: je kan in een LINQ query gebruik maken van local type inference, object initializers, collection initializers, anonymous types, anonymous methods, extension methods (voorzien in de System.Linq namespace), lambda expressies, ... waardoor je met weinig code krachtige en complexe query's kan schrijven. Deze LINQ query's zijn over het algemeen beter leesbaar: complexe lussen, groeperings- en sorteringsalgoritmen, ... kunnen herleid worden tot één LINQ query waarbij de kans op logische fouten minder groot is.

LINQ kan in één query gegevens uit meerdere verzamelingen combineren. LINQ kan zelfs gegevens uit verschillende bronnen combineren.

De namespace **System.Linq** moet geïmporteerd zijn.

34.1 LINQ - Extension methods - Lambda Expressies

34.1.1 Achtergrondinformatie

Elke gegevensverzameling of collection die de interface *IEnumerable* of de generic vorm ervan *IEnumerable<T>* (afgeleid van *IEnumerable*) rechtstreeks of onrechtstreeks implementeert, bevat naast de mogelijkheid om elementen toe te voegen, te verwijderen of op te zoeken, ook de mogelijkheid om doorheen de elementen te itereren d.m.v. een *Enumerator* object. Een *Enumerator* object kan je zien als een verplaatsbare pointer naar elk element van een verzameling. Eerder heb je reeds een *foreach* constructie gebruikt om doorheen een array of een collectie van elementen te itereren. De interne werking van de *foreach* constructie is gebaseerd op een *Enumerator*.

In de static class *Enumerable* uit de namespace *System.Linq* zijn een aantal *extension methods* gedefinieerd voor de interface *IEnumerable*. Deze extension methods kunnen dus toegepast worden op elke gegevensverzameling die de interface *IEnumerable* implementeert zoals bvb. een array of een collection. De gegevens kunnen ook afkomstig zijn van een andere gegevensbron zoals een externe database, een XML document,

D.m.v. 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.

Tot deze methods behoren o.a. de methods *Select()*, *Where()*, *OrderBy()*,

De resultaatwaarde van deze extension methods kan opnieuw van het type *IEnumerable* zijn, zodat ze op elkaar resultaat kunnen opgeroepen worden.

Deze extension methods verwachten een delegate als parameter. Je kan dus een lambda expressie als parameter meegeven aan deze extension methods. Deze lambda expressie wordt dan uitgevoerd voor elk element van de verzameling.

Vandaar dat je op een *verzameling van elementen* met een zeer beknopte syntax een groot aantal bewerkingen kan uitvoeren.

34.1.2 Voorbeelden

Ter illustratie volgen hier enkele voorbeelden.

Voorbeeld 1:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var getallen = new[] { 0, 2, 1, 4, 3, 5, 7, 6, 8, 9 }; (1)

            Console.WriteLine("Getallen groter dan 3:");
            var getallenGroterDan3 = getallen.Where(getal => getal > 3); (2)
            foreach (var getal in getallenGroterDan3)
            {
                Console.WriteLine(getal);
            }

            Console.WriteLine("Gesorteerde lijst:");
            var gesorteerdeGetallen = getallen.OrderBy(getal=>getal); (3)
            foreach (var getal in gesorteerdeGetallen)
            {
                Console.WriteLine(getal);
            }

            Console.WriteLine("Aantal getallen groter dan 3:");
        }
    }
}
```

```
        Console.WriteLine(getallen.Count(getal => getal > 3));  
    }  
}
```

- (1) Declaratie en initialisatie van een array van getallen. Over deze array kan geïtereerd worden en op een array kunnen een aantal extension methods toegepast worden.
 - (2) De extension method *Where()* method bevat een lambda expressie als parameter. De lambda expressie bepaalt de voorwaarde waaraan de elementen moeten voldoen. Enkel de getallen waarvoor de lambda expressie true oplevert worden weerhouden. De resultaatwaarde van de *Where()* method is een verzameling van het type *IEnumerable<T>*, die de weerhouden getallen bevat. T is hier van het type *System.Int32*.

Over dit resultaat kan dus geïtereerd worden met een [foreach](#).

Merk op dat we het resultaat van de `Where()` method toekennen aan een variabele die met het sleutelwoord `var` gedefinieerd is. Het type van deze variabele wordt via *local type inference* bepaald:

```
Console.WriteLine("Getallen groter dan 3:");
var getallenGroterDan3 = getallen.Where(getal => getal > 3);
interface System.Collections.Generic.IEnumerable<out T>
Exposes the enumerator, which supports a simple iteration over a collection of a specified type.

T is System.Int32
```

- (3) De extension method *OrderBy()* method bevat een lambda expressie als parameter. Hier bepaalt de lambda expressie volgens welke sleutel de elementen van de verzameling gesorteerd moeten worden. Het resultaat van de *OrderBy()* method is opnieuw een verzameling van het type *IEnumerable<T>*, waarover geïtereerd kan worden.
 - (4) Aan de extension method *Count()* wordt een lambda expressie als parameter meegegeven. De *Count()* method geeft een getal terug, namelijk het *aantal* elementen dat voldoet aan een specifieke voorwaarde. Hier wordt de voorwaarde bepaald door de meegegeven lambda expressie die voor elk getal uit de array uitgevoerd wordt.

Voorbeeld 2:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var namen = new List<String> { "Asterix", "Obelix", "Idefix", "Ambiorix" }; (1)
            Console.WriteLine("Geef de beginletter(s) in: ");
            var beginVanNaam = Console.ReadLine();
            var gevondenNamen = namen.Where(naam =>
                naam.StartsWith(beginVanNaam));
            foreach (var naam in gevondenNamen) (2)
                Console.WriteLine(naam);
        }
    }
}
```

```
        Console.WriteLine(naam);
    }
}
```

- (1) Declaratie en initialisatie van een collectie van het type *List<string>*.

(2) De extension method *Where()* bevat een lambda expressie als parameter. De lambda expressie bepaalt de voorwaarde waaraan de elementen moeten voldoen: de namen moeten beginnen met de door de gebruiker opgegeven beginletter(s).
De resultaatwaarde van de *Where()* method is een *verzameling* die de geselecteerde namen bevat. Over dit resultaat kan geïtereerd worden met een *foreach*.

Voorbeeld 3:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var landen = new[]
            {
                new { Naam = "Frankrijk", Oppervlakte = 643427 },
                new { Naam = "Nederland", Oppervlakte = 41528 }
            };
            (1)

            var gesorteerdeLanden = landen.OrderBy(land => land.Oppervlakte);
            foreach (var eenLand in gesorteerdeLanden)
                Console.WriteLine(eenLand.Naam + " " + eenLand.Oppervlakte);

            Console.WriteLine("Totale oppervlakte van alle landen: {0}",
                landen.Sum(land => land.Oppervlakte));
            (2)

        }
    }
}
```

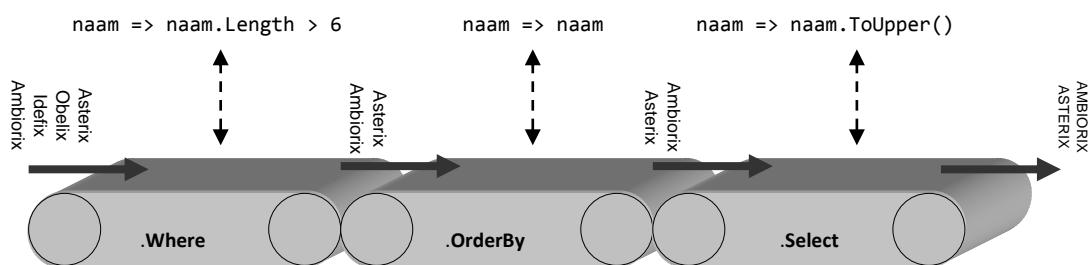
- (1) Hier wordt een array van objecten gecreëerd op basis van een *anonymous type* (zie paragraaf 33.6 ANONYMOUS TYPES). Elk object bevat een property *Naam* en een property *Oppervlakte*.
 - (2) De extension method *OrderBy()* bevat een lambda expressie die bepaalt op welke waarde deze array gesorteerd moet worden, hier de property *Oppervlakte*. Het resultaat van de *OrderBy()* method is een gesorteerde verzameling waarover geïtereerd kan worden.
 - (3) Hier wordt de statistische method *Sum()* toegepast op de verzameling: de lambda expressie, die als parameter meegegeven wordt, bepaalt de waarde die gesommeerd moet worden.

Voorbeeld 4: chaining

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var namen = new List<String> { "Asterix", "Obelix", "Idefix", "Ambiorix" };
            var geselecteerdeNamenInHoofdletters = namen.Where(naam =>
                naam.Length > 6).OrderBy(naam => naam).Select(naam =>
                naam.ToUpper());                                         (1)
            foreach (var naam in geselecteerdeNamenInHoofdletters)
                Console.WriteLine(naam);
        }
    }
}
```

- (1) Hier worden een aantal extension methods achtereenvolgens toegepast op een verzameling van namen. Hierbij wordt de volgende method toegepast op het resultaat van de vorige method. Eerst worden uit de opgegeven verzameling met de *Where()* method de namen geselecteerd waarvan de lengte groter is dan 6 (*naam => naam.Length > 6*). Het resultaat van deze method – terug een verzameling van namen – wordt via de *OrderBy()* method gesorteerd (*naam => naam*). De namen van deze gesorteerde verzameling worden vervolgens omgezet in hoofdletters (*naam => naam.ToUpper()*). Het resultaat van deze volledige instructie is een verzameling van gesorteerde namen met een lengte > 6 in hoofdletters, waarover geïtereerd kan worden.

Je kan je dit voorstellen alsof de gegevens over een transportband lopen en op verschillende plaatsen een proces ondergaan. De originele gegevens blijven ongewijzigd. Het resultaat van deze processen is een nieuwe gegevensverzameling.



34.2 LINQ - Query Comprehension Syntax/Query Expression Syntax

In de vorige paragraaf werden de LINQ query's opgebouwd d.m.v. extension methods en lambda expressies, waardoor je op een beknopte manier krachtige query's kan schrijven.

C# voorziet echter een nog meer beknopte syntax om LINQ query's te schrijven, de zogenaamde *Query Comprehension Syntax of Query Expression Syntax*: deze syntax is kort en krachtig zodat je met

nog minder code zeer complexe query's kan schrijven.

	<p>In de praktijk kan/moet je in een LINQ query soms een mix gebruiken van extension methods, lambda expressies en query comprehension syntax. De query expression syntax bevat immers niet alle query operatoren (bvb. de Distinct operator).</p> <p>LINQ query's met extension methods zijn interessant wanneer meerdere extension methods na elkaar – op elkaars resultaat – uitgevoerd worden, waarbij elke method een resultaat van het type <code>IEnumerable<T></code> teruggeeft.</p>  <p>LINQ query's met de query expression syntax zijn interessant bij joins en groups. Over het algemeen zijn deze query's beter leesbaar, vandaar dat deze syntax vaak de voorkeur geniet.</p> <p>Bij de uitvoering worden deze query's door de C# compiler vertaald naar equivalentie query's die gebruik maken van extension methods en lambda expressies.</p> <p>In één LINQ query kunnen beide schrijfwijzen gecombineerd worden, maar dit wordt eerder afgeraden o.w.v. de leesbaarheid.</p>
--	--

We kunnen de vorige LINQ query (uit voorbeeld 4) ook als volgt schrijven:

```
using System.Linq; (1)
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var namen = new List<String> { "Asterix", "Obelix", "Idefix", "Ambiorix" };
            var geselecteerdeNamenInHoofdletters =
                from naam in namen
                where naam.Length > 6
                orderby naam
                select naam.ToUpper(); (2)
            foreach (var naam in geselecteerdeNamenInHoofdletters) (3)
                Console.WriteLine(naam);
        }
    }
}
```

- (1) De namespace `System.Linq` moet geïmporteerd zijn.
- (2) Deze LINQ query geeft hetzelfde resultaat als de query met extension methods en lambda expressies in het vorige voorbeeld: een gesorteerde verzameling van namen in hoofdletters, met enkel de namen waarvan de lengte groter is dan 6.
Bij de uitvoering wordt deze query door de C# compiler vertaald naar een equivalentie query die gebruik maakt van extension methods en lambda expressies.
Het resultaat van de LINQ query wordt toegekend aan een variabele die gedeclareerd is met het

sleutelwoord `var`. Het type van deze variabele wordt via *local type inference* door de compiler bepaald en is hier `System.Collections.Generic.IEnumerable<string>`.

- (3) Het resultaat van deze LINQ query is eveneens een verzameling waarover geïtereerd kan worden.

	Deze LINQ query expressies kunnen SQL-achtige sleutelwoorden als <code>where</code> , <code>select</code> , <code>group by</code> , <code>orderby</code> , <code>join</code> , ... bevatten. 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.

```
var geselecteerdeNamenInHoofdletters =
    from naam in namen
    where naam.Length > 6
    orderby naam
    select naam.ToUpper();
```

- De `from` clausule bepaalt waaruit informatie gehaald wordt (hier de *List* `namen`), en introduceert ook een iteratie variabele (`naam`) die in de query expressie verder kan gebruikt worden om naar een element uit de gegevensverzameling te verwijzen. Deze iteratie variabele wordt hier verder gebruikt in de `where`-, `orderby`- en `select`-clauses.
- De `where` clausule bepaalt de voorwaarde(n) waaraan de geselecteerde elementen moeten voldoen en is dus een filter voor de gegevensverzameling.
- De `orderby` clausule sorteert de gefilterde gegevens.
- De `select` clausule bepaalt wélke gegevens uit de verzameling in het resultaat opgenomen worden.

	Een LINQ query wordt pas uitgevoerd op het ogenblik dat het resultaat van de LINQ query in de code gebruikt wordt in bvb. een <code>foreach</code> iteratie. In het bovenstaande voorbeeld wordt de LINQ query gedefinieerd op lijn (2), maar op dit ogenblik wordt de query nog niet uitgevoerd. Pas op lijn (3), waar het resultaat van de LINQ query met een <code>foreach</code> doorlopen wordt, wordt de LINQ query effectief uitgevoerd. Dit wordt <i>deferred execution</i> of <i>lazy evaluation</i> genoemd.
---	---

Met het volgende voorbeeld kan je dit nagaan:

```
using System.Linq;
namespace CSharpPFCursus
{
```

```

class Program
{
    public static void Main(string[] args)
    {
        var namen = new List<String>() { "Asterix", "Obelix" };
        var geselecteerdeNamen =
            from naam in namen
            where naam.StartsWith("A")
            select naam; (1)
        namen.Add("Assurancetourix"); (2)
        foreach (var naam in geselecteerdeNamen) (3)
            Console.WriteLine(naam);
    }
}

```

- (1) Hier wordt de LINQ query gedefinieerd: deze query selecteert alle namen die met de letter A beginnen. Deze LINQ query wordt op dit ogenblik echter nog niet uitgevoerd.
- (2) Ná de definitie van de LINQ query wordt er nog een naam (die met de letter A begint) aan de verzameling namen toegevoegd.
- (3) Hier wordt het resultaat van de LINQ query doorlopen: pas op dit ogenblik wordt de query uitgevoerd. Je stelt vast dat het resultaat ook de naam Assurancetourix bevat, die pas na de definitie van de LINQ query werd toegevoegd.

- Om er voor te zorgen dat een LINQ query toch onmiddellijk uitgevoerd wordt, kan je gebruik maken van de **ToList()** method. Deze geeft een *List<T>* terug

Je kan dit uittesten met het volgende voorbeeld:

```

var namen = new List<String>() { "Asterix", "Obelix" };
var geselecteerdeNamen =
    (from naam in namen
     where naam.StartsWith("A")
     select naam).ToList();

namen.Add("Assurancetourix");
foreach (var naam in geselecteerdeNamen)
    Console.WriteLine(naam);

```



In het resultaat wordt enkel de naam *Asterix* getoond. M.a.w. bij de definitie van de LINQ query wordt deze ook onmiddellijk uitgevoerd.

- De **FirstOrDefault()** method geeft je meteen het eerste element van het resultaat van de LINQ query. Indien het resultaat geen elementen bevat, geeft deze method **null** terug (of een andere default waarde, afhankelijk van het type van de elementen):

```
var namen = new List<String>()
    { "Asterix", "Obelix", "Assurancetourix" };
Console.WriteLine("Beginletter? ");
var beginLetter = Console.ReadLine();
var eersteNaam =
    (from naam in namen
     where naam.StartsWith(beginLetter)
     select naam).FirstOrDefault();
if (eersteNaam != null)
{
    Console.WriteLine(eersteNaam);
}
else
{
    Console.WriteLine("Het resultaat bevat geen elementen");
}
```

34.3 LINQ - Voorbeelden

Hier volgen enkele voorbeelden waarbij LINQ toegepast wordt op verzamelingen van zelf gedefinieerde objecten.

