

03/02/2020

**GRADUAAT IN HET PROGRAMMEREN**

**SEMESTER 2**

**ACADEMIEJAAR 2019-2020**

**LECTOREN DRIES DEBOOSERE, STEF VERCAEMER, MAXIM LESY,  
DIETER MOURISSE**

# PROGRAMMING BASICS

Graduaat Programmeren

---

**howest.be**



# **HOOFDSTUK 1**

## **INLEIDING**



## INHOUDSOPGAVE

<b>1</b>	<b>INLEIDING</b>	<b>7</b>
<b>1.1</b>	<b>Over deze syllabus</b>	<b>7</b>
<b>2</b>	<b>OVER DE PROGRAMMEERTAAL</b>	<b>8</b>
<b>2.1</b>	<b>C#</b>	<b>8</b>
<b>2.2</b>	<b>.NET</b>	<b>8</b>
2.2.1	.NET Land, population: 3	8
2.2.2	.NET Framework	9
2.2.3	Hoe het werkt	9
<b>2.3</b>	<b>Programmeren is analyseren</b>	<b>9</b>
<b>2.4</b>	<b>Belangrijke begrippen</b>	<b>10</b>
2.4.1	Assembly	10
2.4.2	.NET	10
2.4.3	Compiler	10
2.4.4	Design-time	10
2.4.5	Compile-time	10
2.4.6	Run-time	10
<b>3</b>	<b>VISUAL STUDIO</b>	<b>11</b>
<b>3.1</b>	<b>Installatie</b>	<b>11</b>
<b>3.2</b>	<b>Rondleiding Visual Studio</b>	<b>12</b>
3.2.1	Menu en toolbars	13
3.2.2	Windows	13
3.2.3	Extensions and updates	14
3.2.4	Projecten en solutions	14
<b>4</b>	<b>.NET PROJECTEN</b>	<b>16</b>
<b>4.1</b>	<b>Console applicatie</b>	<b>16</b>
4.1.1	Een nieuw project maken	16
4.1.2	Code, Build, Test, Repeat	19
4.1.3	Build output	24
<b>4.2</b>	<b>Windows Presentations Forms (WPF)</b>	<b>25</b>
4.2.1	Een project toevoegen aan een solution	25
4.2.2	Control eigenschappen wijzigen	28
<b>4.3</b>	<b>Webapplicatie</b>	<b>29</b>
4.3.1	Een webapplicatie project maken	29

<b>4.4</b>	<b>De broncode bundelen</b>	<b>32</b>
<b>5</b>	<b>EEN EERSTE WPF APPLICATIE</b>	<b>34</b>
<b>5.1</b>	<b>Project aanmaken</b>	<b>34</b>
<b>5.2</b>	<b>GUI elementen toevoegen en bewerken</b>	<b>35</b>
<b>5.3</b>	<b>Events afhandelen</b>	<b>39</b>
<b>6</b>	<b>WPF REFERENTIEMATERIAAL</b>	<b>42</b>
<b>6.1</b>	<b>Solutions en Projecten</b>	<b>42</b>
<b>6.2</b>	<b>Referentiemateriaal</b>	<b>42</b>

# 1 INLEIDING

## 1.1 OVER DEZE SYLLABUS

---

Deze syllabus leert je programmeren met het .NET Framework aan de hand van de programmeertaal C# en WPF.

Elk hoofdstuk behandelt een sleutelaspect van het programmeren aan de hand van voorbeelden en oefeningen. Tegelijkertijd leer je de basis voor het maken van een eenvoudige desktopapplicatie voor Windows met behulp van WPF.

Beperk je niet enkel tot het bestuderen van de codevoorbeelden die in deze syllabus staan, maar **experimenteer zelf** met eigen mini-projecten om zo tot een beter begrip van het geheel te komen.



Tijdens de lessen worden belangrijke aspecten uitgediept met deze syllabus als leidraad. Er wordt verwacht dat je deze syllabus zelfstandig doorneemt en de contactmomenten met de lector gebruikt om specifieke vragen te stellen.

## 2 OVER DE PROGRAMMEERTAAL

### 2.1 C#

Microsoft Visual C Sharp (zeg maar C#) is een moderne object-georiënteerde programmeertaal. Het is een zorgvuldig ontworpen taal met heel wat ingebouwde mogelijkheden en is zeer uitgebreid gedocumenteerd. Zoals bij elke hogere programmeertaal is C# zeer goed leesbaar voor een mens.



De programmeertaal werd oorspronkelijk ontworpen om te werken met Microsoft's .NET platform, en is een rechtstreeks concurrent met Java. Inmiddels heeft C# het klassieke .NET platform oversteegen en kan het ook gebruikt worden in het nieuwere, open source .NET Core en Mono, zodat talloze platforms (Windows, Linux, Mac, iOS, Android, spelconsole, enz.) ondersteund worden.

### 2.2 .NET

#### 2.2.1 .NET LAND, POPULATION: 3

.NET (uitspraak: *dot net*) kent verschillende frameworks, die los van elkaar beschouwd kunnen worden. Als C# ontwikkelaar heb je **drie** smaken om uit te kiezen. Wanneer je informatie zoekt op het internet is het belangrijk om goed te weten voor welk Framework de informatie van toepassing is.

- **.NET Framework**  
Dit is het oorspronkelijke, traditionele framework dat geleverd wordt met Windows. Het wordt voornamelijk gebruikt voor de ontwikkeling van Windows applicaties of standaard ASP.NET 4.x webapplicaties voor IIS, de webserver voor Windows systemen. **Deze syllabus behandelt voornamelijk dit Framework.**
- **.NET Core**  
Een cross platform versie van het oorspronkelijke framework waarmee applicaties gebouwd kunnen worden die op verschillende systemen kunnen uitgevoerd worden: Windows, Linux, Mac en Docker.
- **Xamarin (Mono.Net)**  
Dit framework wordt voornamelijk gebruikt voor de ontwikkeling van mobiele apps. Het is gebaseerd op Mono.NET, de oorspronkelijke open source versie dat het .NET Framework volledig tracht na te bouwen.

.NET FRAMEWORK	.NET CORE	XAMARIN
Platform for .NET applications on Windows	Cross-platform and open source framework optimized for modern app needs and developer workflows	Cross-platform and open source Mono-based runtime for iOS, OS X, and Android devices
Distributed with Windows	Distributed with app	Distributed with app

Voor applicaties gebouwd voor het .NET Framework is het vereist om dat Framework vooraf te hebben geïnstalleerd, dat is vergelijkbaar met Java. Voor .NET Core en Xamarin is dat niet zo. De benodigde bibliotheken kunnen met de applicatie verscheept worden naar de eindgebruiker.



### 2.2.2 .NET FRAMEWORK

In deze syllabus zal je voornamelijk applicaties programmeren voor het .NET Framework. Dit Framework is momenteel het meest verregaande van alle drie, maar heeft wel de beperking dat het enkel beschikbaar is voor Windows platformen.

Het .NET Framework ondersteunt ook andere open source programmeren talen zoals F# en Visual Basic (VB).

Wens je later ook applicaties te maken voor Mac, iOS, Linux, Android of zelfs je eigen obscuur Internet-Of-Things apparaat? Je kan dan blijven programmeren in C#, maar je kiest dan een ander Framework dat jouw apparaat ondersteunt, zoals .NET Core of Mono.

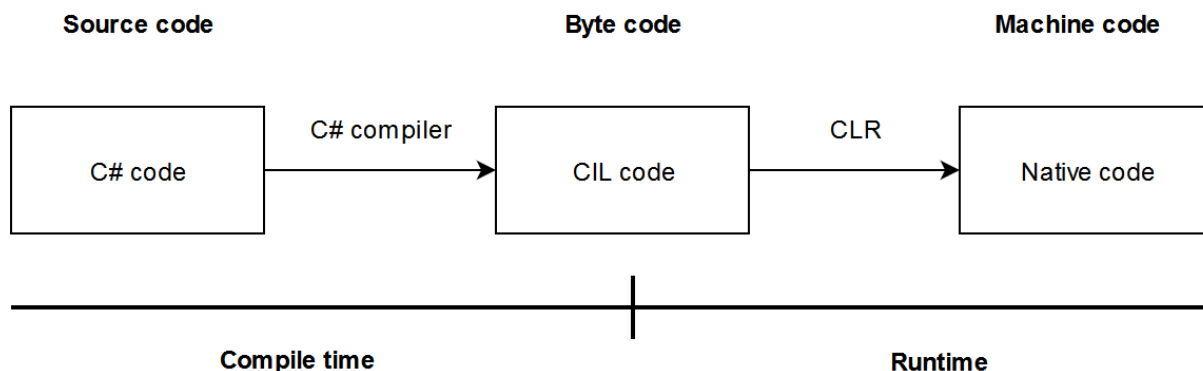
Het .NET Framework wordt automatisch geïnstalleerd wanneer je de programmeeromgeving Visual Studio installeert. Dat betekent dat jouw systeem automatisch de meest recente .NET applicaties kan uitvoeren.

De eindgebruiker van jouw applicatie zal echter de .NET Framework versie waarvoor jouw applicatie geschreven werd geïnstalleerd moeten hebben. Elke nieuwe versie van Windows bevat de meest recente versie van het .NET Framework op dat moment.

### 2.2.3 HOE HET WERKT

Wanneer je een programma schrijft, is het uiteindelijke de bedoeling om dit programma uit te voeren op een machine. In C# wordt de broncode **gecompileerd** naar een tussentaal, de Common Intermediate Language (CIL). Het is deze tussentaal die terug te vinden is in de resulterende **.exe** en **.dll** bestanden. Na het maken van deze **binaries** is de compilatiefase, ook wel de **compile time** genoemd, afgerond.

Elke machine waarop je programma wordt uitgevoerd, krijgt dezelfde assembly (.exe bestand) en dus dezelfde code. Machines en CPU's verschillen echter qua capaciteiten en dus wordt de **Common Language Runtime** (CLR) ingeschakeld. Bij het uitvoeren van het programma zal de CLR de CIL code vertalen naar verstaanbare instructies voor die specifieke machine. De uitvoering van het programma noemen we **run time**.



## 2.3 PROGRAMMEREN IS ANALYSEREN

Wanneer je een applicatie programmeert dan is er een duidelijk doel voor die applicatie. Er zijn een aantal vereisten waaraan de applicatie moet voldoen, bijvoorbeeld: beheren van digitale facturen, afdrukken van facturen.

Voor je kan beginnen met het programmeerwerk moet je deze vereisten goed begrijpen en analyseren. Denk daarbij aan het volgende:

- Vereisten
  - Zijn de vereisten duidelijk?
  - Kunnen de vereisten opgesplitst worden in kleinere deelproblemen?
- Functioneren van het programma:
  - Wat zijn de gegevens waarmee gewerkt moet worden?
  - Wat moet er met die gegevens gebeuren?
  - Wat moet er met het resultaat gebeuren?
  - Welke gebruikersinteractie moet er plaatsvinden?

## 2.4 BELANGRIJKE BEGRIPPEN

---

### 2.4.1 ASSEMBLY

Een binair bestand dat CIL bytecode bevat die door de .NET Framework CLR uitgevoerd wordt. Het bevat de vertaling van jouw broncode. Assemblies komen meestal in twee vormen:

- **Executable:**  
een uitvoerbaar bestand dat eindigt op de .exe bestandsextensie.
- **Library:**  
een bestand dat gemeenschappelijke functionaliteiten bevat die door andere assemblies gebruikt kunnen worden. De bestandsnaamextensie is meestal .dll

### 2.4.2 .NET

.NET is een verzameling assemblies (meestal libraries) die de programmeur toegang geven tot zeer uiteenlopende functionaliteiten. Er zijn drie versies van .NET: het klassieke .NET Framework, .NET Core en Mono.

### 2.4.3 COMPILER

Het programma dat jouw broncode omzet naar bytecode. In C# is dat de C Sharp Compiler, csc.exe dat te vinden is de installatiemap van het .NET Framework.

### 2.4.4 DESIGN-TIME

De ontwikkelingsfase waar de programmeur broncode schrijft en bewerkt.

### 2.4.5 COMPILE-TIME

De ontwikkelingsfase waar de compiler de broncode omzet naar een assembly.

### 2.4.6 RUN-TIME

Het uitvoeren van een programma door een eindgebruiker wordt ook wel run-time genoemd. Voor .NET applicaties wordt deze uitvoering geregeld door de Common Language Runtime, waarvoor een installatie van het .NET Framework nodig is op de machine.



#### Meer informatie

- [MS: Download .NET](#)
- [MS: .NET Programming languages](#)
- [MS: C# Guide](#)

### 3 VISUAL STUDIO

Microsoft Visual Studio is de meest gebruikte, meest complete Integrated Development Environment (IDE) voor het ontwikkelen van applicaties voor het .NET Framework. Het .NET Framework, samen met heel wat tools voor het ontwikkelen, wordt gratis ter beschikking gesteld. Visual Studio komt in verschillende uitvoeringen: Visual Studio Professional, Enterprise en Ultimate zijn commerciële versies waarvoor een licentiekost moet worden betaald.