- Creëer een class *Bier* met de (automatic) properties: *BierNr*, *Biernaam*, *Alcohol* en *Brouwer*. De property *Brouwer* is van het type *Brouwer*, een class die je ook zelf ontwerpt (zie verder). Deze property houdt bij door welke brouwer een bier gebrouwen wordt. Voorzie ook de method *ToString()* die de *naam van het bier* en het *alcoholgehalte* teruggeeft.

```
namespace CSharpPFCursus
{
    public class Bier
    {
        public int BierNr { get; set; }
        public string Biernaam { get; set; }
        public float Alcohol { get; set; }
        public Brouwer Brouwer { get; set; } //associatie met een brouwer
        public override string ToString()
        {
            return Biernaam + ": " + Alcohol + "% alcohol";
        }
    }
}
```

- Creëer een class *Brouwer* met de (automatic) properties: *BrouwerNr*, *Brouwernaam* en *Belgisch*. De property *Belgisch* geeft aan of het een Belgische brouwerij betreft of niet. Een brouwer kan meerdere bieren brouwen. Voorzie in de class *Brouwer* eveneens een property *Bieren* die een lijst van bieren van een brouwer bijhoudt. Voorzie de method *ToString()* die de *Brouwernaam* teruggeeft met tussen haakjes of het een Belgische brouwerij is of niet.

```

namespace CSharpPFCursus
{
    public class Brouwer
    {
        public int BrouwerNr { get; set; }
        public string Brouwernaam { get; set; }
        public bool Belgisch { get; set; }
        public List<Bier> Bieren { get; set; }

        public override string ToString()
        {
            return "Brouwerij " + Brouwernaam + " (" +
                (Belgisch ? "Belgisch": "Niet Belgisch") + ")";
        }
    }
}

```

- Creeer een class *Brouwers* waarin je een method *GetBrouwers()* voorziet. Deze method creeert een lijst van *Brouwer* objecten (via object initializers) en geeft een brouwerslijst als resultaat terug. Voor elke brouwer wordt ook een lijst van *Bier* objecten (via object initializers) voorzien.

```

namespace CSharpPFCursus
{
    public class Brouwers
    {
        public List<Brouwer> GetBrouwers()
        {
            Brouwer palm = new Brouwer { BrouwerNr = 1, Brouwernaam = "Palm",
                Belgisch = true };
            palm.Bieren = new List<Bier> {
                new Bier {BierNr=1,Biernaam="Palm Doppel", Alcohol=6.2F, Brouwer=palm},
                new Bier {BierNr=2, Biernaam="Palm Green", Alcohol=0.1F, Brouwer=palm},
                new Bier {BierNr=3, Biernaam="Palm Royale", Alcohol=7.5F, Brouwer=palm}
            };

            Brouwer hertogJan = new Brouwer { BrouwerNr = 2, Brouwernaam = "Hertog Jan",
                Belgisch = false };
            hertogJan.Bieren = new List<Bier> {
                new Bier{ BierNr=4, Biernaam="Hertog Jan Dubbel", Alcohol=7.0F,
                    Brouwer=hertogJan},
                new Bier{ BierNr=5, Biernaam="Hertog Jan Grand Prestige", Alcohol=10.0F,
                    Brouwer=hertogJan}
            };

            Brouwer inBev = new Brouwer { BrouwerNr = 3, Brouwernaam = "InBev",
                Belgisch = true };
            inBev.Bieren = new List<Bier> {
                new Bier { BierNr=6, Biernaam="Belle-vue kriek L.A", Alcohol=1.2F,
                    Brouwer=inBev},
                new Bier { BierNr=7, Biernaam="Belle-vue kriek", Alcohol=5.2F,
                    Brouwer=inBev},
                new Bier { BierNr=8, Biernaam="Leffe Radieuse", Alcohol=8.2F,Brouwer=inBev},
                new Bier { BierNr=9, Biernaam="Leffe Triple", Alcohol=8.5F,Brouwer=inBev}
            };

            return new List<Brouwer> {palm, hertogJan, inBev };
        }
    }
}

```

- (1) Creatie van een nieuw *Brouwer* object d.m.v. *object initializers*.
- (2) De property *Bieren* van elk *Brouwer* object wordt opgevuld met een lijst van *Bier objecten*. Deze *Bier objecten* worden ook d.m.v. *object initializers* gecreëerd.
- (3) De *Brouwer* objecten worden verzameld in een lijst. Deze lijst is de returnwaarde van de method.

We passen nu een aantal LINQ query's toe op deze verzameling *Brouwer* objecten:

- **Een eenvoudige selectie**

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers(); (1)
            var alleBrouwers = from brouwer in brouwers select brouwer; (2)
            foreach (var brouwer in alleBrouwers) (3)
                Console.WriteLine(brouwer.ToString()); (4)
        }
    }
}
```

- (1) Je creëert eerst een verzameling van brouwers via een *Brouwers* object: de method *GetBrouwers()* van dit object geeft een verzameling *Brouwer* objecten (*List<Brouwer>*) als resultaat terug. Op deze verzameling van *Brouwer* objecten kan je een aantal LINQ query's loslaten.
- (2) Definitie van een LINQ query die alle brouwers van de verzameling selecteert.
- (3) Het resultaat – een verzameling van *Brouwer* objecten – kan met een iteratie doorlopen worden,
- (4) waarbij voor elke brouwer de method *ToString()* uitgevoerd wordt.

Opmerking:

In dit voorbeeld heeft deze LINQ query weinig praktische waarde want je kan evengoed de oorspronkelijke verzameling brouwers doorlopen om hetzelfde resultaat te bekomen:

```
foreach (var brouwer in brouwers)
    Console.WriteLine(brouwer.ToString());
```

- **LINQ query met een sortering**

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
```

```

public static void Main(string[] args)
{
    var brouwers = new Brouwers().GetBrouwers();
    var brouwersGesorteerdAsc = from brouwer in brouwers
        orderby brouwer.Brouwernaam select brouwer; (1)
    foreach (var brouwer in brouwersGesorteerdAsc)
        Console.WriteLine(brouwer.ToString());

    Console.WriteLine();
    var brouwersGesorteerdDesc = from brouwer in brouwers
        orderby brouwer.Brouwernaam descending select brouwer; (2)
    foreach (var brouwer in brouwersGesorteerdDesc)
        Console.WriteLine(brouwer.ToString());
}
}
}

```

- (1) Met het sleutelwoord `orderby` geef je aan op welke waarde je wil sorteren: hier de property *Brouwernaam* van elk *Brouwer* object.
Je kan op één of meerdere waarden sorteren. Meerdere waarden worden gescheiden door een komma.
- (2) Om aflopend te sorteren vermeld je het sleutelwoord `descending` bij de opgegeven sorteervaarde(n).

- **LINQ query met een anonymous type**

```

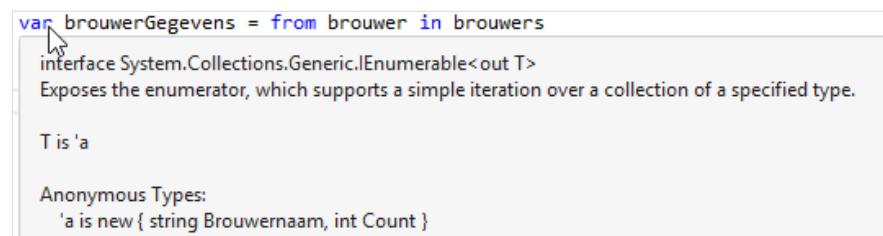
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var brouwerGegevens = from brouwer in brouwers
                select new {brouwer.Brouwernaam, brouwer.Bieren.Count}; (1)
            foreach (var brouwer in brouwerGegevens)
            {
                Console.WriteLine("{0}: {1} bier(en)",
                    brouwer.Brouwernaam, brouwer.Count); (2)
            }
        }
    }
}

```

- (1) Met het sleutelwoord `select` geef je aan welke gegevens in het resultaat van de LINQ query gestopt worden. Je kan hier ter plaatse 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:



```
var brouwerGegevens = from brouwer in brouwers
    interface System.Collections.Generic.IEnumerable<out T>
    Exposes the enumerator, which supports a simple iteration over a collection of a specified type.

T is 'a

Anonymous Types:
'a is new { string Brouwernaam, int Count }
```

- (2) In de `foreach` constructie kunnen deze properties (*Brouwernaam* en *Count*) van deze objecten aangesproken worden.

Je kan de properties van het *anonymous type* ook een naam geven, die je dan verder in de code kan gebruiken om de properties aan te spreken:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var brouwerGegevens = from brouwer in brouwers
                select new
                {
                    Brouwernaam = brouwer.Brouwernaam,
                    AantalBieren = brouwer.Bieren.Count };
            foreach (var brouwer in brouwerGegevens)
            {
                Console.WriteLine("{0}: {1} bier(en)",
                    brouwer.Brouwernaam, brouwer.AantalBieren);
            }
        }
    }
}
```

- **LINQ query met voorwaarden - where**

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
```

```

Console.WriteLine("Belgische brouwerijen:");
var belgischeBrouwerijen = from brouwer in brouwers
    where brouwer.Belgisch select brouwer; (1)
foreach (var brouwer in belgischeBrouwerijen)
    Console.WriteLine(brouwer.ToString());

Console.WriteLine("Niet-Belgische brouwerijen:");
var nietBelgischeBrouwerijen = from brouwer in brouwers
    where !brouwer.Belgisch select brouwer; (1)
foreach (var brouwer in nietBelgischeBrouwerijen)
    Console.WriteLine(brouwer.ToString());

Console.WriteLine("Brouwer met 2 bieren:");
var brouwersMet2Bieren = from brouwer in brouwers
    where brouwer.Bieren.Count == 2 select brouwer; (1)
foreach (var brouwer in brouwersMet2Bieren)
    Console.WriteLine(brouwer.Brouwernaam);

Console.WriteLine("Brouwers met een ingegeven aantal bieren: ");
Console.Write("Geef een aantal: ");
var aantal = int.Parse(Console.ReadLine());
var brouwersMetXAantalBieren = from brouwer in brouwers
    where brouwer.Bieren.Count == aantal select brouwer; (1)
foreach (var brouwer in brouwersMetXAantalBieren)
    Console.WriteLine(brouwer.Brouwernaam);

Console.WriteLine("Belgische brouwerijen met 3 bieren:");
var belgischeBrouwerijenMet3Bieren = from brouwer in brouwers
    where brouwer.Belgisch && brouwer.Bieren.Count == 3 (2)
        select brouwer;
foreach (var brouwer in belgischeBrouwerijenMet3Bieren)
    Console.WriteLine(brouwer.Brouwernaam);
}

}
}
}

```

- (1) Na het sleutelwoord `where` volgt een expressie die een `bool` waarde teruggeeft: voor elke brouwer wordt deze expressie geëvalueerd: enkel de brouwers waarvoor de expressie `true` oplevert, worden opgenomen in het resultaat.
- (2) Je kan meerdere voorwaarden combineren. Hiervoor gebruik je de C# syntax.

- **Van een hiërarchische structuur naar een “platte structuur”**

In het statement `var brouwers = new Brouwers().GetBrouwers();` is `brouwers` een verzameling (een `List`) van `Brouwer` objecten, waarbij elk `Brouwer` object op zich een verzameling (eveneens een `List`) van `Bier` objecten bevat. Het `brouwers` object is dus een genestte of hiërarchische structuur.



Als je de gegevens van de bieren wil aanspreken, dan volstaat de volgende code **NIET**:

```

using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var alleBrouwers = new Brouwers().GetBrouwers();
            var bieren = from brouwer in alleBrouwers
                         select brouwer.Bieren;

            foreach (var b in bieren)
                Console.WriteLine(b.GetType());
        }
    }
}
  
```

(1)

(1) Het resultaat van deze LINQ query is een verzameling van elementen van het type *List<Bier>* :

```

C:\WINDOWS\system32\cmd.exe
System.Collections.Generic.List`1[CSharpPFCursus.Bier]
System.Collections.Generic.List`1[CSharpPFCursus.Bier]
System.Collections.Generic.List`1[CSharpPFCursus.Bier]
System.Collections.Generic.List`1[CSharpPFCursus.Bier]
Press any key to continue . . .
  
```

Elk element is op zich een verzameling van elementen, namelijk een lijst van bieren voor elke brouwer. Je krijgt de bieren dus niet als afzonderlijke elementen maar gegroepeerd in verzamelingen, één verzameling voor elke brouwer.

Om de gegevens van de bieren aan te spreken, kan je als volgt tewerk gaan:

```

using System.Linq;
namespace CSharpPFCursus
{
    class Program
  
```

```

{
    public static void Main(string[] args)
    {
        var brouwers = new Brouwers().GetBrouwers();
        var bieren = from brouwer in brouwers
                     from bier in brouwer.Bieren
                     select bier; (1)
        foreach (var bier in bieren)
            Console.WriteLine(bier.ToString());
    }
}
}

```

- (1) Je hebt dus een dubbele iteratie nodig: één iteratie over de brouwer elementen (`from brouwer in brouwers`) en één iteratie over de bier elementen van elke brouwer (`from bier in brouwer.Bieren`).

Het resultaat is een verzameling van *Bier* objecten

Op deze manier zet je een hiërarchische structuur om in een “platte structuur”.

	<p>Je kan ook gebruik maken van de extension method SelectMany() om hetzelfde resultaat te bekomen:</p>  <pre> var brouwers = new Brouwers().GetBrouwers(); var bieren = brouwers.SelectMany(brouwer => brouwer.Bieren); foreach (var bier in bieren) Console.WriteLine(bier.ToString()); </pre> <p>Deze method geeft de elementen van de drie bierenverzamelingen (één verzameling per brouwer) als afzonderlijke Bier objecten terug.</p>
--	--

Voorbeeld: een overzicht van de alcoholarme bieren

```

namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var alcoholarmeBieren = from brouwer in brouwers
                                    from bier in brouwer.Bieren
                                    where bier.Alcohol < 2.0F
                                    select bier;
            foreach (var bier in alcoholarmeBieren)
                Console.WriteLine(bier.ToString());
        }
    }
}

```

```
}  
of  
  
namespace CSharpPFCursus  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            var brouwers = new Brouwers().GetBrouwers();  
            var alcoholarmeBieren = brouwers.SelectMany(brouwer => brouwer.Bieren)  
                .Where(bier=>bier.Alcohol <2.0F);  
            foreach (var bier in alcoholarmeBieren)  
                Console.WriteLine(bier.ToString());  
        }  
    }  
}
```

- **LINQ query met functies**

In een LINQ query kan je gebruik maken van functies om berekeningen of bewerkingen op gegevens uit te voeren. De volgende voorbeelden kan je uittesten:

```
using System.Linq;  
namespace CSharpPFCursus  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            var brouwers = new Brouwers().GetBrouwers();  
            // aantal Belgische brouwers  
            var belgischeBrouwers = from brouwer in brouwers  
                where brouwer.Belgisch  
                select brouwer; (1)  
            Console.WriteLine("Aantal Belgische brouwers: {0}",  
                belgischeBrouwers.Count()); (2)  
  
            // totaal aantal bieren  
            var alleBieren = from brouwer in brouwers  
                from bier in brouwer.Bieren  
                select bier; (3)  
            Console.WriteLine("Totaal aantal bieren: {0}", alleBieren.Count()); (4)  
  
            // of in één statement  
            Console.WriteLine("Totaal aantal bieren: {0}",  
                (from brouwer in brouwers from bier in brouwer.Bieren  
                select bier).Count()); (5)
```

```

//aantal Belgische bieren
var belgischeBieren = from brouwer in brouwers
                        where brouwer.Belgisch
                        from bier in brouwer.Bieren
                        select bier; (6)
Console.WriteLine("Aantal Belgische bieren: {0}",
                  belgischeBieren.Count()); (7)

//Gemiddelde alcoholgehalte van alle bieren
var bieren = from brouwer in brouwers
              from bier in brouwer.Bieren
              select bier; (8)
var gemiddeldeAlcohol = bieren.Average(bier => bier.Alcohol); (9)
Console.WriteLine("Gemiddelde alcoholgehalte van de bieren: {0}",
                  gemiddeldeAlcohol);

//Gemiddelde van alle getallen van een array
int[] getallen = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 }; (10)
var gemiddelde = getallen.Average(); (11)
Console.WriteLine("Het gemiddelde van deze getallen is: {0}",
                  gemiddelde);
}
}
}

```

- (1) Het resultaat is een verzameling van *Brouwer* objecten.
- (2) Op deze verzameling kan de Count() method toegepast worden, die het aantal elementen van de verzameling, dus het aantal Belgische brouwers, teruggeeft.
- (3) Het resultaat is een verzameling van *Bier* objecten.
- (4) Op deze verzameling kan de Count() method toegepast worden, die het aantal elementen van de verzameling, dus het totaal aantal bieren, teruggeeft.
- (5) De statements (3) en (4) kunnen door één statement vervangen worden. Let op de haakjes: de Count() method wordt toegepast op het resultaat van de LINQ query, die daarom volledig tussen ronde haakjes vermeld wordt.
- (6) Het resultaat is een verzameling van *Bier* objecten.
- (7) Op deze verzameling kan de Count() method toegepast worden, die het aantal elementen van de verzameling, dus het totaal aantal Belgische bieren, teruggeeft.
- (8) Het resultaat is een verzameling van *Bier* objecten.
- (9) Hier wordt het gemiddelde alcoholgehalte van de bieren uit de vorige verzameling berekend. Om aan te geven voor welke property van de *Bier* objecten het gemiddelde berekend moet worden, gebruik je een lambda expressie als parameter van de Average() method.
- (10) Definitie en initialisatie van een array integers.
- (11) Berekening van het gemiddelde van deze getallen.