Microsoft stelt echter een Community-editie van Visual Studio ter beschikking. Deze versie heeft dezelfde mogelijkheden als de Professional Edition en wordt aangeboden voor educatief, persoonlijk en zelfs commercieel gebruik voor bedrijven met max. 5 werknemers.

In deze cursus ga je aan de slag met de **Visual Studio 2019 Community Edition**.

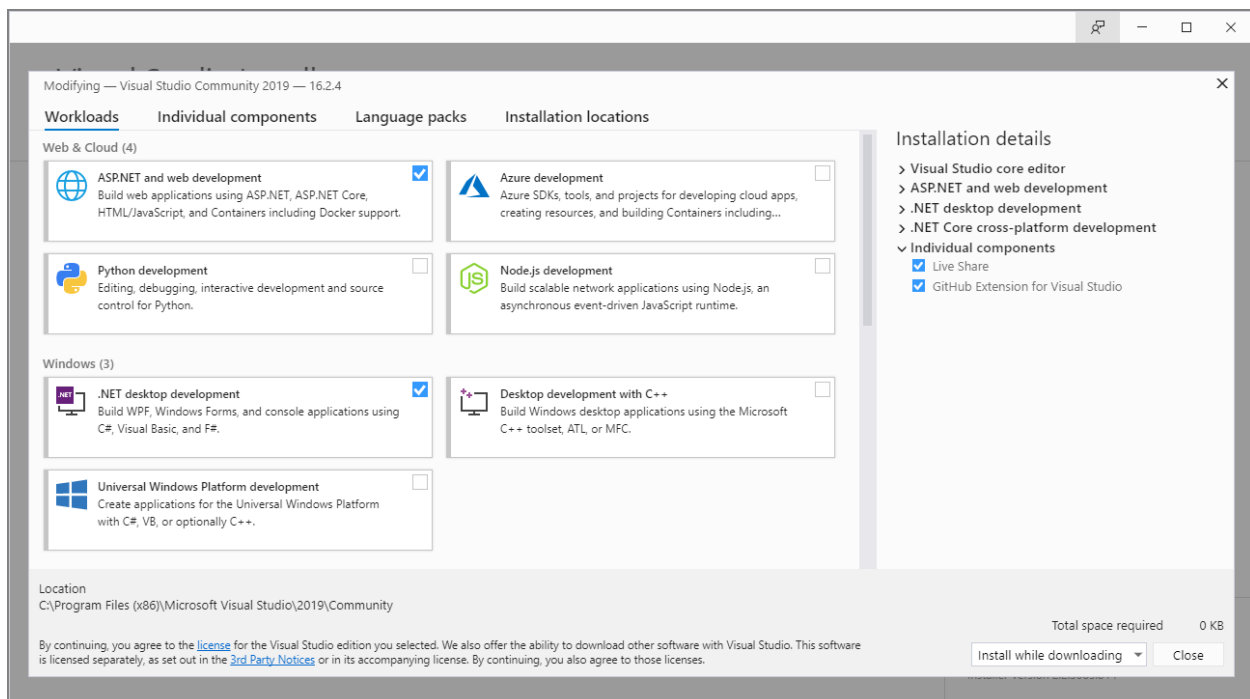


Visual Studio is slechts een **hulpmiddel** om makkelijk een .NET applicatie te programmeren. Je kan ook .NET applicaties ontwikkelen met tal van andere gratis editoren; kladblok, notepad++, MonoDevelop en Visual Studio Code (niet te verwarren met Visual Studio zelf) zijn daar voorbeelden van.

#### 3.1 INSTALLATIE

Je installeert Visual Studio als volgt:

1. Download Visual Studio Community Edition via <https://www.visualstudio.com/vs/community/>
2. Voer het installatiebestand uit.
3. Je krijgt een vraag om akkoord te gaan met de Licentieovereenkomst voor je kan doorgaan.
4. Eens het installatieprogramma is opgestart, krijg je het volgende scherm te zien:



Een ontwikkelaar hoeft zelden over alle ontwikkeltools te beschikken die Visual Studio te bieden heeft. Daarom worden alle beschikbare functionaliteiten van Visual Studio gemakshalve gebundeld in zogenaamde **Workloads**. Hiermee krijg je alle tools en functionaliteiten gekoppeld

aan een bepaalde ontwikkelingstak; bijvoorbeeld webapplicaties, desktopapplicaties, mobile apps, enzovoort.

5. Installeer de volgende Workloads, zodat je alle voorbeelden in deze syllabus kan uittesten:

- **.NET desktop development**
- **ASP.NET and web development**
- **.NET Core cross-platform development**

Ben je nieuwsgierig van aard en wil je op verkenning? Goed zo! Installeer dan ook eens de volgende Workloads:

- Mobile development with .NET
- Game development with Unity

Controleer of je genoeg schijfruimte over hebt voor de gekozen Workloads.