- **LINQ query met een groepering**

In een LINQ query kan je gegevens groeperen met een **group ... by ...** clause.

Een **group ... by ...** splitst een verzameling op in meerdere deelgroepen, waarbij elk element van eenzelfde deelgroep dezelfde sleutelwaarde of *Key* heeft. Zo'n deelgroep bevat dus een *Key* property en een *verzameling* van elementen met dezelfde waarde voor deze *Key* property. Het resultaat van een **group ... by ...** is een verzameling van *IGrouping* objecten.

Je kan dit vergelijken met een woordenlijst van een boek. In een woordenlijst worden de trefwoorden gegroepeerd op basis van de eerste letter. Je kan een woordenlijst dus beschouwen als een verzameling van groepen trefwoorden, waarbij alle trefwoorden van eenzelfde groep dezelfde beginletter (*Key*) hebben. Alle trefwoorden die bvb. met de letter A beginnen behoren tot één groep met als *Key* de waarde A.

Index

A	}	groep 1: Key waarde = 'A'
<i>Add method</i>		
<i>Aggregate method</i>		
<i>Average method</i>		
...	}	groep 2: Key waarde = 'B'
B		
<i>Base class</i>		
<i>Button class</i>		
...		

Wanneer we alle trefwoorden van deze index willen overlopen, hebben we twee iteraties nodig:

- een iteratie om over de groepen te itereren op basis van de *Key* (*elke letter*).
- een tweede (geneste) iteratie om voor elke *Key* over de objecten (*de trefwoorden*) van de betreffende groep te itereren.

In het volgende voorbeeld wordt dit toegepast.

Stel dat we de brouwers willen opsplitsen in Belgische en Niet-Belgische brouwers. Deze opsplitsing gebeurt op basis van de property *Belgisch* van de *Brouwer* objecten.

In dit geval ontstaan er *twee* groepen:

- een groep van Belgische brouwers: een verzameling van *Brouwer* objecten met als *Key* waarde *true* voor de property *Belgisch*.
- een groep van Niet-Belgische brouwers: een verzameling van *Brouwer* objecten met als *Key* waarde *false* voor de property *Belgisch*.

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
```

```

{
    public static void Main(string[] args)
    {
        var brouwers = new Brouwers().GetBrouwers();
        var opBelgisch = from brouwer in brouwers
                        group brouwer by brouwer.Belgisch;           (1)
        foreach (var welNietBelgisch in opBelgisch)            (2)
        {
            Console.WriteLine(welNietBelgisch.Key ? "Belgisch" : "Niet-Belgisch");   (3)
            foreach (var brouwer in welNietBelgisch)          (4)
                Console.WriteLine(brouwer.Brouwernaam);
        }
    }
}
}

```

- (1) Tussen **group** en **by** vermeld je welke elementen je wil groeperen, ná **by** vermeld je op basis van welke waarde(n) je wil groeperen. Vermits de property *Belgisch* slechts twee waarden kan aannemen (true of false) is het resultaat hier een verzameling van twee groepen: een groep met als Key waarde true en een groep met als Key waarde false.
Daarnaast bevat elke groep een verzameling van *Brouwer* objecten, die dezelfde waarde voor deze *Key* hebben.

De variabele *opBelgisch* is dus een verzameling met twee groepen.

	Key (Belgisch)	verzameling brouwers
groep1	true	Palm InBev
groep2	false	Hertog Jan

- (2) Een eerste iteratie over alle groepen op basis van de *Key*. De variabele *welNietBelgisch* is een iteratie variabele waarmee je naar elke groep (*IGrouping* object) van de verzameling groepen kan verwijzen.
- (3) Van elke groep wordt de *Key* property getoond, hier omgevormd naar een betekenisvolle tekst: voor de *Key* waarde true wordt de tekst “Belgisch” getoond, voor de *Key* waarde false de tekst “Niet-Belgisch”.
- (4) Een tweede iteratie om per groep de *Brouwer* objecten te overlopen: elk *IGrouping* object bevat een verzameling van *Brouwer* objecten met dezelfde *Key* waarde.

	<p>De bovenstaande LINQ query kan je ook schrijven met de extension method <i>GroupBy()</i> waaraan je een lambda expressie als parameter meegeeft:</p> <pre> var brouwers = new Brouwers().GetBrouwers(); var opBelgisch = brouwers.GroupBy(brouwer => brouwer.Belgisch); </pre> <p>Het resultaat is eveneens een verzameling van deelgroepen, die je op dezelfde manier kan overlopen.</p>
---	---

In het resultaat van een groepquery worden de groepen niet gesorteerd. Indien je de groepen wil sorteren moet je deze sortering zelf in de LINQ query voorzien.

Voorbeeld: groepering op aantal bieren **zonder** sortering op het aantal

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var opAantalBieren = from brouwer in brouwers
                                  group brouwer by brouwer.Bieren.Count;
            foreach (var aantalBieren in opAantalBieren)
            {
                Console.WriteLine(aantalBieren.Key);
                foreach (var brouwer in aantalBieren)
                    Console.WriteLine(brouwer.Brouwernaam);
            }
        }
    }
}
```

	Key	verzameling brouwers
groep1	3	Palm
groep2	2	Hertog Jan
groep3	4	InBev

Merk op dat de verzameling brouwers van elke groep hier telkens slechts één brouwer bevat.

Dezelfde query **met** een sortering op aantal:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var opAantalBieren = from brouwer in brouwers
                                  group brouwer by brouwer.Bieren.Count
                                  into mijnGroep
                                  orderby mijnGroep.Key
                                  select mijnGroep; (1)
            foreach (var aantalBieren in opAantalBieren)
            {
                Console.WriteLine(aantalBieren.Key);
                foreach (var brouwer in aantalBieren)
```

```
        Console.WriteLine(brouwer.Brouwernaam);
    }
}
```

- (1) Met het sleutelwoord **into** in een LINQ query creëer je een tijdelijke variabele waarin je het resultaat van een **select**, **group by** of **join** (zie verder) bijhoudt. Via deze variabele kan je verder in de query naar de groepen verwijzen om ze bvb. te sorteren of om de elementen van de groepen te tellen of om andere query operaties op de groepen uit te voeren.

In deze LINQ query wordt het resultaat van de groepeerquery gesorteerd op de Key, het aantal bieren.

	Key	verzameling brouwers
groep1	2	Hertog Jan
groep2	3	Palm
groep3	4	InBev



Een LINQ query moet steeds eindigen op **group** of **select**.

In het volgende voorbeeld worden enkel de brouwers die 3 bieren brouwen getoond:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var aantalBieren = from brouwer in brouwers
                                group brouwer by brouwer.Bieren.Count
                                into mijnGroep
                                where mijnGroep.Key == 3
                                select mijnGroep;
            foreach (var aantalBieren in aantalBieren)
            {
                Console.WriteLine(aantalBieren.Key);
                foreach (var brouwer in aantalBieren)
                    Console.WriteLine(brouwer.Brouwernaam);
            }
        }
    }
}
```

In het onderstaande voorbeeld wordt het aantal bieren van respectievelijk de Belgische en Niet-Belgische brouwerijen getoond:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var opBelgisch = from brouwer in brouwers
                            from bier in brouwer.Bieren
                            group bier by brouwer.Belgisch
                            into bieren
                            select new { Belgisch = bieren.Key,
                                         AantalBieren = bieren.Count() }; (1)
            foreach (var groep in opBelgisch)
            {
                Console.WriteLine((groep.Belgisch ? "Belgische bieren: " :
                                         "Niet-Belgische bieren: ") +
                                         groep.AantalBieren); (2)
            }
        }
    }
}
```

- (1) Deze LINQ query heeft een verzameling van twee groepen als resultaat:

	Belgisch	AantalBieren
groep 1	true	7
groep 2	false	2

- (2) Voor elke groep wordt de *Key* waarde (property *Belgisch*, hier omgezet naar een betekenisvolle tekst), en het aantal bieren van de groep (property *AantalBieren*) getoond.

Het volgende voorbeeld toont ook de namen van de bieren, gegroepeerd op Belgische en Niet-Belgische brouwerijen:

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var brouwers = new Brouwers().GetBrouwers();
            var opBelgisch = from brouwer in brouwers
```

```

        from bier in brouwer.Bieren
        group bier by brouwer.Belgisch
        into bieren
        select new { Bieren = bieren,
                    Belgisch = bieren.Key,
                    AantalBieren = bieren.Count() };           (1)
foreach (var groep in opBelgisch)
{
    Console.WriteLine((groep.Belgisch ? "Belgische bieren: " :
                           "Niet-Belgische bieren: ") + groep.AantalBieren);
    foreach (var bier in groep.Bieren)                  (2)
        Console.WriteLine(bier.Biernaam);
}
}
}
}

```

- (1) Deze LINQ query heeft een verzameling van twee groepen als resultaat:

	Belgisch	AantalBieren	Bieren
groep 1	true	7	verzameling bieren
groep 2	false	2	verzameling bieren

Hierbij bevat de property *Bieren* de verzameling van de bieren van de betreffende groep.

- (2) Voor elke groep kan over deze verzameling bieren geïtereerd worden om bvb. de biernamen te tonen.

- **LINQ query met een join**

D.m.v. een LINQ query kan je gegevens uit verschillende verzamelingen combineren op basis van een property met dezelfde waarde. Het volgende voorbeeld illustreert dit:

```

using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var personen = new[]
            {
                new { Naam = "Jan", AantalKinderen = 1 },
                new { Naam = "Mieke", AantalKinderen = 2 },
                new { Naam = "Els", AantalKinderen = 1 }
            };
                                         (1)
            var toegangsprijzen = new[]
            {
                new { AantalKinderen = 1, Bedrag = 10 },
                new { AantalKinderen = 2, Bedrag = 25 }
            };
                                         (2)
        }
    }
}

```

```

    var toegangsPrijzenPerPersoon =
        from persoon in personen
        join toegangsprijs in toegangsprijzen
        on persoon.AantalKinderen equals toegangsprijs.AantalKinderen
        select new { persoon.Naam, toegangsprijs.Bedrag };           (3)

    foreach (var persoon in toegangsPrijzenPerPersoon)
        Console.WriteLine("{0}: {1} euro", persoon.Naam, persoon.Bedrag); (4)
    }
}
}
}

```

- (1) Definitie en initialisatie van een array van objecten op basis van een anonymous type: elk object bevat een property *Naam* en een property *AantalKinderen*.
- (2) Definitie en initialisatie van een array van objecten op basis van een anonymous type: elk object bevat een property *AantalKinderen* en een property *Bedrag*, waarbij de property *Bedrag* de toegangsprijs bevat voor het bijhorende aantal kinderen.
- (3) Via deze LINQ query wordt voor elk object van de array personen de toegangsprijs opgehaald uit de array toegangsprijzen op basis van het aantal kinderen. Enkel de personen waarvoor er een overeenkomstig aantal kinderen voorkomt in de array toegangsprijzen worden opgenomen in het resultaat van de join (= inner join).
Het resultaat van deze query is een verzameling van objecten op basis van een anonymous type.
- (4) Over het resultaat van de LINQ query kan geïtereerd worden.

- **LINQ query op basis van een string als gegevensbron**

Een string (tekst) kan je ook beschouwen als een verzameling van gegevens waarop je LINQ query's kan toepassen.

Voorbeeld 1: een string als een verzameling van tekens

```

using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var tekst = "Vlaamse Dienst Voor Arbeidsbemiddeling en Beroepsopleiding";
            var zoekLetterA = from letter in tekst.ToUpper()
                             where letter == 'A' select letter;           (1)
            Console.WriteLine("De tekst {0} bevat {1} A's.", 
                             tekst, zoekLetterA.Count());                  (2)
        }
    }
}

```

- (1) Het resultaat van deze LINQ query is een verzameling van *char* elementen met de waarde 'A', dus alle A's uit de tekst.

- (2) Op deze verzameling kan je de `Count()` method toepassen om het aantal elementen, dus het aantal A's, te kennen.

Voorbeeld 2: een string als een verzameling van woorden

```
using System.Linq;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            var zin = "The quick brown fox jumps over the lazy dog"; (1)
            var woorden = zin.Split(new[] { ' ' });
            var woordenVan3Tekens = from woord in woorden
                                   where woord.Length == 3
                                   select woord.ToLower(); (2)
            foreach (var woord in woordenVan3Tekens.Distinct())
                Console.WriteLine(woord);
        }
    }
}
```

- (1) Met de `Split()` method splits je een tekst in stukjes op basis van één of meerdere scheidingstekens, die je als een array van `char` tekens als parameter van deze method meegeeft. In dit voorbeeld wordt de spatie (`' '`) als scheidingsteken opgegeven. Meerdere scheidingstekens worden gescheiden door een komma.
Bvb. `zin.Split(new[] { ' ', ';' })` splitst de opgegeven zin op basis van spaties en puntkomma's.
Het resultaat van deze method is een verzameling van alle woorden van de tekst.
- (2) Met een LINQ query worden uit deze verzameling woorden enkel de woorden met een lengte van drie tekens geselecteerd. Het resultaat is een deelverzameling van woorden van de tekst.
- (3) Over deze deelverzameling kan geïtereerd worden. Met de `Distinct()`method verwijder je dubbels uit het resultaat van de LINQ query.

	Op het internet vind je heel wat voorbeelden van LINQ, zie bvb. <i>"101 LINQ Samples"</i> .
---	--



oefeningen: LINQ

35 Files and Streams – I/O

Tot nu toe heb je in een programma gebruik gemaakt van variabelen en constanten om tijdens de uitvoering van het programma gegevens te onthouden. Denk aan getallen, strings, datums, objecten, ... die tijdens de uitvoering van het programma gebruikt worden.

Deze variabelen worden echter enkel in het interne geheugen van je computer onthouden, zolang het programma draait. Dit betekent dat deze gegevens vroeg of laat verloren gaan bvb. wanneer een bepaalde method uitgevoerd is, maar zeker wanneer het programma afgesloten wordt.

Soms is het echter de bedoeling dat de gegevens, die in een programma gebruikt worden, **elders** opgeslagen worden zodat ze achteraf opnieuw kunnen opgevraagd en gebruikt worden.

In dit hoofdstuk leer je hoe je gegevens kan schrijven naar en terug kan inlezen vanuit een bestand op je harde schijf. Je hebt reeds kennis gemaakt met de class *StreamReader* in het hoofdstuk EXCEPTIONS paragraaf GEGARANDEerde TIJDige OPKUIS VAN RESOURCES.

In dit hoofdstuk komen naast de class *StreamReader* ook andere classes aan bod.

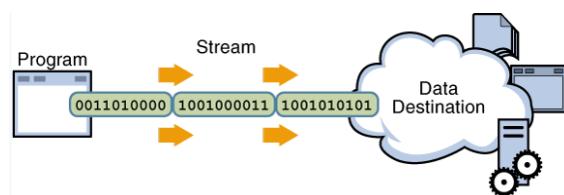
Verder leer je classes kennen waarmee je bestanden en directories kan manipuleren vanuit programmacode.

35.1 Streams – FileStream

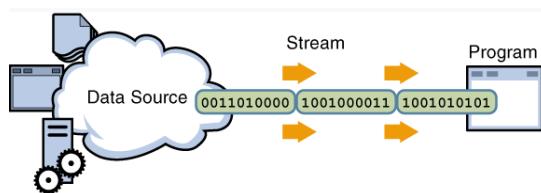
.NET maakt gebruik van zogenaamde **streams** om gegevens te transporteren. Een stream is een gegevensstroom, een verzameling van bytes.

Om gegevens naar een andere locatie te versturen of van een andere locatie te ontvangen, maakt een programma altijd gebruik van een **Stream object**.

gegevens wegschrijven

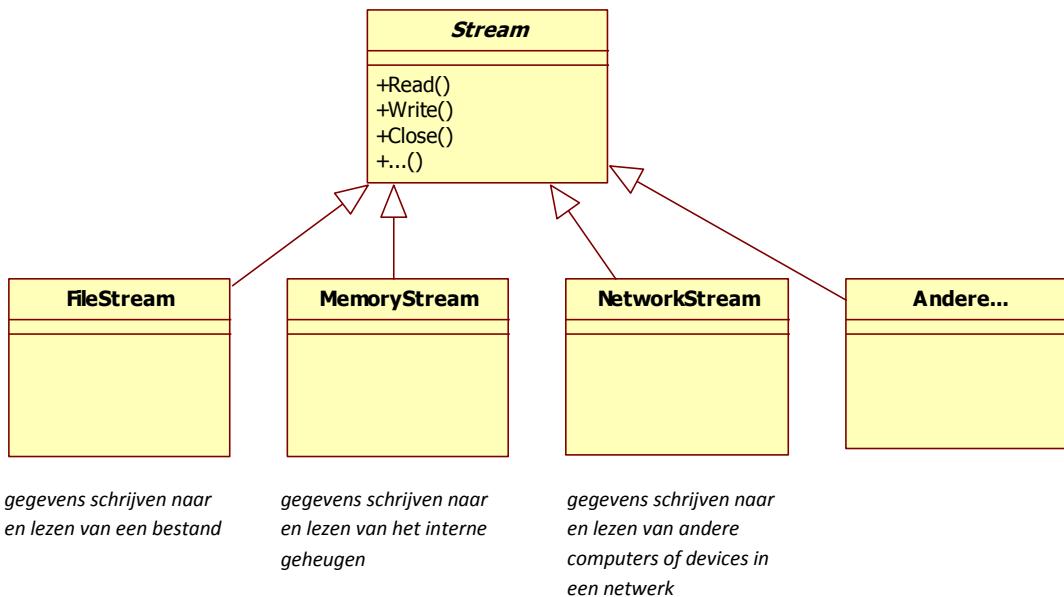


gegevens ophalen



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

Volgende figuur toont een greep uit deze verzameling stream classes:

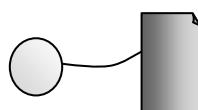


Welke stream class je moet gebruiken wordt bepaald door het soort gegevens en de bestemming van de gegevens die je wil transporteren.

In deze cursus beperken we ons tot het wegschrijven naar en het inlezen van gegevens van een bestand (file) op de harde schijf. Hiervoor gebruik je de class **FileStream**. Met objecten van deze class verpak je de gegevens die je wil transporter in een array van bytes.

Als je bvb. tekst wil opslaan in een bestand, moeten er meerdere dingen gebeuren:

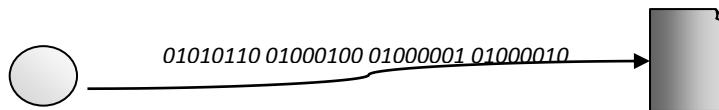
- Eerst wordt een **FileStream** object gecreëerd
- Dit *FileStream* object wordt gekoppeld aan een *bestand*. Let op: één *FileStream* object kan tegelijkertijd maar aan één bestand gekoppeld worden.



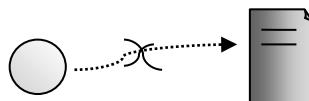
- Streams schrijven “bytes” naar een bestand. De tekst die je wil wegschrijven, moet dus geconverteerd worden naar een array van bytes. Dit wordt *encoding* genoemd. Het Stream object neemt dit voor zijn rekening.

“VDAB” —————→ 01010110 01000100 01000001 01000010

Met de *Write()* method van het *FileStream* object worden deze bytes naar het bestand weggeschreven.



- Eenmaal de gegevens opgeslagen zijn in het bestand, moet het *FileStream* object gesloten worden. Hierdoor wordt de koppeling met het bestand verbroken. 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 gebruik gemaakt van een *FileStream* object.

35.2 Bestanden en directories

Het .Net Framework bevat een aantal classes die je toelaten om vanuit je programmacode informatie te verkrijgen over bestanden en directories, en op een eenvoudige manier bewerkingen met bestanden en directories uit te voeren.

Deze classes behoren eveneens tot de namespace *System.IO*.

35.2.1 De classes Directory en DirectoryInfo

Met de classes **Directory** en **DirectoryInfo** kan je :

- 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

```
using System;
using System.IO;                                         (1)

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string directorynaam = @"C:\Data";
```

```
if (Directory.Exists(directorynaam)) (2)
{
    Console.WriteLine("De directory {0} bestaat.", directorynaam);
    if (Directory.GetDirectories(directorynaam).Count() != 0 (3)
        || Directory.GetFiles(directorynaam).Count() != 0)
    {
        Console.WriteLine("De directory met inhoud wordt verwijderd"); (4)
        Directory.Delete(directorynaam, true);
    }
    else
    {
        Console.WriteLine("De directory wordt verwijderd"); (5)
        Directory.Delete(directorynaam);
    }
}
else
{
    Console.WriteLine("De directory {0} bestaat niet.", directorynaam); (6)
    Console.WriteLine("De directory wordt gecreëerd");
    Directory.CreateDirectory(directorynaam);
}
}
```

- (1) Importeer de namespace *System.IO*. Deze namespace bevat de classes waarmee je gegevens kan inlezen en wegschrijven alsook classes om te werken met bestanden, directories.
 - (2) Hier ga je na of de opgegeven directory bestaat.
 - (3) Zo ja, dan ga je na of de opgegeven directory subdirectories bevat. De method *GetDirectories()* geeft een array van strings terug met de namen van de subdirectories.
 - (4) Je kan ook nagaan of de opgegeven directory bestanden bevat. De method *GetFiles()* geeft een array van strings terug met de namen van de bestanden.
 - (5) De opgegeven directory met subdirectories en/of bestanden wordt verwijderd. Met de tweede parameter **true** geef je aan dat ook de inhoud van de directory moet verwijderd worden. Doe je dit niet, dan treedt een Exception op.
 - (6) De tweede parameter van de *Delete()* method mag je weglaten als de directory die je wil verwijderen, leeg is.
 - (7) Als de opgegeven directory niet bestaat, kan je op deze manier de directory creëren.



- Merk op dat de class *Directory static* methods bevat zodat je geen object van deze class hoeft te creëren om deze methods te gebruiken.
- De functionaliteit van de class **DirectoryInfo** overlapt met deze van de class *Directory*. Van deze class moet je echter eerst een object creëren om de methods te kunnen gebruiken.

35.2.2 De classes File en FileInfo

Met de classes **File** en **FileInfo** kan je:

- 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

```
using System;
using System.IO;

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string directorynaam = @"C:\Data";
            string bestandsnaam = "Pizzas.txt";
            string bestand = directorynaam + @"\" + bestandsnaam;

            Directory.CreateDirectory(directorynaam); (1)

            if (!File.Exists(bestand)) (2)
            {
                Console.WriteLine("Het bestand {0} bestaat niet", bestand);
                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); (4)

        }
        else
        {
            Console.WriteLine("Het bestand {0} bestaat", bestand);
            string bestandsinformatie =
                "Datum creatie: " + File.GetCreationTime(bestand) + (5)

                System.Environment.NewLine +
                "Datum laatst gebruikt: " + File.GetLastAccessTime(bestand); (5)
            Console.WriteLine(bestandsinformatie);

            Console.WriteLine("De inhoud van het bestand:");
        }
    }
}
```

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

        File.Delete(bestand);
    }

}

}

}
```

- (1) De directory C:\Data wordt gecreëerd. Als deze directory reeds bestaat, wordt de method genegeerd.
 - (2) Hier ga je na of het opgegeven bestand bestaat.
 - (3) Zo niet, dan wordt het bestand aangemaakt. Dit kan op verschillende manieren gebeuren (zie verder).
Hier wordt de *static* method *WriteAllText()* van de class *File* gebruikt waarbij je de naam van het bestand en de weg te schrijven tekst als parameters meegeeft.
Als het bestand reeds bestaat, wordt het overschreven.
Als de method uitgevoerd is, wordt het bestand automatisch gesloten.
 - (4) Met de method *AppendAllText()* van de class *File* kan je tekst aan een bestaand bestand toevoegen of het bestand creëren als het nog niet bestaat. Als de method uitgevoerd is, wordt het bestand automatisch gesloten.
 - (5) Met de methods *GetCreationTime()* en *GetLastAccessTime()* van de class *File* kan je informatie over een bestaand bestand opvragen.
 - (6) De method *ReadAllText()* van de class *File* laat je toe de inhoud van een bestaand bestand te lezen. Deze inhoud wordt als een string teruggegeven.
 - (7) Hier wordt het opgegeven bestand verwijderd.



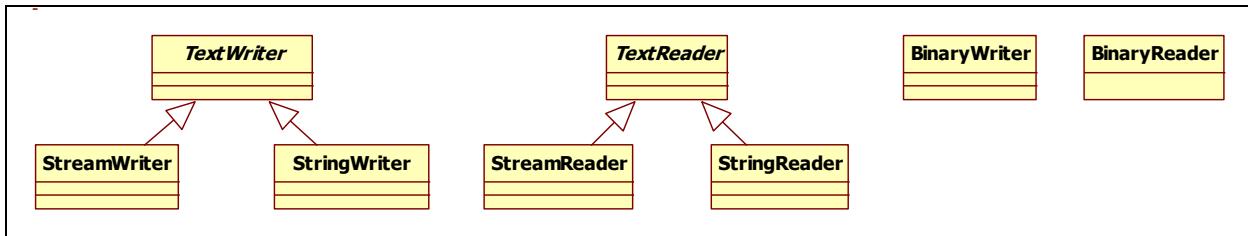
- Merk op dat de class *File static* methods bevat zodat je geen object van deze class hoeft te creëren om deze methods te gebruiken.
 - Deze methods kunnen *Exceptions* veroorzaken zodat het aangewezen is om een try-catch blok te gebruiken.
 - De functionaliteit van de class **FileInfo** overlapt met deze van de class *File*. Van deze class moet je echter eerst een object creëren om de methods te kunnen gebruiken.

35.3 Schrijvers – 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* objecten en koppelen deze aan het bestand waarnaar je wil schrijven of waaruit je wil lezen.

Een overzicht:



TextWriter (abstract)	een schrijver waarmee je een sequentiële reeks karakters (tekst) kan wegschrijven.
StreamWriter	een schrijver om karakters naar een <i>stream</i> te schrijven met een specifieke encoding (standaard UTF-8).
StringWriter	een schrijver om gegevens naar een <i>string</i> te schrijven.
TextReader (abstract)	een lezer waarmee je een sequentiële reeks karakters (tekst) kan inlezen.
StreamReader	een lezer waarmee je karakters uit een <i>stream</i> kan inlezen met een specifieke encoding.
StringReader	een lezer om gegevens uit een <i>string</i> 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.

35.3.1 Voorbeeld 1: StreamWriter - StreamReader

35.3.1.1 Tekst wegschrijven naar een bestand - StreamWriter

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

Werkwijze:

- 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.

Voorbeeld:

```
using System;
using System.IO; (1)

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string locatie = @"C:\Data\";

            try
            {
                using (var schrijver = new StreamWriter(locatie + "Pizzas.txt")) (2)
                {
                    string titel = "Prijslijst pizza's";
                    schrijver.WriteLine(titel); (3)
                    for (var teller = 1; teller <= titel.Length; teller++)
                        schrijver.Write('*'); (3)
                    schrijver.WriteLine();
                    schrijver.WriteLine("Pizza Margherita (tomatensaus - mozzarella): " +
                        "8 EUR");
                    schrijver.WriteLine("Pizza Napoli " +
                        "(tomatensaus-mozzarella-ansjovis-kappers-olijven): 9.5 EUR");
                    schrijver.WriteLine("Pizza Vegetariana " +
                        "(tomatensaus-mozzarella-assortiment van groenten): " +
                        "9.5 EUR");
                } (4)

                //gegevens toevoegen
                using (var schrijver = new StreamWriter(locatie + "Pizzas.txt", true)) (5)
                {
                    schrijver.WriteLine("Pizza Lardiera (tomatensaus-mozzarella-spek): " +

```

```
        "9.5 EUR");
    }
}
catch (IOException) (6)
{
    Console.WriteLine("Fout bij het schrijven naar het bestand!");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
}
}
```

- (1) Importeer de namespace *System.IO*, die de class *StreamWriter* bevat.
- (2) Creëer een *StreamWriter* object dat je toelaat om tekst naar een bestand te *schrijven*. De naam (+ de locatie) van het bestand – C:\Data\Pizzas.txt – geef je mee via een parameter van de constructor van dit *StreamWriter* object.

	<ul style="list-style-type: none">• De class <i>StreamWriter</i> bevat meerdere overloaded constructors die je bvb. toelaten om aan te geven of een bestaand bestand met dezelfde naam mag overschreven worden of dat de nieuwe gegevens aan het bestaande bestand moeten toegevoegd worden. Je kan ook meegeven welke encoding er moet gebruikt worden bij het wegschrijven van de gegevens (standaard UTF-8).• Je omsluit de creatie en het gebruik van het <i>StreamWriter</i> object met een <i>using</i> structuur. Op het einde van de <i>using</i> structuur worden het bestand en de gekoppelde stream automatisch gesloten en de gebruikte resources terug vrijgegeven (zie hoofdstuk EXCEPTIONS paragraaf GEGARANDEERDE TIJDIGE OPKUIS VAN RESOURCES).
---	---

- (3) Met de *Write()* en *WriteLine()* methods schrijf je gegevens naar het bestand.

	Deze methods bevatten meerdere overloads. Raadpleeg hiervoor de MSDN.
---	---

- (4) Bij de sluit accolade van de *using* structuur wordt het *StreamWriter* object automatisch gesloten. Je hoeft dit dus niet zelf te doen. Hierdoor komen de gebruikte resources vrij en kan het bestand *Pizzas.txt* door andere programma's gebruikt worden.

- (5) Bij de creatie van het *StreamWriter* object kan je in de constructor een tweede parameter (**true** / **false**) meegeven. Als het te gebruiken bestand nog niet bestaat, wordt er automatisch een nieuw bestand gecreëerd en geopend. Als het bestand echter wel al bestaat, kan je met deze tweede parameter aangeven of er toch een nieuw bestand moet gecreëerd worden (**false**) of dat het bestaande bestand moet geopend worden en de nieuwe gegevens hieraan toegevoegd worden (**true**).
- (6) Werken met een bestand kan tot verschillende fouten leiden: *DirectoryNotFoundException*, *FileNotFoundException*, ... Al deze exceptions zijn afgeleid van *System.IO.IOException*. Met deze *catch* opdracht vang je dus mogelijke I/O fouten op.



Open het bestand *Pizzas.txt* met een teksteditor bvb. Notepad. Merk op dat de tekst, die met een *StreamWriter* weggeschreven werd, perfect leesbaar is.

35.3.1.2 Tekst inlezen vanuit een bestand – *StreamReader*

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

Werkwijze:

- 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 te lezen.
- Sluit het *StreamReader* object waardoor de onderliggende stream en dus het bestand losgekoppeld wordt en door andere programma's kan gebruikt worden.

Voorbeeld:

```
using System;
using System.IO; (1)