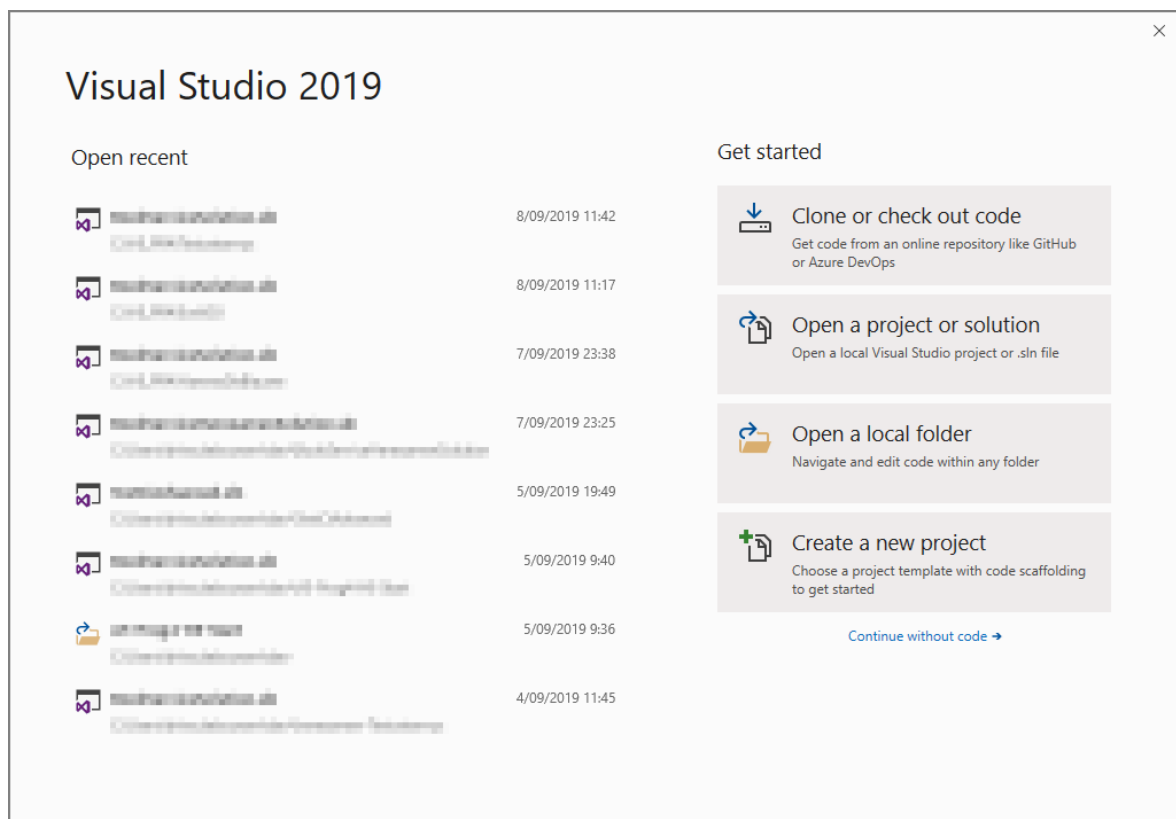
6. Als je klaar bent met kiezen, klik je op **Install**.

7. Wacht tot de installatie voltooid is. Daarna kan je Visual Studio starten via de **Launch** knop, of gewoon vanuit het Startmenu.

### 3.2 RONDLEIDING VISUAL STUDIO

Als je Visual Studio opstart dan komt je in het Start scherm terecht. Dit toont je de meest recente projecten waaraan je hebt gewerkt en enkele andere opties die we later nog zullen bespreken.

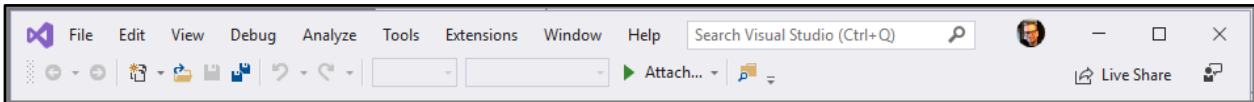
Klik voor nu op **Continue without code** om op verkenning te gaan in Visual Studio.



### 3.2.1 MENU EN TOOLBARS

Bovenaan het hoofdvenster van Visual Studio bevinden zich het menu en de toolbars. Het menu biedt toegang tot alle functionaliteiten die de IDE te bieden heeft en is te veelomvattend om hier te bespreken. In deze syllabus leer je gaandeweg het menu beter kennen.

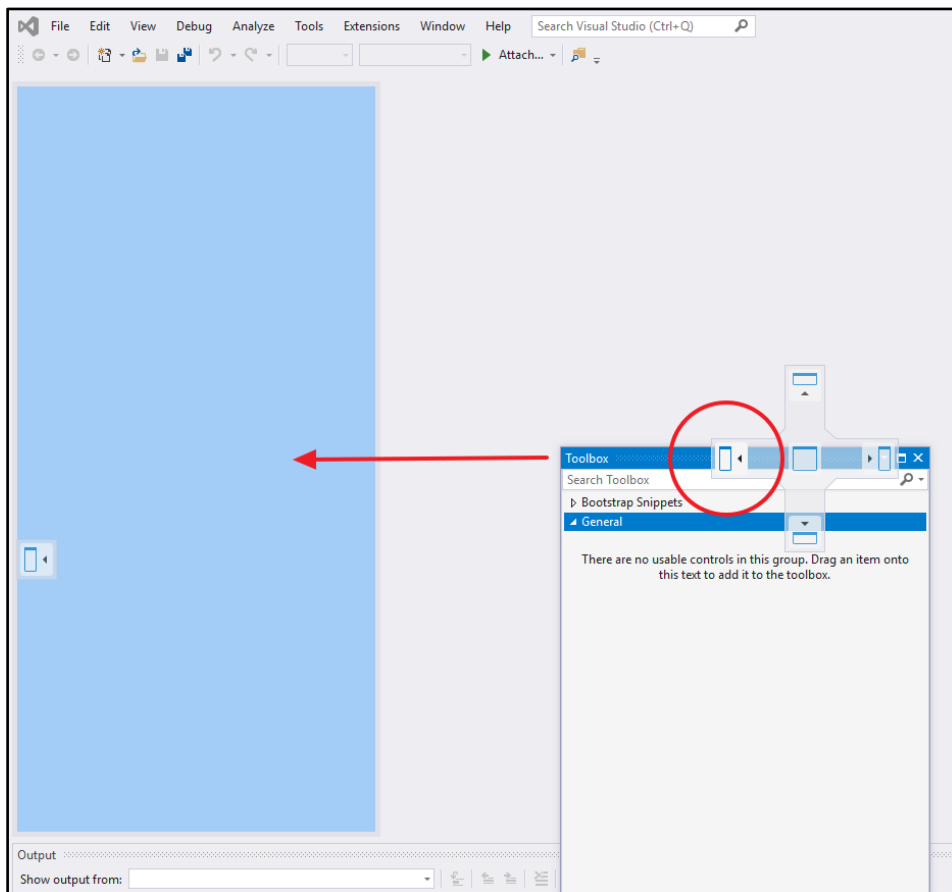
De toolbars bevatten de contextuele taken die beschikbaar zijn voor de huidige situatie. Dat betekent dat de knoppen en toolbars zullen veranderen naargelang het bestand dat je bewerkt. Ook deze zaken leer je stap voor stap kennen.



### 3.2.2 WINDOWS

Visual Studio is een zeer uitgebreide IDE die je volledig zelf kan inrichten. Het bestaat uit heel wat vensters die je kan tonen, verbergen en wijzigen van positie en grootte. Je hoeft ze heus niet allemaal te kennen, zolang je maar weet waar ze te vinden zijn. Maak het alvast een beetje gezellig, want dit wordt je nieuwe programmeeromgeving!