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string locatie = @"C:\Data\";
            try
            {
                string regel;
                using (var lezer = new StreamReader(locatie + "Pizzas.txt")) (2)
                {
                    while ((regel = lezer.ReadLine()) != null) (3)
                    {
                        Console.WriteLine(regel);
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}
catch (IOException) (5)
{
    Console.WriteLine("Fout bij het lezen van het bestand!");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
}
}
```

- (1) Importeer de namespace *System.IO* die de class *StreamReader* bevat.
- (2) Creëer een *StreamReader* object dat je toelaat om gegevens uit een tekstbestand te *lezen*. De naam (+ de locatie) van het bestand – **C:\Data\Pizzas.txt** – geef je mee via een parameter van de constructor van dit *StreamReader* object.
Je omsluit de creatie en het gebruik van het *StreamWriter* object eveneens met een *using* structuur zodat op het einde van de *using* structuur het bestand en de gekoppelde stream automatisch gesloten en de gebruikte resources terug vrijgegeven worden.
- (3) Met de *ReadLine()* method van het *StreamReader* object lees je de tekst van het bestand regel per regel. Deze method geeft de gelezen regel als een string terug. De method geeft **null** terug als je ná de laatste regel nog een regel probeert te lezen.

	<p>Je kan ook de <i>Read()</i> method van het <i>StreamReader</i> object gebruiken om gegevens in te lezen. Hiermee lees je de tekst teken per teken, waarbij elk teken als een <i>Int32</i> object teruggegeven wordt. De method geeft -1 terug als je na het laatste teken nog een teken probeert te lezen:</p> <p> <code>int teken; using (var lezer = new StreamReader(locatie + "Pizzas.txt")) { while ((teken = lezer.Read()) != -1) { Console.Write((char)teken); } }</code></p>
--	--

- (4) Bij de sluitaccoade van de *using* structuur wordt het *StreamReader* object automatisch gesloten. Je hoeft dit dus niet zelf te doen. Hierdoor komen de gebruikte resources vrij en kan het bestand *Pizzas.txt* door andere programma's gebruikt worden.
- (5) Werken met een bestand kan tot verschillende fouten leiden: *DirectoryNotFoundException*, *FileNotFoundException*, ... Al deze exceptions zijn afgeleid van *System.IO.IOException*. Met deze *catch* opdracht vang je dus mogelijke I/O fouten op.

35.3.2 Voorbeeld 2: StreamWriter – StreamReader

In dit voorbeeld creëren we een verzameling van *Pizza* objecten. Daarna schrijven we de pizzagegevens als tekst weg naar een bestand zodat we de gegevens terug kunnen opvragen indien nodig.

35.3.2.1 De class Pizza

Creëer eerst de class *Pizza* met de properties *Naam* (string), *Onderdelen* (List<String>) en *Prijs* (decimal) en overschrijf de method *ToString()* die de naam en de prijs van een pizza toont:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CSharpPFCursus
{
    public class Pizza
    {
        public string Naam { get; set; }
        public List<String> Onderdelen { get; set; }
        public decimal Prijs { get; set; }

        public override string ToString()
        {
            return this.Naam + " : " + this.Prijs + "EUR";
        }
    }
}
```

(2)

35.3.2.2 Pizzagegevens wegschrijven – StreamWriter

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {

            List<Pizza> pizzas = new List<Pizza>
            {
                new Pizza
                {
                    Naam = "Pizza Margherita",
                    Onderdelen = new List<string> { "Tomaten saus", "Mozzarella" },
                    Prijs = 8m
                },
            };
        }
    }
}
```

```
new Pizza
{
    Naam = "Pizza Vegetariana",
    Onderdelen = new List<string>
        { "Tomatenstaat", "Mozzarella", "Groenten" },
    Prijs = 9.5m
},
new Pizza
{
    Naam = "Pizza Napoli",
    Onderdelen = new List<string>
        { "Tomatenstaat", "Mozzarella",
            "Ansjovis", "Kappers", "Olijven" },
    Prijs = 10m
}
};

//de pizzagegevens wegschrijven:
//elke pizza wordt als één regel tekst weggeschreven
string locatie = @"C:\Data\";
StringBuilder pizzaRegel;
try
{
    using (var schrijver = new StreamWriter(locatie + "Pizzas.txt"))
    {
        foreach (var pizza in pizzas)
        {
            pizzaRegel = new StringBuilder();
            pizzaRegel.Append(pizza.Naam + ':') ; (1)
            pizzaRegel.Append(pizza.Onderdelen.Count.ToString() + ':'); (2)
            foreach (string onderdeel in pizza.Onderdelen)
            {
                pizzaRegel.Append(onderdeel + ':');
            }
            pizzaRegel.Append(pizza.Prijs);
            schrijver.WriteLine(pizzaRegel);
        }
    }
    catch (IOException)
    {
        Console.WriteLine("Fout bij het schrijven naar het bestand!");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
}
```

- (1) De pizzagegevens worden gescheiden door een dubbele punt.
- (2) Van elke pizza wordt ook het *aantal* onderdelen weggeschreven zodat we achteraf de pizzagegevens correct kunnen inlezen.

35.3.2.3 Pizzagegevens inlezen – StreamReader

```
using System;
using System.Collections.Generic;
using System.IO;
namespace CSharpPFCursus
{
    class Program
    {
        public static void Main(string[] args)
        {
            string locatie = @"C:\Data\";

            List<Pizza> pizzas = new List<Pizza>();

            string pizzaRegel;
            string pizzaNaam;
            int aantalOnderdelen;
            List<string> pizzaOnderdelen;
            decimal prijs;

            //de pizzagegevens inlezen
            try
            {
                using (var lezer = new StreamReader(locatie + "Pizzas.txt"))
                {
                    while ((pizzaRegel = lezer.ReadLine()) != null)
                    {
                        string[] gegevens = regel.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
                            });
                    }
                }
            }
            catch (IOException)
            {
                Console.WriteLine("Fout bij het lezen van het bestand!");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
//de pizza gegevens tonen
foreach (Pizza pizza in pizzas)
{
    Console.WriteLine(pizza.ToString());
    foreach (string onderdeel in pizza.Onderdelen)
    {
        Console.WriteLine(onderdeel);
    }
    Console.WriteLine();
}
}
```

35.3.3 Gegevens binair wegschrijven: BinaryWriter – BinaryReader

35.3.3.1 BinaryWriter

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
- ...

In een aantal gevallen zal het binair wegschrijven van gegevens minder bytes in beslag nemen dan indien hetzelfde gegeven als tekst wordt weggeschreven. Zo wordt de waarde **false** binair als één byte opgeslagen waar de tekst "**false**" vijf bytes

(**File.WriteAllText(@"C:\Data\False.txt", "false");**) in beslag neemt.

Voorbeeld:

```
using System;
using System.IO;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string tekst = "Lottogetallen";
            int aantalGetallen = 45;
            byte[] lottogetallen = { 1, 27, 32, 33, 44, 12, 5 };
            float winst = 18f;
            string datum = DateTime.Now.ToShortDateString();

            using (BinaryWriter schrijver = new BinaryWriter(
```

```

        File.Open(@"C:\Data\Lottogetallen.bin", FileMode.Create))) (1)
    {
        schrijver.Write(aantalGetallen); (2)
        schrijver.Write(tekst); (2)
        schrijver.Write(lottogetallen); (2)
        schrijver.Write(winst); (2)
        schrijver.Write(datum); (2)
    } (3)
}
}
}
}

```

- (1) 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`).

- (2) De *BinaryWriter* class bevat meedere overloaded *Write()* methods waarmee je de verschillende gegevenstypen kan wegschrijven.
- (3) Het *BinaryWriter* object wordt binnen een **using** structuur gecreëerd zodat dit object en dus ook de onderliggende *FileStream* met het gekoppelde bestand op het einde van de **using** gesloten wordt.



Open het bestand *Lottogetallen.bin* met een teksteditor bvb. Notepad. Merk op dat de gegevens die met een *BinaryWriter* weggeschreven werden, niet echt leesbaar zijn. Enkel **strings** en **chars** zijn herkenbaar.

35.3.3.2 *BinaryReader*

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

Voorbeeld:

```

using System;
using System.IO;
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            string tekst;
            int aantalGetallen;

```

```
byte[] lottogetallen;
float winst;
string datum;

using (BinaryReader lezer = new BinaryReader(
    File.Open(@"C:\Data\Lottogetallen.bin", FileMode.Open))) (1)
{
    aantalGetallen = lezer.ReadInt32(); (2)
    tekst = lezer.ReadString(); (2)
    lottogetallen = lezer.ReadBytes(7); (2)
    winst = lezer.ReadSingle(); (2)
    datum = lezer.ReadString(); (2)
}

Console.WriteLine(aantalGetallen);
Console.WriteLine(tekst);
foreach (byte lottogetal in lottogetallen)
    Console.Write("{0} ", lottogetal);
Console.WriteLine();
Console.WriteLine(winst);
Console.WriteLine(datum);
}
}
```

- (1) 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`).

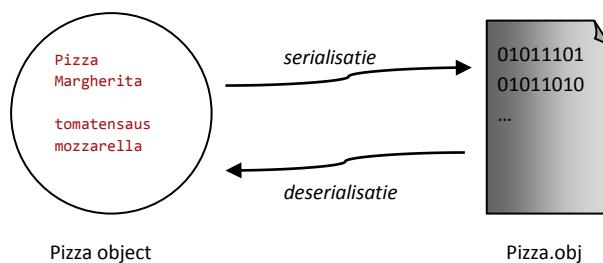
- (2) De class *BinaryReader* bevat verschillende methods om de respectievelijke gegevenstypes in te lezen: *ReadInt32()*, *ReadSingle()*, *ReadString()*, ...
Vermits bij het binair wegschrijven van een string de lengte van de string ook weggeschreven wordt, kan de string achteraf perfect terug ingelezen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

35.4 Objecten wegschrijven – Serialization

Naast primitieve data types en tekst kunnen ook “volledige objecten” naar een bestand weggeschreven worden of van een bestand ingelezen worden.

Een object wordt weggeschreven als een “stream van bytes”. Bij het terug inlezen ervan wordt het weggeschreven object teruggedraaid.

Het proces van wegschrijven van objecten wordt “serialisatie” genoemd. Het terug inlezen van de objecten wordt “deserialisatie” genoemd.



Bij het serialisatieproces wordt de zogenaamde “state” van het object weggeschreven: de waarde en het type van elke membervariabele evenals het type van het object. Zo kan het object achteraf exact gereconstrueerd worden.

Een voorwaarde hiervoor is dat de class, waarvan de objecten instanties zijn, zogenaamd “**Serializable**” of “**serialiseerbaar**” is.

In .NET maak je een class *Serializable* door de class te voorzien van het attribuut [\[Serializable\]](#).

```
[Serializable]
public class Pizza
{}
```

	<ul style="list-style-type: none"> Default worden alle public en private membervariabelen (properties) geserialiseerd, behalve de membervariabelen die gemarkeerd zijn met het attribuut [NonSerialized]. Als je objecten van een afgeleide klasse wil serialiseren, moet de base class ook <i>Serializable</i> zijn. Members die niet geserialiseerd kunnen worden of waarvan je zelf niet wil dat ze geserialiseerd worden (vb. een wachtwoord of een kredietkaartnummer), markeer je met het attribuut [NonSerialized]. Bij de reconstructie van het object krijgen deze members een default waarde. Wil je volledige controle over het serialisatie-/deserialisatieproces, dan kan je de class de interface <i>ISerializable</i> uit de namespace <i>System.Runtime.Serialization</i> laten importeren. Dit valt buiten het bestek van deze cursus.
---	---

In het volgende voorbeeld gaan we een Pizza object opslaan in een bestand d.m.v. Serialisatie.

Pas hiervoor de class *Pizza* aan:

```
namespace CSharpPFCursus
{
    [Serializable]                                         (1)
    public class Pizza
    {
        public string Naam { get; set; }
        public List<String> Onderdelen { get; set; }           (2)
        public decimal Prijs { get; set; }

        public override string ToString()
        {
            return this.Naam + " : " + this.Prijs + "EUR";
        }
    }
}
```

- (1) Daar we Pizza objecten willen wegschrijven naar een bestand, moet de class Pizza *Serializable* zijn en dus voorzien zijn van het attribuut `[Serializable]`.
- (2) Wanneer een class members bevat die als type een reference type (uit het .NET Framework of een eigen geschreven class) hebben, moet dit reference type eveneens serializable zijn.
De class Pizza bevat een property van het type `List<String>`.
Dit levert geen probleem bij het serialiseren van de class Pizza daar de class `List<T>` eveneens serialiseerbaar is (zie MSDN).

35.4.1 Objecten wegschrijven – BinaryFormatter

Voorbeeld:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;          (1)
using System.Runtime.Serialization;                                (2)

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            Pizza pizzaMargherita = new Pizza
            {
                Naam = "Pizza Margherita",
                Onderdelen = new List<string> { "Tomatensaus", "Mozzarella" },
                Prijs = 8m
            };
        }
    }
}
```

```
try (4)
{
    using (var bestand = File.Open(@"C:\Data\Pizza.obj",
        FileMode.OpenOrCreate))
    {
        var schrijver = new BinaryFormatter();
        schrijver.Serialize(bestand, pizzaMargherita);
    }
    Console.WriteLine("Serialisatie geslaagd"); (5)
}
catch (SerializationException) (6)
{
    Console.WriteLine("Fout bij het serialiseren");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
}
```

- (1) Importeer de namespace `System.Runtime.Serialization.Formatters.Binary` die de class **BinaryFormatter** bevat.
 - (2) Importeer de namespace `System.Runtime.Serialization` die de class **SerializationException** bevat.
 - (3) Creëer een Pizza object.
 - (4) Creëer een *FileStream* object waaraan het bestand, waarin je het Pizza object wil opslaan, gekoppeld wordt. Dit *FileStream* object wordt bij het serialiseren gebruikt.
 - (5) Om objecten naar een bestand weg te schrijven (serialiseren) heb je een **BinaryFormatter** object nodig. Dit object wordt hier gecreëerd.
 - (6) 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.
 - (7) Bij het serialisatieproces kan een zogenaamde *SerializationException* optreden, bvb. als het attribuut [**Serializable**] in de class ontbreekt. Deze exception vang je uiteraard best op.



Open het bestand *Pizza.obj* met een teksteditor bvb. Notepad. Merk op dat de gegevens die met een *BinaryFormatter* weggeschreven werden, niet echt leesbaar zijn. Enkel **strings** en **chars** zijn herkenbaar.

35.4.2 Objecten inlezen – BinaryFormatter

Voorbeeld:

```
using System;
```

```
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                using (var bestand = File.Open(@"C:\Data\Pizza.obj",
                    FileMode.Open, FileAccess.Read)) (1)
                {
                    var lezer = new BinaryFormatter(); (2)
                    Pizza pizza;
                    pizza = (Pizza)lezer.Deserialize(bestand); (3)
                    Console.WriteLine(pizza.Naam);
                    foreach (var onderdeel in pizza.Onderdelen)
                        Console.WriteLine(onderdeel);
                    Console.WriteLine(pizza.Prijs);
                }
            }
            catch (SerializationException) (4)
            {
                Console.WriteLine("Fout bij het deserialiseren");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

- (1) Creëer een *FileStream* object waarmee het gekoppelde bestand geopend wordt om het te lezen.
- (2) Om objecten van een bestand te kunnen inlezen (deserialiseren) heb je eveneens een **BinaryFormatter** object nodig. Dit object wordt hier gecreëerd.
- (3) Met de method *Deserialize()* van de class *BinaryFormatter* lees je een object in vanuit het bestand. Als parameter geef je aan deze method het *FileStream* object uit punt (1) mee.

Let op: deze method geeft het ingelezen object als een type **object** terug. Je moet dit object dus nog casten naar een *Pizza* object.

- (4) Ook bij het deserialisatieproces kunnen fouten optreden, die je best opvangt.

35.4.3 Meerdere objecten tegelijk wegschrijven en inlezen

Je kan meerdere objecten tegelijk wegschrijven naar eenzelfde bestand. Hierbij kan je de verschillende objecten aan een verzameling toevoegen en de verzameling in zijn geheel wegschrijven naar het bestand. Achteraf kan je deze verzameling in zijn geheel terug inlezen.

Voorbeeld:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Pizza> pizzas = new List<Pizza>
            {
                new Pizza
                {
                    Naam = "Pizza Margherita",
                    Onderdelen = new List<string> { "Tomatensaus", "Mozzarella" },
                    Prijs = 8m
                },
                new Pizza
                {
                    Naam = "Pizza Vegetariana",
                    Onderdelen = new List<string> { "Tomatensaus", "Mozzarella", "Groenten" },
                    Prijs = 9.5m
                }
            };

            try
            {
                //de verzameling wegschrijven...
                using (var bestand = File.Open(@"C:\Data\Pizzas.obj",
                                              FileMode.OpenOrCreate))
                {
                    var schrijver = new BinaryFormatter();
                    schrijver.Serialize(bestand, pizzas);
                }
                //en terug inlezen
                using (var bestand = File.Open(@"C:\Data\Pizzas.obj",
                                              FileMode.Open, FileAccess.Read))
                {
                    var lezer = new BinaryFormatter();
                    pizzas = (List<Pizza>)lezer.Deserialize(bestand);
                    foreach (var pizza in pizzas)
```

```
{  
    Console.WriteLine(pizza.Naam);  
    foreach (var onderdeel in pizza.Onderdelen)  
        Console.WriteLine(onderdeel);  
    Console.WriteLine(pizza.Prijs);  
}  
}  
}  
}  
catch (IOException)  
{  
    throw new Exception("Fout bij het openen van het bestand!");  
}  
catch (SerializationException)  
{  
    Console.WriteLine("Fout bij het serialiseren/deserialiseren");  
}  
catch (Exception ex)  
{  
    Console.WriteLine(ex.Message);  
}  
}  
}  
}
```



oefening: Files and Streams – I/O

36 Asynchrone methods- `async - await`

36.1 Asynchrone programmeren – wat en waarom?

Als gebruiker van computerprogramma's komt volgend beeld je wellicht bekend voor:



Je hebt ergens op geklikt om een actie te starten, en dan lijkt het alsof het programma "hangt": je kan niets meer doen, je wacht ... totdat de taak voltooid is. Pas dan kan je de applicatie verder gebruiken.