- Het actieve vensters is steeds gekleurd in een accent. In de beginsituatie is dat de Start Page.
- Om een vensters te sluiten, klik op het kruisje.
- Om een vensters te verslepen, versleep je de titelbalk ervan.
- Je kan deze uit het Visual Studio hoofdvenster loskoppelen (handig als je twee monitoren hebt)
- Je kan deze **docken** op een andere locatie in het hoofdvenster. Daartoe sleep je het venster naar het gewenste dock-icoontje. Onderstaande screenshot toont je hoe het toolbox venster aan de linkerkant geplaatst kan worden:



- Vensters die niet zichtbaar zijn kunnen meestal gevonden worden via het menu **View**. Andere, specifiekere Windows zijn te vinden onder het submenu Windows van de volgende menu's:
  - Debug
  - Test
  - Analyse
- Experimenteer eens grondig met deze vensters. Open, sluit en verplaats ze naar hartenlust! Eens je er een grondig zootje van hebt gemaakt, kan je de standaard lay-out terug oproepen via de volgende optie: **Window → Reset Window Layout**.

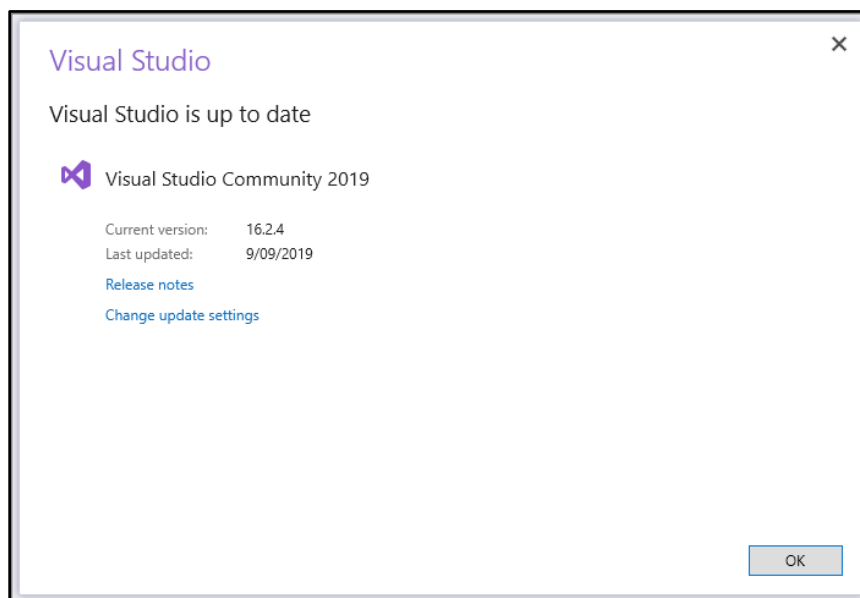
### 3.2.3 EXTENSIONS AND UPDATES

Behalve de standaard ingebouwde functionaliteiten zijn er heel wat handige tools en uitbreidingen beschikbaar voor Visual Studio. Deze kan je vinden via **Extensions → Manage Extensions**

Je ziet standaard welke tools en extensies je hebt geïnstalleerd via **Installed**. Om te zoeken naar nog niet geïnstalleerde uitbreidingen klik je op **Online**.

Tenslotte kan je de extensies up-to-date houden via **Updates**.

Indien je **Visual Studio** al op voorhand had geïnstalleerd, neem je best eens een kijkje of jouw versie de meest recente is. Dit doe je via **Help → Check for Updates**:



### 3.2.4 PROJECTEN EN SOLUTIONS

De manier waarop Visual Studio broncode samenhoudt gebeurt in de vorm van projecten. Een project is een verzameling bestanden die bij elkaar horen en die bij compilatie een assembly opleveren (een .dll of .exe bestand).

Een project heeft meestal de extensie **.csproj** en is een wezen een oplijsting van alle bestanden die tot dat project behoren. De bestanden zelf worden uiteraard op schijf bewaard en zijn al dan niet in (sub)mappen ondergebracht.

Sommige applicaties bestaan uit meerdere assemblies, en dus meerdere projecten. In dat geval kunnen deze projecten gebundeld worden met behulp van een Solution.



Een **project** bevat een verzameling bestanden die onderdeel uitmaken van een programma of library.

Een **solution** bevat een verzameling projecten die met elkaar verwant zijn of elkaar nodig hebben om te functioneren.

## 4 .NET PROJECTEN

De eerste toepassing die je maakt met een nieuwe programmeertaal heeft dikwijls niet veel om het lijf: het is een test of er 'leven in de brouwerij' is. Vaak wordt in het eerste programma enkel een zin op het scherm gezet: "Hello World".

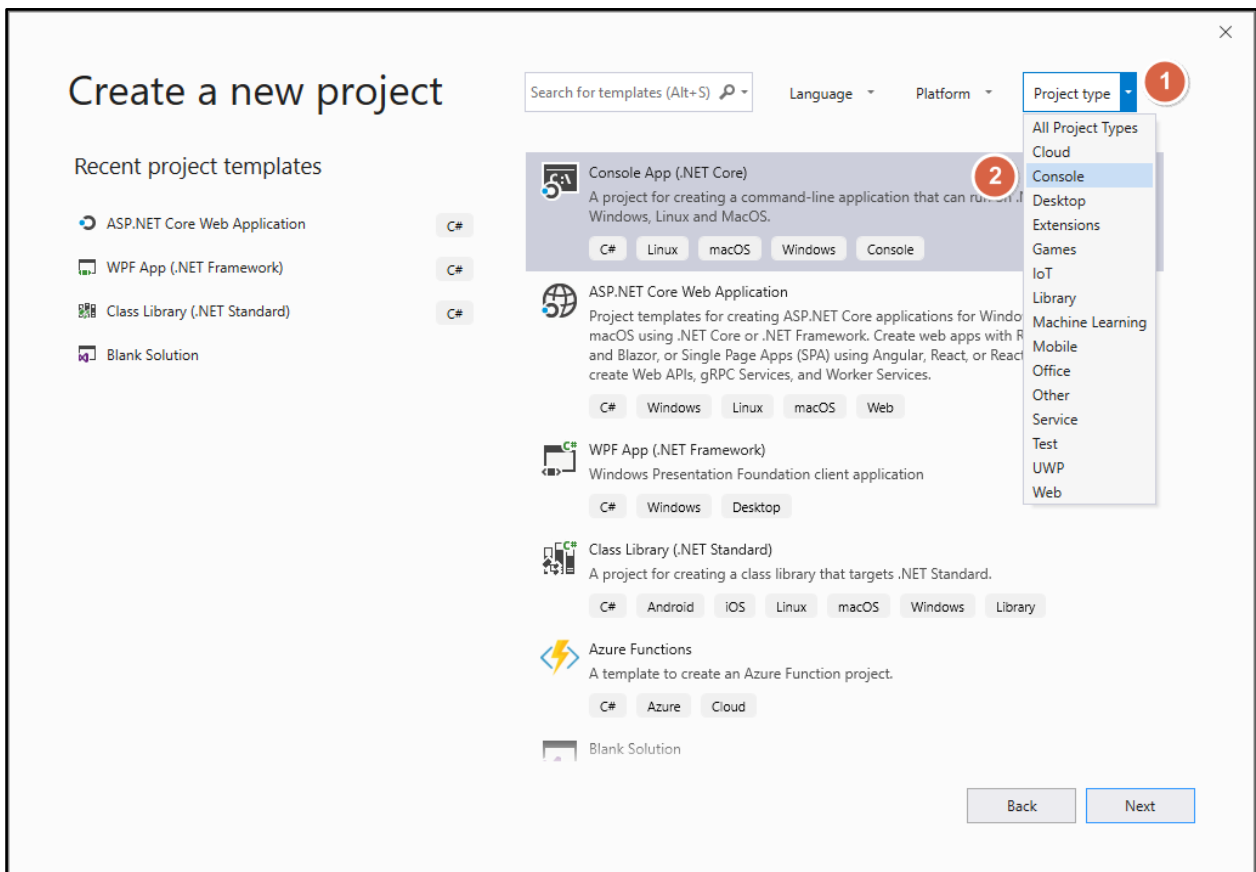
Je gaat nu zo'n Hello World applicatie een aantal keer bouwen in een aantal verschillende applicaties die het .NET Framework ons biedt.