Sommige taken in een applicatie kunnen immers veel tijd in beslag nemen:

- grote bestanden downloaden of uploaden
- grote bestanden inlezen of wegschrijven
- afbeeldingen verwerken
- webservices oproepen
- ingewikkelde berekeningen
- ...

Bovendien worden programma's tegenwoordig vaak op afstand uitgevoerd i.p.v. op de eigen computer, en maken ze steeds vaker gebruik van gegevens die niet lokaal aanwezig zijn op het systeem waarop het programma draait. De gegevens worden gelezen uit databases of via een webservice opgehaald vanop een netwerklocatie zoals een intern netwerk, intranet of internet. De snelheid waarmee dergelijke taken uitgevoerd worden is dan afhankelijk van de snelheid van de netwerk- of internetverbinding en/of van de beschikbaarheid en de hoeveelheid van de gegevens. Als het ophalen van de gegevens daarbij door een externe partij beheerd wordt, heb je vaak geen enkele invloed op de responsietijd ervan.

Gebruikers vinden het echter zeer vervelend wanneer een applicatie lijkt te blokkeren en niet onmiddellijk reageert: ze denken al snel dat er niets gebeurt, ze klikken nog eens opnieuw, ... zeker met de moderne "touch" user interfaces van tegenwoordig. Gebruikers van de huidige applicaties en apps verwachten een snelle respons op hun handelingen en een hoge performantie.

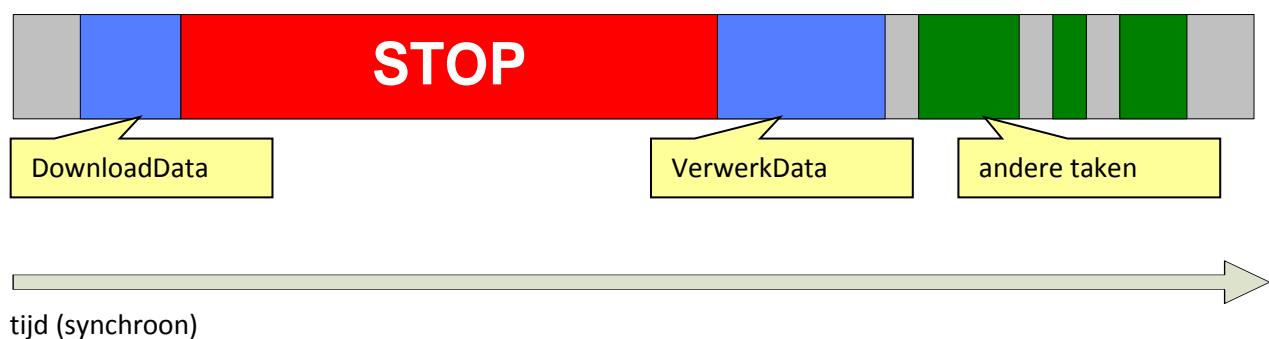
Moderne applicaties moeten snel, vloeiend en responsive zijn. Bij de ontwikkeling ervan moet hier dan ook meer en meer aandacht aan besteed worden.

36.1.1 Synchroon vs asynchroon

Tot hertoe heb je enkel programmacode geschreven die **synchroon** uitgevoerd wordt. Wanneer een method synchroon opgeroepen wordt, moet deze method volledig afgewerkt zijn vooraleer de oproepende method verder kan gaan. Dit veroorzaakt een blokkering van het programma bij langdurige operaties. Zo wordt bvb. bij een Windows Forms applicatie of een WPF applicatie de user interface bevroren bij het ophalen van een grote hoeveelheid gegevens en moet de gebruiker geduldig wachten totdat alle gegevens beschikbaar zijn. Ondertussen kan hij niets anders doen.

Volgend schema geeft dit weer in de tijd:

```
var data = DownloadData();
VerwerkData(data);
```



Zodra het ophalen van de gegevens gestart wordt (*DownloadData*), wordt het programma geblokkeerd en kunner er geen andere taken meer uitgevoerd worden zolang het ophalen van de gegevens duurt. Pas als alle gegevens beschikbaar zijn kan het programma deze gegevens verwerken (*VerwerkData*) en eventuele andere taken afhandelen.

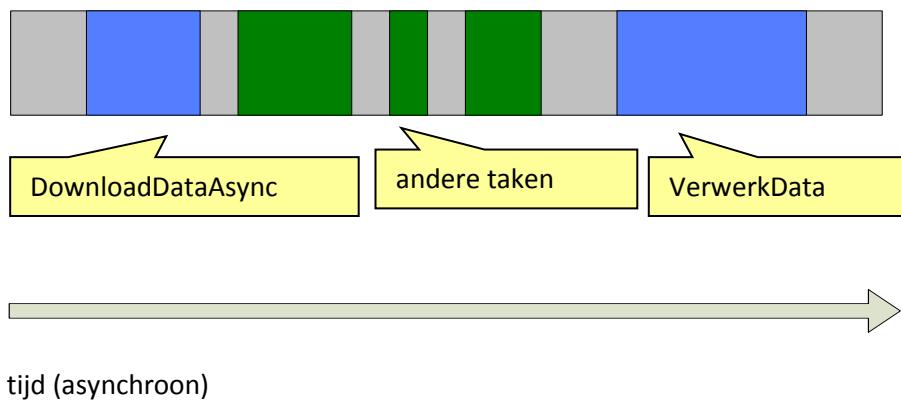
M.a.w. tijdens de synchrone uitvoering van een (langdurige) method, wordt de oproepende method geblokeerd en blijft de programmacontrole bij de opgeroepen method. Pas als deze volledig uitgevoerd is gaat de programmacontrole terug naar de oproepende method die dan verder uitgevoerd wordt.

Je kan lange responsietijden en bottlenecks in performantie vermijden door zogenaamde "**asynchrone code**" te schrijven.

Wanneer een method **asynchroon** opgeroepen wordt, wordt deze method gestart en uitgevoerd, maar ondertussen krijgt de oproepende method terug de programmacontrole zodat er wel andere taken kunnen uitgevoerd worden. Zo kan de gebruiker bvb. bij een Windows Forms applicatie of een WPF applicatie tijdens het ophalen van een grote hoeveelheid gegevens toch andere acties uitvoeren vanuit de gebruikersinterface.

Volgend schema geeft het verloop van bovenstaand programma in de tijd weer bij een asynchrone uitvoering ervan:

```
var data = DownloadDataAsync();
VerwerkData(data);
```



M.a.w. tijdens de asynchrone uitvoering van een (langdurige) method, wordt de oproepende method niet geblokkeerd. Ze krijgt na de start van de asynchrone method de programmacontrole terug en kan ondertussen eventuele andere taken uitvoeren.

	Asynchrone uitvoering van programmacode levert meestal een tijdsinst t.o.v. de synchrone uitvoering ervan.
--	--

36.2 Synchrone code

In het volgende voorbeeld worden de method *DoeIets()* synchroon opgeroepen. Deze method simuleert een langdurige method:

```
using System;
using System.Threading; //niet vergeten

namespace VbSyncConsole
{
    class Program
    {
        static void Main(string[] args) (1)
        {
            Console.WriteLine("method Main is gestart...");
            Start(); (2)
            Console.WriteLine("terug in method Main");
            Console.WriteLine("druk een willekeurige toets ...");
            Console.WriteLine();
            Console.ReadLine();
        }

        static void Start() (3)
        {
            string tekst = DoeIets();
        }
    }
}
```

```
    Console.WriteLine(tekst); (7)
}

static string DoeIets() (4)
{
    Console.WriteLine("Bezig met taak ...DoeIets");
    Thread.Sleep(5000); (5)
    return ("De taak is afgewerkt."); (6)
}
}
```

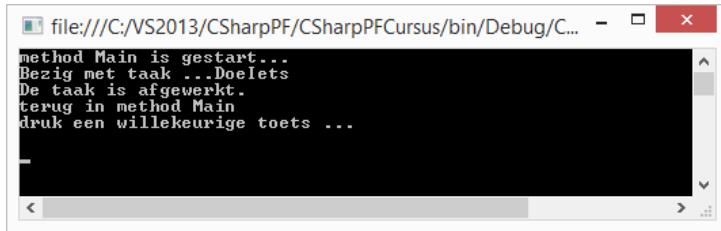
- (1) Het programma start uiteraard met de *Main()* method.
- (2) Vanuit de *Main()* wordt de method *Start()* synchroon opgeroepen. Wanneer een method synchroon opgeroepen wordt, moet deze method volledig uitgevoerd zijn vooraleer de programmacontrole terugkeert naar de oproepende method. M.a.w. de controle van het programma blijft in de method *Start()* totdat deze volledig uitgevoerd is.
- (3) Vanuit de method *Start()* wordt de method *DoeIets()* op een synchrone manier opgeroepen en krijgt *DoeIets()* de programmacontrole.
- (4) De method *DoeIets()* wordt uitgevoerd en behoudt de programmacontrole totdat ze volledig uitgevoerd is. Vermits deze een langdurige method simuleert, worden ondertussen de methods *Start()* en *Main()* geblokkeerd.
- (5) Om een langdurige method te simuleren wordt de method *DoeIets()* gedurende een vijftal seconden gepauzeerd (`Thread.Sleep(5000);`). Je stelt vast dat het programma inderdaad een bepaalde tijd blokkeert en dat er ondertussen niets gebeurt.



Wil je iets meer weten over threads, lees dan Bijlage 2: Processen en threads.

- (6) Pas daarna wordt de method *DoeIets()* verder afgewerkt.
- (7) *DoeIets()* is beëindigd, *Start()* wordt verder uitgevoerd, het resultaat van *DoeIets()* wordt getoond.
- (8) Zodra de method *Start()* beëindigd is, krijgt het hoofdprogramma (*Main()*) terug de controle en wordt het verder afgewerkt.

Het resultaat:



Bij de uitvoering van dit programma stel je inderdaad vast dat de *Main()* gepauzeerd wordt en pas verder uitgevoerd nadat *Doelets()* (en ook *Start()*) volledig uitgevoerd is.

In de volgende paragraaf wordt de synchrone method *Doelets()* omgevormd naar de asynchrone versie *DoeletsAsync()*.

36.3 Asynchrone code – async – await – Task

Zoals reeds eerder aangegeven wil een gebruiker bij bvb. het ophalen van een grote hoeveelheid gegevens liefst niet op het antwoord “wachten”. Hij wil ondertussen andere dingen kunnen doen en als alle gegevens opgehaald zijn, wil hij hiervan een seintje ontvangen zodat hij ze kan verwerken. Kortom, langdurige taken worden best **asynchroon** opgeroepen.

De mogelijkheid om asynchroon te programmeren is reeds lang beschikbaar in .NET d.m.v. het “asynchronous pattern” en het “event-based asynchronous pattern”.

Veel classes in het .NET Framework implementeren deze patterns, doch beide patterns zijn vrij moeilijk te programmeren:

- de programmaflow wordt door elkaar gehaald
- een asynchrone method moet opgesplitst worden in meerdere stukken code
- waar je in een synchrone method bvb. `using` kan gebruiken om gebruikte resources (bvb. een bestand) op te kuisen, moet je bij de asynchrone versie het sluiten van een bestand zelf schrijven.
- bovendien moet er extra code voorzien worden om bvb. de status van de aanroep naar de database bij te houden en het resultaat terug kunnen te communiceren
- ...

M.a.w. de code van een asynchrone method is tot nu toe een stuk ingewikkelder dan deze voor dezelfde synchrone method en dan spreken we nog niet over het afhandelen van Exceptions.

Vandaar dat de asynchrone mogelijkheden niet altijd toegepast werden.

Vanaf het .NET Framework 4.5 is er echter een nieuwe manier beschikbaar om asynchroon te programmeren. Je kan namelijk gebruik maken van de nieuwe sleutelwoorden **async** en **await** in combinatie met de class **Task**. Hierdoor wordt het schrijven van asynchrone methods veel eenvoudiger.

De nieuwe keywords **async** en **await** zorgen er voor dat de compiler de programmeur het meeste werk uit handen neemt.

Als voorbeeld schrijven we een asynchrone versie van bovenstaand voorbeeld:

```
using System;
using System.Threading.Tasks;

namespace VbAsyncConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("method Main is gestart..."); (1)
            Start(); (2)
            Console.WriteLine("terug in method Main"); (8)
            Console.WriteLine("druk een willekeurige toets als je het resultaat ziet ...");
            Console.WriteLine();
            Console.ReadLine();
        }

        static async void Start() (3)
        {
            Console.WriteLine("method Start is gestart..."); (4)
            string tekst = await DoeIetsAsync(); (5)
            Console.WriteLine(tekst); (10)
        }

        static async Task<string> DoeIetsAsync()
        {
            Console.WriteLine("Bezig met taak ... DoeIetsAsync"); (6)
            await Task.Delay(5000); (7)
            return "De taak is afgewerkt."; (9)
        }
    }
}
```

- (1) Het programma start uiteraard met de *Main()* method.
- (2) Vanuit de *Main()* wordt de method *Start()* **synchroon** opgeroepen.
- (3) Merk op dat de method *Start()* gemarkerd is met het sleutelwoord **async**. Het sleutelwoord **async** vertelt de compiler dat we in deze method het sleutelwoord **await** gaan gebruiken om een andere method – *DoeIetsAsync()* – **asynchroon** op te roepen. Door het sleutelwoord **await** bij de oproep van een method te vermelden, wordt deze method op een asynchrone manier uitgevoerd zonder dat er een blokkering van het programma optreedt.



Markeer je een method met *async*, maar gebruik je binnen deze method geen *await* voor de oproep van een andere langdurige method, dan wordt de huidige method toch

gewoon synchroon uitgevoerd, met een pauze als gevolg.

De compiler geeft je een waarschuwing:

```
static async void Start()
{
    Console.WritThis async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls.t
    Task<string> tekst = DoeIetsAsync();
    Console.WriteLine(tekst);
}
```

Door het sleutelwoord **await** zal de compiler immers de method *Start()* bij uitvoering opsplitsen in 2 delen:

- het deel vóór de *await* wordt gewoon synchroon uitgevoerd (4), de asynchrone method *DoeletsAsync()* wordt gestart.
- vanaf *await* wordt er ‘gewacht’ op het resultaat van de asynchrone operatie (5): alle code na de *await* opdracht kan pas uitgevoerd worden zodra de asynchrone method voltooid is. Het programmaverloop wordt echter niet geblokkeerd. M.a.w. terwijl de method *DoeletsAsync()* afgewerkt wordt, kunnen er andere opdrachten uitgevoerd worden, op voorwaarde dat deze uiteraard niet afhankelijk zijn van het resultaat van de asynchrone method.

- (6) De method *DoeletsAsync()* wordt uitgevoerd.

Met de code `await Task.Delay(5000);` (7)

wordt een langdurige method gesimuleerd. Merk op dat bij de oproep van *Task.Delay()* eveneens het sleutelwoord **await** gebruikt wordt om deze oproep asynchroon te laten gebeuren. Vandaar dat bij de method *DoeletsAsync()* ook het sleutelwoord **async** vermeld is.