### 4.1 CONSOLE APPLICATIE

Je gaat nu je eerste project maken, een Console applicatie. Dat zijn programma's die uitgevoerd worden in de command line en zijn louter textueel. Ze kunnen tekst inlezen en afbeelden op het scherm. Dat laatste is wat we voorlopig zullen doen.

#### 4.1.1 EEN NIEUW PROJECT MAKEN

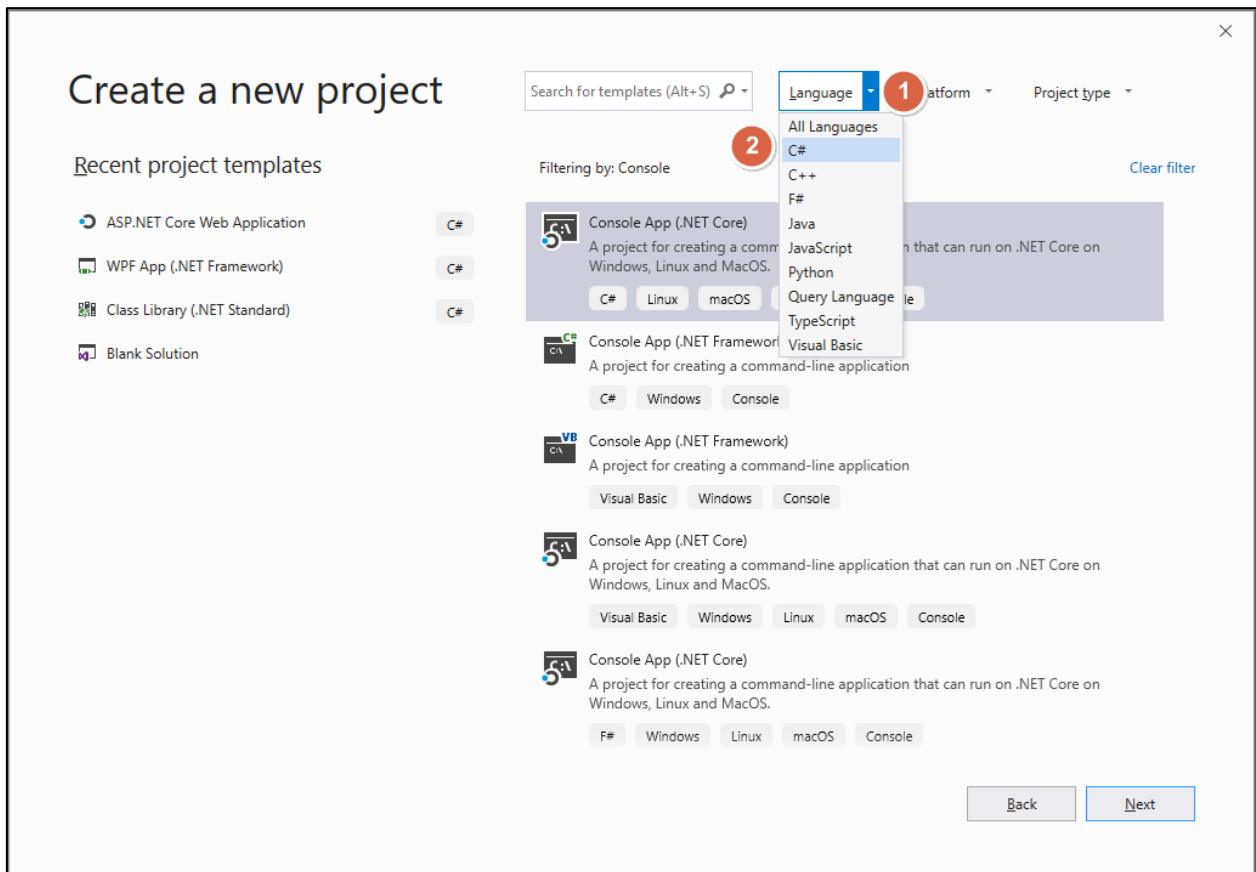
1. Start Visual Studio en kies **Create a new project** (Indien je Visual Studio al opgestart was kan je ook kiezen voor **File → New Project**)
2. In het nieuw dialoogvenster kun je filteren op bepaalde projecten. Klik op **Project Type** en kies voor **Console**:



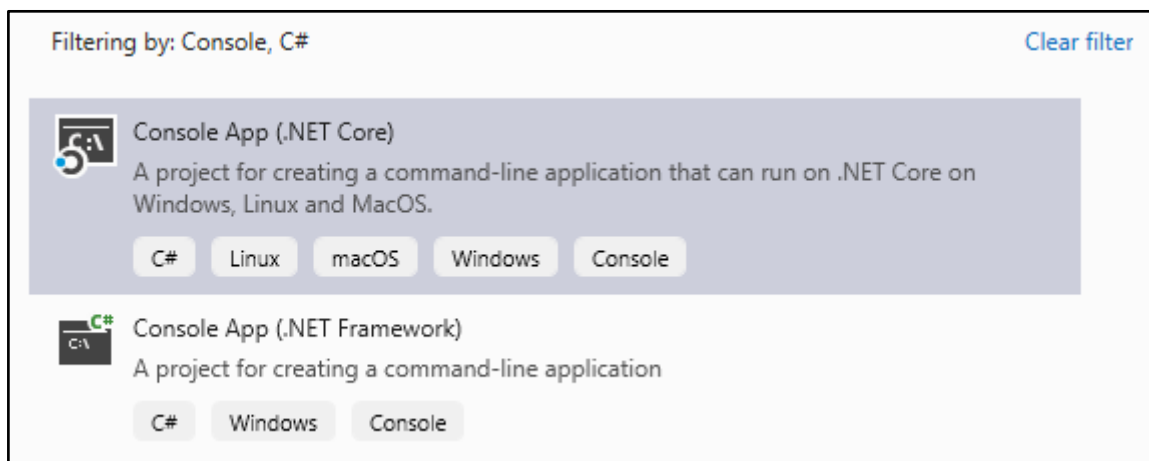
We zien nu enkel en alleen de Console template projecten. Aangezien we Visual Studio ook kunnen gebruiken om een console applicatie in Visual Basic te programmeren in plaats van C#, zien we ook deze opties nog staan in de lijst.



We gaan onze lijst met templates nog wat meer filteren. Klik op **Language** en kies voor **C#**:



We houden nu nog twee opties over: Console App (.NET Core) en Console App (.NET Framework):



Zoals reeds eerder vermeld hebben we meerdere .NET Frameworks. Bij onze opties is al goed te zien dat we onze .NET Framework Console App enkel kunnen gebruiken op Windows. Waar we bij de .NET Core Console App zien dat we deze ook kunnen gebruiken voor Linux, macOS en Windows. Wij zullen in de module Programming Basics hoofdzakelijk gebruik maken van het .NET Framework en **niet** van .NET Core. Het .NET Core framework wordt pas later gebruikt in onze opleiding.

3. Selecteer **Console App (.NET Framework)** en klik op **Next**.
4. Op het volgende scherm geef je volgende zaken in:
  - Je Project name (HelloWorlds.Cons)

- Je Location (de locatie van je project op je computer)
- Je Solution name (HelloWorlds)
- Framework (kies .NET Framework 4.7.2 of lager indien deze niet aanwezig is op je computer)
- Klik daarna op de knop Create.

Configure your new project

Console App (.NET Framework) C# Windows Console

Project name  
HelloWorlds.Cons 1

Location  
C:\Users\dries.deboosere\dev\programming-basics\ 2

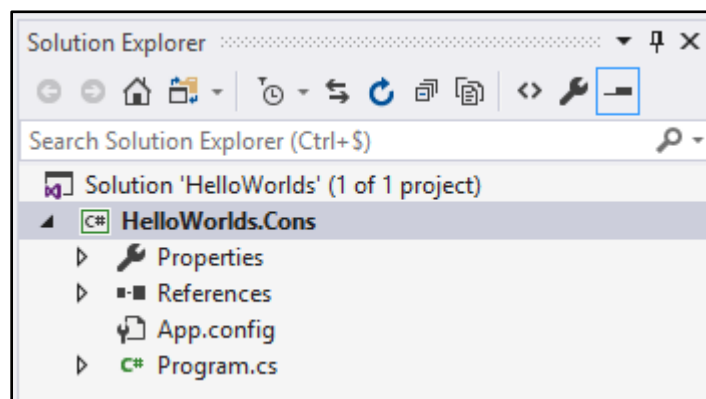
Solution name 1  
HelloWorlds 3

☐ Place solution and project in the same directory

Framework  
.NET Framework 4.7.2 4

Back Create 5

- Visual Studio genereert nu een basisproject aan de hand van het gekozen sjabloon.
- Je kan de inhoud van je project zien in het **Solution Explorer** venster van Visual Studio. Je vindt dit meestal aan de rechterkant van je scherm, of via **View** → **Solution Explorer**.



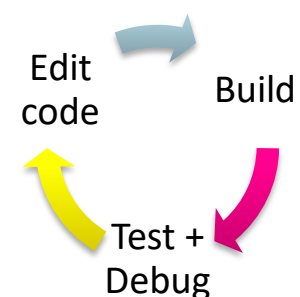
- Je merkt dat er al een aantal bestanden zijn aangemaakt, onder meer een C# bestand genaamd **Program.cs**. Deze bestanden zijn uiteraard ook op je bestandssysteem terug te vinden, met name in de map die je hebt gekozen toen je het project genereerde.
- Een overzicht van wat er te zien valt in de Solution Explorer:

<b>Solution</b> <b>"CsCourseHelloWorlds"</b>	Het solutionbestand. Een solution bundelt projecten zodat je ze in hetzelfde Visual Studio hoofdvenster instantie kan bewerken.
<b>CsCourseHelloWorlds.Cons</b>	Het C# projectbestand. Een project verzamelt files met broncode, figuren, compilatie instellingen, ... voor een c-sharp project.
<b>Properties</b>	Maakt onderdeel uit van het project en bevat een bestand AssemblyInfo.cs. Hier kan je extra info opgeven over je code zoals de naam van de auteur, versienummer, ...
<b>References</b>	Een map die de verwijzingen bijhoudt die vanuit je project legt naar andere assemblies die ofwel deel uitmaken van het .NET Framework, aangeleverd werden door een andere ontwikkelaar of gewoon door jezelf werden geprogrammeerd.
<b>App.config</b>	Een XML bestand dat eventuele instellingen voor jouw applicatie kan bevatten. Welke instellingen dat juist zijn kan je zelf volledig kiezen. Dit bestand is optioneel.
<b>Program.cs</b>	Een eerste C# codefile dat reeds wat code bevat. Dit bestand werd automatisch aangemaakt door Visual Studio en werd alvast geopend in een venster.

#### 4.1.2 CODE, BUILD, TEST, REPEAT

De programmeercyclus bestaat uit het wijzigen of toevoegen van codebestanden, waarop de broncode gecompileerd wordt (Build). De programma uitvoering wordt vervolgens getest (Test) op het gewenste resultaat, en fouten worden opgespoord (Debug). Hierna worden er opnieuw wijzigen aangebracht en herhaalt deze cyclus zich tot het eindresultaat bereikt wordt.

Het is een goede gewoonte om regelmatig de effecten van je code te bekijken volgens dit vereenvoudigd patroon. Zo niet kunnen de fouten zich opstapelen tot een onoverzichtelijk kluwen die voor een beginnende programmeur vele uren kan kosten.