Zolang de langdurige method *Task.Delay()* niet afgewerkt is, kan de method *DoeletsAsync()* het resultaat niet teruggeven aan de method *Start()*. Maar 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 *DoeletsAsync()*, die met **await** opgeroepen werd, en dus ook niet verder afgewerkt kan worden, gaat de programmacontrole terug naar de method *Main()*.

- (8) Terwijl de langdurige method uitgevoerd wordt, heeft de *Main()* terug de programmacontrole en wordt deze verder afgewerkt.

- (9) Zodra *Task.Delay()* afgewerkt is, wordt ook de method *DoeletsAsync()* verder uitgevoerd en wordt het resultaat aan de method *Start()* teruggegeven.

- (10) De variabele *tekst* bevat nu het resultaat van *DoeletsAsync()*.

Start() wordt verder uitgevoerd, de inhoud van *tekst* wordt afgebeeld.

Merk op dat het hoofdprogramma ondertussen reeds volledig afgewerkt werd.

Het resultaat:

Bij de uitvoering van dit programma stel je inderdaad vast dat de *Main()* niet gepauzeerd wordt maar verder afgewerkt wordt tijdens de uitvoering van de asynchrone method. Het resultaat van de asynchrone method wordt pas later getoond.

Om te onthouden:

- Bij overeenkomst eindigt de naam van een asynchrone method met het suffix **Async**: `DoeIetsAsync()`.
Zo herkent de programmeur de methods die asynchroon opgeroepen kunnen worden.
- Een asynchrone method wordt gemarkeerd met het sleutelwoord **async**
`static async Task<string> DoeIetsAsync()`
- Een method die een asynchrone method oproeft, moet zelf gemarkeerd worden met het sleutelwoord **async**.
Bij de *Main()* method mag dit sleutelwoord niet vermeld worden!
Dit geeft een compilerfout:

```
static async void Main(string[] args)
{
    Console.WriteLine("DoeIetsAsync() is gestart");
    string tekst = await DoeIetsAsync();
```



Vandaar dat in dit voorbeeld de method *Start()* toegevoegd werd om de asynchrone method *DoeletsAsync ()* op te roepen.

- **Een asynchrone method moet als returntype `Task<TResult>` of `Task` hebben** (void kan ook, best enkel bij event handlers):
`static async Task<string> DoeIetsAsync()`

De *Task* class is geïntroduceerd in het .Net Framework 4.0 en een *Task* object representeert een asynchrone method/operatie in uitvoering.

De *Task* class bevat methods en properties om de status van de operatie te monitoren, te wachten totdat de operatie is afgelopen en om te bepalen wat er gebeurt als de operatie afgelopen is. Aan de hand van het *Task* object kan er bvb. nagegaan worden of de method (correct) beëindigd is, of er een fout opgetreden is bij de uitvoering, of de method gecanceld is, ...

Bij `Task<TResult>` stelt de generic type parameter `TResult` het eigenlijke type voor dat door de method teruggegeven wordt, hier dus `string`. Vandaar dat de method *DoeletsAsync()* een `Task<string>` object teruggeeft.

`Task` wordt als returntype gebruikt wanneer de asynchrone method geen

returnwaarde heeft (een procedure met als returntype void), zoals bij `await Task.Delay(5000);`

Je kan het zo stellen:

Een asynchrone method creëert tijdens zijn uitvoering een `Task` object dat ervoor zorgt dat de method zo uitgevoerd wordt dat er geen blokkering van het programma optreedt: de CLR creëert een nieuwe execution thread om de taak van de method af te handelen, maar de controle wordt wel teruggegeven aan de oproepende code.

Als de asynchrone method beëindigd is wordt het `Task` object al dan niet met een returnwaarde teruggegeven aan de oproepende method.

- Merk echter op dat deze returnwaarde opgevangen wordt in de variabele `tekst`, die zelf van het type `string` is en niet van het type `Task<string>`.

Het sleutelwoord `await` zorgt ervoor dat er automatisch een casting gebeurt van het `Task<string>` object naar een `string` object.

Wanneer de asynchrone method volledig uitgevoerd is, bevat het `Task<string>` object namelijk in zijn `Result` property de eigenlijke returnwaarde (de `string`) van de asynchrone method. De `await` haalt deze string op uit het `Task` object.

Je zou de code ook als volgt kunnen schrijven:

```
static async void Start()
{
    Console.WriteLine("method Start is gestart...");
    Task<string> asyncMethod = DoeIetsAsync();
    Console.WriteLine(await asyncMethod);
}
```

- Ook lambda expressies en anonymous methods kunnen met `async` gemarkerd worden zodat ze asynchroon uitgevoerd kunnen worden.
- Een asynchrone method, lambda expressie of anonymous method zal pas asynchroon uitgevoerd worden vanaf een `await` statement. Zonder `await` worden deze methods synchroon uitgevoerd.
- Een method gemarkerd met `async` kan meerdere `await` statements bevatten.

36.4 Asynchrone methods in het .NET Framework

Zoals reeds aangegeven zijn de huidige moderne applicaties vaak ‘connected’ applicaties die gebruik maken van diensten op afstand via het internet, webservices, de cloud, ... en daarbij vlot, vloeiend en responsive moeten zijn.

Asynchroon programmeren wordt dan ook meer en meer de norm bij de ontwikkeling van deze applicaties.

Het .NET Framework bevat reeds een groot aantal asynchrone methods. Je herkent deze methods aan het **suffix Async** in de naam en aan het returntype **Task<TResult>** of **Task**.

Zo zijn er o.a. asynchrone versies van een aantal methods voor het lezen en schrijven naar bestanden in de namespace System.IO:

Class *StreamReader* (zie MSDN):

ReadLine	Reads a line of characters from the current stream and returns the data as a string. (Overrides TextReader.ReadLine() .)
ReadLineAsync	Reads a line of characters asynchronously from the current stream and returns the data as a string. (Overrides TextReader.ReadLineAsync() .)
ReadToEnd	Reads all characters from the current position to the end of the stream. (Overrides TextReader.ReadToEnd() .)
ReadToEndAsync	Reads all characters from the current position to the end of the stream asynchronously and returns them as one string. (Overrides TextReader.ReadToEndAsync() .)

Let op de returntypes van de methods:

`ReadLine(): string`

`ReadLineAsync(): Task<string>`

Class *StreamWriter* (zie MSDN)

WriteLineAsync()	Writes a line terminator asynchronously to the stream. (Overrides TextWriter.WriteLineAsync() .)
WriteLineAsync(Char)	Writes a character followed by a line terminator asynchronously to the stream. (Overrides TextWriter.WriteLineAsync(Char) .)
WriteLineAsync(Char[])	Writes an array of characters followed by a line terminator asynchronously to the text string or stream. (Inherited from TextWriter .)
WriteLineAsync(String)	Writes a string followed by a line terminator asynchronously to the stream. (Overrides TextWriter.WriteLineAsync(String) .)
WriteLineAsync(Char[], Int32, Int32)	Writes a subarray of characters followed by a line terminator asynchronously to the stream. (Overrides TextWriter.WriteLineAsync(Char[], Int32, Int32) .)

Returntypes:

`WriteLine(): void`

`WriteLineAsync(): Task`

Bij Windows RT, een versie van Windows 8 voor tablets, zijn er zelfs enkel asynchrone versies beschikbaar van die methods waarvan verwacht wordt dat ze langer dan 40 ms zal duren.

37 De .NET softwarelibrary

37.1 Algemeen

Microsoft levert bij .NET veel interessante classes mee.

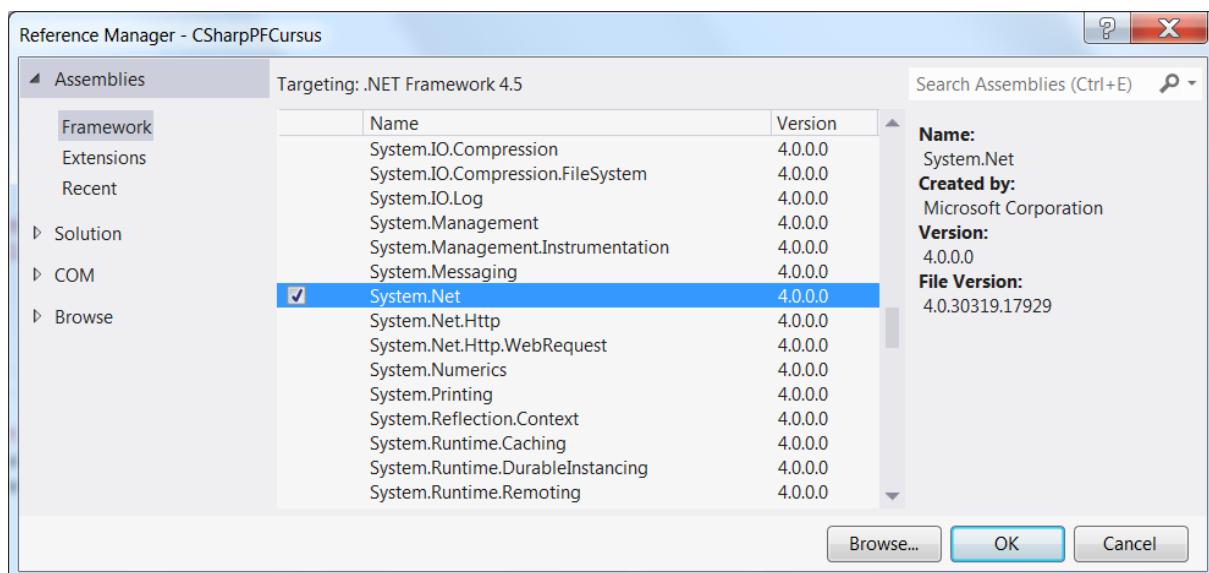
Deze verzameling classes (de .NET Framework class library) is ingedeeld in namespaces.

Hieronder enkele van de belangrijkste namespaces:

- **System.Data**
Classes om databases (SQL Server, Oracle, Access, ...) te gebruiken.
- **System.XML**
Classes om XML (uitwisselbaar dataformaat tussen de meest voorkomende computers en programmeertalen) te gebruiken.
- **System.Diagnostics**
Classes om fouten in je programma op te sporen.
- **System.DirectoryServices**
Classes om Active Directory Services te gebruiken.
- **System.ServiceProcess**
Classes om services van Windows NT, 2000, of XP te gebruiken.
- **System.Messaging**
Classes om boodschappen te lezen en te verzenden over netwerken.
- **System.Timers**
Classes om op geregelde tijdstippen (iedere dag, om de minuut, ...) code uit te voeren.
- **System.Globalization**
Classes om taal- en cultuurverschillen in je toepassing te verwerken
(wij schrijven bvb. een datum als dag/maand/jaar, in de USA is dit maand/dag/jaar).
- **System.Net**
Classes om netwerken, het internet en hun protocollen te gebruiken.
- **System.Collections**
Classes om verzamelingen te maken die meer functionaliteit aanbieden dan arrays: lists, queues, arraylists, hashtables, dictionaries.
- **System.Collections.Generic**
Zelfde als bij System.Collections maar dan voor generic lists.
- **System.IO**
Classes om bestanden en directories te gebruiken.
- **System.Text**
Classes om verschillende karaktercoderingen (ANSII, Unicode, ...) te gebruiken.
- **System.Threading**
Classes om multithreading (gelijktijdige uitvoering van code) in je toepassing te gebruiken.
- **System.Reflection**
Classes om tijdens de uitvoering van het programma de eigenschappen en methods van Classes op te vragen.
- **System.Drawing**
Classes om punten, lijnen, rechthoeken, cirkels, afbeeldingen, tekst, ... te tekenen.
- **System.Windows.Forms**
Classes om Windows vensters in je programma te gebruiken met in die vensters alle populaire controls: tekstvakken, knoppen, menu's, ...

- System.Runtime.Remoting
Classes om gedistribueerde programma's te ontwikkelen (programma's die op meerdere PC's samenwerken).
- System.Web
Classes om internet- en intranetapplicaties te maken (gegevens naar een browser sturen, gegevens die een gebruiker in de browser intikt (bvb. een bestelling in je webwinkel) verwerken).
- System.Linq
Classes om LINQ (Language Integrated Query) query's te gebruiken. LINQ is een technologie waarmee je op een universele manier (met dezelfde basis syntax) met gegevens kan werken, ongeacht de bron van deze gegevens (een array, een List, een relationele database, een XML bestand, ...).

Om sommige van deze namespaces te kunnen gebruiken, moet je eerst een reference leggen naar de DLL-file waarin deze namespaces gedefinieerd zijn. Dit gebeurt door het menu *PROJECT - Add Reference* te kiezen (of in de Solution Explorer met de rechtermuisknop op de map References van het project te klikken en *Add Reference...* te kiezen). Je plaatst een vinkje voor de DLL die je wil toevoegen en klikt op OK:



38 Bijlage 1: Debugging

Bij het programmeren maak je wel eens fouten. In deze bijlage worden een aantal debugging tools van VS.NET beschreven, die de programmeur helpen bij het opsporen van fouten.

38.1 Soorten fouten

Afhankelijk van het type fout dat je maakt, zijn er andere hulpmiddelen ter beschikking om de fout op te sporen en op te lossen. Het is daarom belangrijk om een onderscheid te maken tussen de soorten fouten:

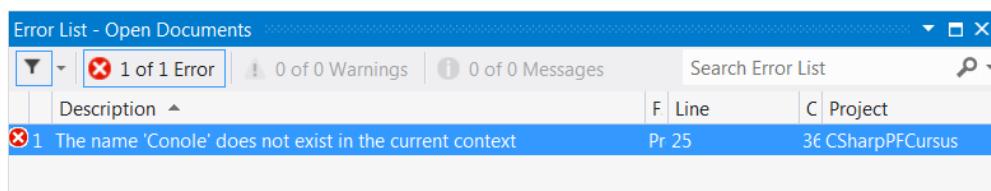
- **Programmacode fouten**

Deze fouten zijn een gevolg van onjuiste programmacode of syntaxfouten. Je hebt bvb. een sleutelwoord verkeerd ingetypoed of je bent een accolade vergeten te typen om een bepaalde programmeerstructuur af te sluiten, Deze fouten worden door de IDE onmiddellijk gesigneerd d.m.v. een rode gegolfde lijn in de code. Als je de muisaanwijzer laat rusten op deze rode gegolfde lijn, krijg je een hint over de fout.

```
Console.WriteLine("Geef een eerste getal: ");
int getal1 = int.Parse(Console.ReadLine());
Console.WriteLine("Geef een 
```

Dergelijke fouten worden ook getoond in het *Error List* venster onderaan het scherm. Dit venster wordt automatisch getoond wanneer de IDE fouten ontdekt bij het compileren van de programmacode. Via het menu *VIEW – Error List* kan je eventueel zelf het *Error List* venster zichtbaar maken.

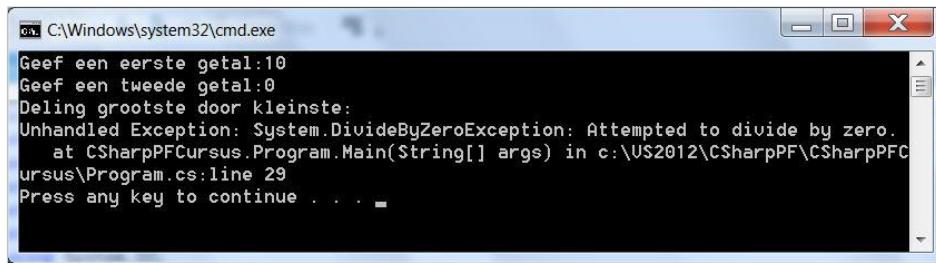
Door dubbel te klikken op het rode symbooltje naast de foutmelding, springt de cursor naar de plaats van de fout in de code. Deze fouten moeten eerst verbeterd worden vooraleer de code kan uitgevoerd worden.



- **Runtime fouten**

Deze fouten treden op wanneer het programma een instructie probeert uit te voeren die onmogelijk is. Alhoewel de syntax van de code correct is, kunnen er tijdens de uitvoering van de code fouten optreden. De uitvoering van het programma wordt dan onmiddellijk gestopt en er wordt door de IDE een Exception getoond.

Voorbeelden van runtime fouten zijn: deling door nul, het programma probeert een bestand te openen dat niet beschikbaar is, ...



Het is de verantwoordelijkheid van de programmeur om dergelijke runtime fouten te voorkomen of op een gepaste wijze af handelen.

De programmeur kan instructies inbouwen die nagaan of de betreffende instructie correct kan uitgevoerd worden. Bvb. bij deling door nul:

```
namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int grootste, kleinste;
            Console.Write("Geef een eerste getal: ");
            int getal1 = int.Parse(Console.ReadLine());
            Console.Write("Geef een tweede getal: ");
            int getal2 = int.Parse(Console.ReadLine());
            Console.WriteLine("Deling grootste door kleinste: ");
            if (getal1 > getal2)
            {
                grootste = getal1;
                kleinste = getal2;
            }
            else
            {
                grootste = getal2;
                kleinste = getal1;
            }
            if (kleinste != 0)
                Console.WriteLine(grootste / kleinste);
            else
                Console.WriteLine("Delen door nul kan niet!");
        }
    }
}
```

De programmeur kan instructies voorzien die de fout opvangen, een gepaste boodschap aan de gebruiker tonen en het programma correct afsluiten (zie hoofdstuk EXCEPTIONS).

- **Logische fouten**

Alhoewel de syntax correct is en er geen runtimefouten optreden, voert het programma toch onjuiste bewerkingen uit en blijken de resultaten niet correct te zijn. Waar zit dan de fout? Je kan deze fouten opsporen door gebruik te maken van de tools die de ontwikkelomgeving voorziet. Een aantal hulpmiddelen zijn ter beschikking.

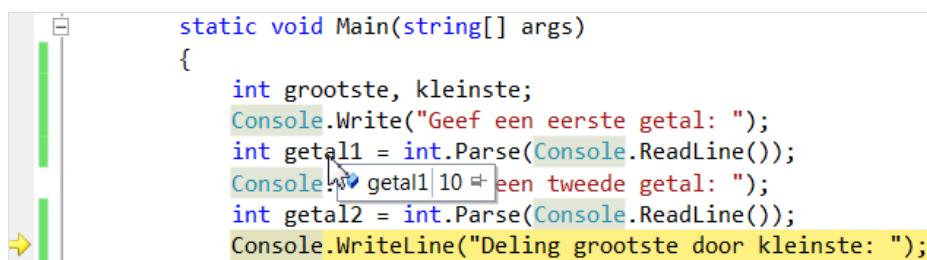
38.2 Een programma stap voor stap uitvoeren – Step Into

Als je niet weet waar de code precies fout loopt, kan je de code stap voor stap laten uitvoeren.

Je start hiervoor het programma niet op de klassieke manier, maar wel via het menu **DEBUG – Step Into** of met de functietoets **F11** of met de knop  van de Debug werkbalk.

Hierdoor start het programma maar houdt het onmiddellijk halt bij de eerste accolade van de *Main* method. Deze coderegel wordt geel gemarkerd en voorzien van een geel pijltje in de marge.

Wanneer je deze actie opnieuw uitvoert, wordt de gemarkerde coderegel uitgevoerd en houdt het programma halt bij de volgende coderegel. Je kan dit blijven herhalen totdat je het hele programma stap voor stap doorlopen hebt. Tijdens de *Step Into* mode kan je de huidige waarde van een variabele bekijken door de muisaanwijzer op deze variabele te laten rusten:



```
static void Main(string[] args)
{
    int grootste, kleinste;
    Console.Write("Geef een eerste getal: ");
    int getal1 = int.Parse(Console.ReadLine());
    Console.WriteLine("Geef een tweede getal: ");
    int getal2 = int.Parse(Console.ReadLine());
    Console.WriteLine("Deling grootste door kleinste: ");
```

Wordt er in de code een andere method opgeroepen, dan kan je op dezelfde manier de code van de opgeroepen method stap voor stap doorlopen.

Wil je dit echter niet en heb je liever dat de opgeroepen method meteen volledig uitgevoerd wordt, dan kan je op de volgende manieren te werk gaan:

- De cursor bevindt zich nog niet in de opgeroepen method

Kies dan het menu **DEBUG – Step Over** of de functietoets **F10** of de knop  van de Debug werkbalk. De opgeroepen method wordt uitgevoerd maar niet stap voor stap.

```

class Program
{
    static void Main(string[] args)
    {
        int grootste, kleinste;
        Console.Write("Geef een eerste getal: ");
        int getal1 = int.Parse(Console.ReadLine());
        Console.Write("Geef een tweede getal: ");
        int getal2 = int.Parse(Console.ReadLine());
        Console.WriteLine("Deling grootste door kleinste: ");
        if (getal1 > getal2)
        {
            grootste = getal1;
            kleinste = getal2;
        }
        else
        {
            grootste = getal2;
            kleinste = getal1;
        }

        //method oproep
        TekenLijn('*');

        if (kleinste != 0)
            Console.WriteLine(grootste / kleinste);
        else
            Console.WriteLine("Delen door nul kan niet!");
    }

    public static void TekenLijn(char teken)
    {
        for(int i = 1; i <= 25;i++)
            Console.Write(teken);
        Console.WriteLine();
    }
}

```

- De cursor bevindt zich reeds in de opgeroepen method

Kies dan het menu **DEBUG – Step Out** of de toetsencombinatie **Shift+F11** of de knop van de *Debug* werkbalk. De opgeroepen method wordt verder uitgevoerd maar niet stap voor stap. De cursor springt uit de opgeroepen method en keert terug naar de plaats waar de betreffende method opgeroepen werd.

```

namespace CSharpPFCursus
{
    class Program
    {
        static void Main(string[] args)
        {
            int grootste, kleinste;
            Console.Write("Geef een eerste getal: ");
            int getal1 = int.Parse(Console.ReadLine());
            Console.Write("Geef een tweede getal: ");
            int getal2 = int.Parse(Console.ReadLine());
            Console.WriteLine("Deling grootste door kleinste: ");
            if (getal1 > getal2)
            {
                grootste = getal1;
                kleinste = getal2;
            }
            else
            {
                grootste = getal2;
                kleinste = getal1;
            }

            //method oproep
            TekenLijn('*');

            if (kleinste != 0)
                Console.WriteLine(grootste / kleinste);
            else
                Console.WriteLine("Delen door nul kan niet!");
        }

        public static void TekenLijn(char teken)
        {
            for(int i = 1; i <= 25;i++)
                Console.Write(teken);
            Console.WriteLine();
        }
    }
}

```

	Zowel met <i>Step Into</i> (<i>F11</i>) als met <i>Step Over</i> (<i>F10</i>) kan je de code stap voor stap uitvoeren. Wordt er in de code een method opgeroepen dan kan je met <i>Step Into</i> (<i>F11</i>) de code van deze method ook stap voor de stap doorlopen. Met <i>Step Over</i> (<i>F10</i>) wordt de method opgeroepen en uitgevoerd maar niet stap voor stap.
---	---

Je kan de *Step Into* mode op elk ogenblik onderbreken door het programma op de gewone manier verder te laten lopen (knop ) of de uitvoer van het programma te beëindigen (knop ).

38.3 Breakpoints

Als je min of meer weet waar de fout zich voordoet in de code, kan je in de buurt ervan – vóór de plaats van de fout – een zogenaamd **breakpoint** plaatsen. Dit doe je door in de marge van de coderegel te klikken, waardoor er een rode stip in de marge verschijnt.

Wanneer je nu het programma start met de functietoets **F5** of via het menu **DEBUG – Start Debugging** of met de knop  , wordt het programma uitgevoerd tot aan dit breakpoint. Vanaf het breakpoint kan je het programma verder stap voor stap uitvoeren met *F11* of met *F10*. Het voordeel is dat je het programma niet volledig stap voor stap moet doorlopen.

Je verwijdert een breakpoint door de rode stip in de marge opnieuw aan te klikken.

	<ul style="list-style-type: none"> Je kan meerdere breakpoints in de code plaatsen. Een gelijkaardige manier van werken biedt de optie Run To Cursor (toetsencombinatie Ctrl+F10). Wanneer je met de rechtermuisknop ergens in de code klikt en vervolgens de optie Run To Cursor kiest, wordt het programma op de normale manier uitgevoerd tot aan de coderegel waar de cursor zich bevindt (tenzij er eerder een breakpoint voorkomt). Vanaf daar kan je het programma verder stap voor stap uitvoeren.
---	---

38.4 Het Locals venster

Wanneer je een programma stap voor stap doorloopt, kan je het zogenaamde Locals venster op het scherm tonen via het menu **DEBUG – Windows – Locals**.

Dit **Locals** venster toont je een lijst van alle variabelen die op dat ogenblik gebruikt worden, met hun inhoud.

Locals		
Name	Value	Type
args	{string[0]}	string[]
grootste	0	int
kleinste	0	int
getal1	10	int
getal2	0	int

Autos | Locals | Watch 1

38.5 Het Watch venster

Wanneer je een programma stap voor stap doorloopt, kan je een zogenaamd *Watch* venster op het scherm tonen via het menu **DEBUG – Windows – Watch – Watch 1**.

Hier kan je zelf aangeven welke variabelen of expressies je wil opvolgen/evalueren tijdens het debuggen van het programma. Je voegt de namen van de variabelen of de expressies toe aan het *Watch* venster d.m.v. een rechtermuisklik.

Watch 1		
Name	Value	Type
getal1	10	int
grootste	0	int
getal1>getal2	true	bool

Autos | Locals | Watch 1

Bij het stap voor stap doorlopen van het programma zie je nu hoe de waarden van deze variabelen/expressies wijzigen tijdens het programmaverloop.

38.6 Het Call Stack venster

Wanneer je een programma stap voor stap doorloopt, kan je het *Call Stack* venster op het scherm tonen via het menu **DEBUG – Windows – Call Stack**.

Dit venster toont een lijst van de methods die op dit ogenblik in uitvoering zijn.

Call Stack	
Name	Lang
CSharpPFCursus.exe!CSharpPFCursus.Program.Main(string[] args = {string[0]}) Line 30	C#
[External Code]	
Call Stack Breakpoints Command Window Immediate Window Output	

De method die met een gele pijl gemarkeerd is, is de method waar de cursor zich op dit ogenblik bevindt. Het is ook de method waarvan de variabelen op dit ogenblik in het *Locals* venster en in het *Watch* venster gevolgd kunnen worden.

Door in het *Call Stack* venster dubbel te klikken op een andere method, kan je in het *Locals* of *Watch* venster de variabelen van deze andere method bekijken.



In de documentatie van VS.NET vind je uitgebreide informatie over de verschillende debugging tools.

39 Bijlage 2: PROCESSEN en THREADS

39.1 Processen en threads

39.1.1 Proces

Een proces is een programma in uitvoering.

Een tekstverwerker en een rekenbladprogramma die je startte, zijn dus 2 processen.

Elk proces heeft zijn eigen interne geheugenruimte. Een proces kan niet lezen of schrijven in de geheugenruimte van een ander proces.

39.1.2 Thread

Een thread is het uitvoeren van code binnen een proces.

Ieder proces heeft minstens één thread.

Een proces kan meerdere threads hebben. Dit heet multithreading: het tegelijk uitvoeren van verschillende code binnen een proces. Een browser is bijvoorbeeld een multithreaded programma: je kan een groot bestand downloaden en tegelijk surfen naar andere pagina's. De browser voert het downloaden uit met een thread, en het surfen met een andere thread. Je kan zelfs meerdere bestanden tegelijk downloaden. De browser voert deze extra taken uit met extra threads.

Alle threads binnen een proces delen de geheugenruimte van dat proces.

Naast multithreading bestaat ook het woord multiprocessing.

Dit betekent het gelijktijdig uitvoeren van meerdere processen (applicaties).

Multithreading en multiprocessing vormen één groot geheel:

de computer voert meerdere gelijktijdige processen uit, die zelf één of meerdere gelijktijdige threads uitvoeren.

39.2 Het verdelen van threads over processoren

Als een computer minstens evenveel processoren bevat als het aantal uit te voeren threads, verdeelt de computer de threads over de processoren. Iedere processor voert een thread uit. Gelijktijdig voeren de andere processoren een andere thread uit.

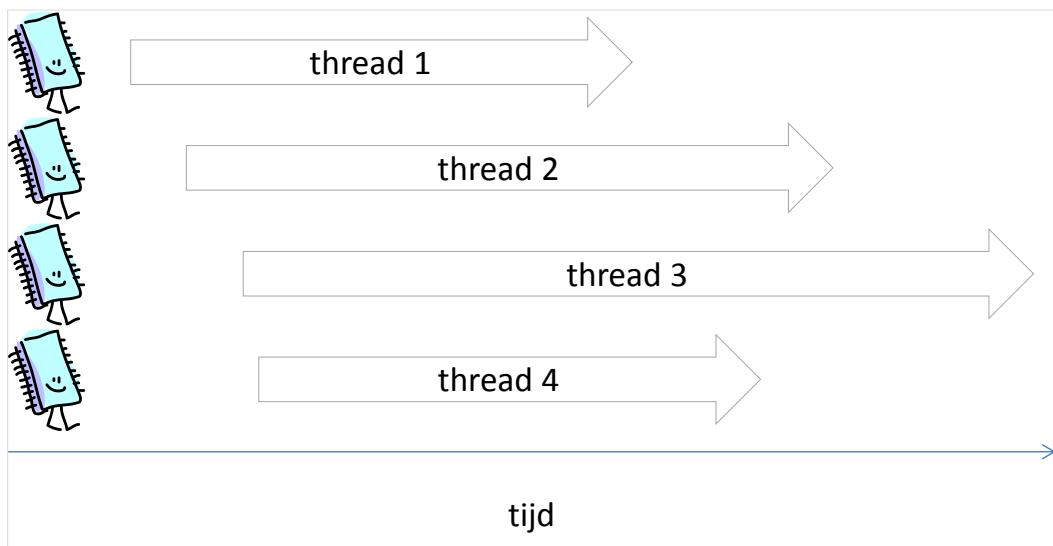
Dit leidt tot een optimale performantie.

Voorbeeld: een computer met vier processoren moet twee processen uitvoeren, die elk twee threads uitvoeren. De computer moet dus in totaal vier threads uitvoeren. Iedere processor voert één van deze threads uit...

Opmerking:

deze threads hoeven niet op hetzelfde moment te starten, en de uitvoeringstijd hoeft niet even lang te zijn.

In de volgende afbeelding staan links de vier processoren, en daarnaast de thread die ze gelijktijdig uitvoeren.



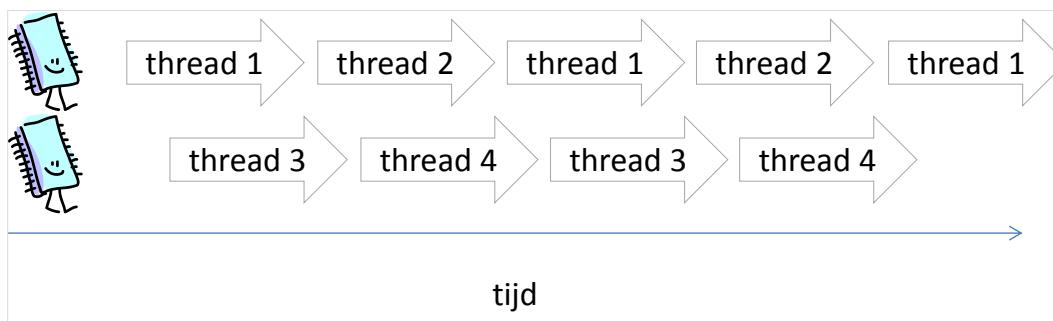
Als de computer minder processoren bevat dan het aantal uit te voeren threads, verdeelt het besturingssysteem de threads over de processoren.

Als er twee processoren zijn en vier threads, voert de ene processor twee threads uit en de andere processor de overige twee threads.

Een processor kan echter op een bepaald moment maar één thread uitvoeren.

Om dit probleem op te lossen gebruikt het besturingssysteem timeslicing:

de processor voert een aantal milliseconden code uit van de eerste thread en zet dan deze thread op pauze. De processor voert daarna een aantal milliseconden code uit van de tweede thread en zet dan deze thread op pauze. De processor voert terug een aantal milliseconden code uit van de eerste thread en zet dan deze thread op pauze, ...



Een .NET applicatie heeft minstens één thread: de thread die de code uitvoert die begint bij **static void Main(string[] args)**

40 Colofon

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	Mariëlla Cleuren
Auteurs	Hans Desmet Mariëlla Cleuren
Versie	30/11/2014

Omschrijving module-inhoud

Abstract	Doelgroep	.NET ontwikkelaar met C#
	Aanpak	Begeleide zelfstudie
	Doelstelling	De basissyntax van C# leren kennen om objectgeoriënteerde console-applicaties te ontwerpen
Trefwoorden		C#
Bronnen/meer info		http://msdn.microsoft.com