## CODE

Open het bestand **Program.cs** door erop te dubbelklikken in de Solution Explorer. De inhoud van dit bestand ziet er zo uit:

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

namespace HelloWorlds.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

In het codebestand zie je reeds een aantal belangrijke keywords uit de C# syntax:

- methode** Het stukje code dat begint met `static void Main` is een **methode**. Methodes bevatten instructies tussen de accoladen `{` en `}` die één voor één worden uitgevoerd. De `Main` methode is hier een zeer belangrijke methode; het vormt het startpunt van de applicatie.
- class** Een basiscomponent voor object-georiënteerd programmeren. In een klasse worden eigenschappen, methoden en andere zaken gedefinieerd die structureel en/of logisch gezien bij elkaar horen. De naam van een klasse moet steeds uniek zijn. Alles wat tot de klasse `Program` behoort staat tussen de bijbehorende accolades `class Program { ... }`.
- namespace** Verzamelnaam voor een groep gerelateerde klassen. De volledige naam van een klasse wordt steeds voorafgegaan door zijn namespace. De klasse `Program` heet dus eigenlijk `CsCourse.HelloWorlds.Cons.Program`.  
  
Door het gebruik van namespaces blijft het mogelijk om een tweede klasse genaamd `Program` te maken, het zij dan wel in een andere namespace zodat de uniciteit gerespecteerd blijft.

Maak nu volgende wijzigingen in dit bestand:

```
using System; ④

namespace HelloWorlds.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            /* let's output some text ③
             * in the console window */
            Console.WriteLine("Hello World"); ③
            ② string input = Console.ReadLine(); //wait for user to hit enter and read
all text input
        }
    }
}
```

Welke wijzigingen hebben we nu precies gedaan?

1. De eerste instructie in de `Main()` methode is het afbeelden van een stukje tekst. Dit doe je met de methode `WriteLine()` die te vinden is in de klasse genaamd `Console`.
2. De tweede en laatste instructie is het inlezen van tekst tot de gebruiker op de enter-toets drukt (en dus een nieuwe lijn ingeeft). Deze tekst wordt bewaard in een stukje geheugen dat je `input` hebt genoemd. Dit kan met behulp van de `Readline()` methode, die eveneens behoort tot de `Console` klasse.
3. De groene tekst is commentaar. Deze wordt genegeerd door de compiler en wordt dus niet uitgevoerd.

Gebruik het om jezelf en andere programmeurs te verklaren wat bepaalde instructies precies doen.

`/* */`      Plaatst commentaar over meerdere lijnen heen. Hiermee kan je uitgebreide uitleg verschaffen over bepaalde stukken code.

`//`      Commentaar voor de rest van de lijn. Hierna kunnen er geen instructies meer volgen. De rest van de lijn wordt genegeerd. Handig voor korte uitleg.

4. Een aantal `using` instructies bovenaan het codebestand werden verwijderd omdat ze toch niet gebruikt worden. De enige `using` statement die benut wordt is `using System;`

Dankzij het keyword `using` gevolgd door de naam van een namespace hoef je niet de volledige naam van een klasse te gebruiken. Alle klassen die bevat zijn in die namespace kan je gewoon in hun korte notatie schrijven.

In dit geval importeren we de .NET Framework namespace genaamd `System`. In deze namespace zit onder meer de klasse genaamd `Console`. Had je deze namespace niet geïmporteerd, dan moest je de volledige naam van die klasse gebruiken: `System.Console`.

Dat zou dus langere, onleesbaardere code opleveren. Het gebruik van `using` is overigens volledig vrijblijvend, maar bijzonder nuttig. Het verwijderen van ongebruikte `using` statements heeft geen enkel effect op de uitvoering van het programma, maar levert ook weer nettere code.



### Opgelet

- Elke statement in C# wordt afgesloten met een **puntkomma** ;
- C# syntax is **hoofdlettergevoelig**: een h is dus niet gelijk aan een H.

## BUILD

Je bent nu klaar voor de compilatie van de kersverse wijzigingen. De compilatie wordt gedaan met het programma `csc.exe` dat op de achtergrond zal worden uitgevoerd. Je doet dat via het menu **Build** → **Build Solution** of de sneltoets **CTRL + SHIFT + B**.

Onderaan in de Output venster zie je de voortgang van de compilatie. Indien er fouten optreden dan wordt je daarvan op de hoogte gesteld.

```

Output
Show output from: Build
1>----- Build started: Project: HelloWorlds.Cons, Configuration: Debug Any CPU -----
1> HelloWorlds.Cons -> C:\Users\dries.deboosere\dev\programming-basics\HelloWorlds\HelloWorlds.Cons\bin\Debug\HelloWorlds.Cons.exe
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
  
```

Na een succesvolle compilatie wordt het executable bestand gemaakt. In dit geval is dat **HelloWorlds.Cons.exe** dat zich bevindt in de **/bin/debug** map van ons project.

Mocht je foutieve code hebben ingevoerd, dan zal de compiler stoppen. Er wordt geen exe bestand aangemaakt. In plaats daarvan verschijnt het venster genaamd Error List, dat je vertelt waar de fout zich bevindt. In het onderstaande voorbeeld heb ik twee fouten gemaakt. Kan je ontcijferen welke?

Error List						
Entire Solution		2 Errors	0 Warnings	0 Messages	Build + IntelliSense	
	Code	Description	Project	File	Line	
✖	CS1002	; expected	HelloWorlds.Cons	Program.cs	12	
✖	CS0103	The name 'console' does not exist in the current context	HelloWorlds.Cons	Program.cs	11	

Dit zijn de fouten:

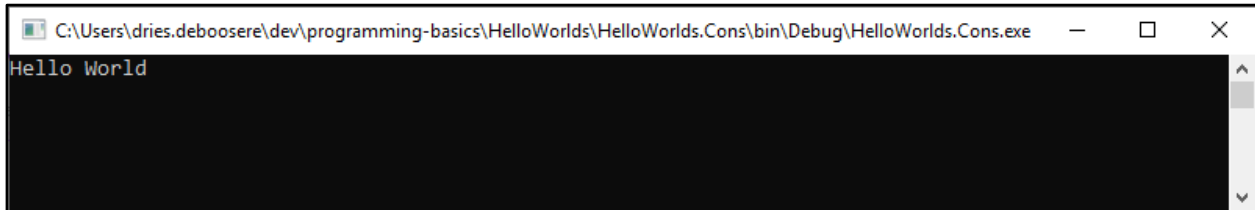
- Op lijn 11 staat “console” met een kleine “c” geschreven. De compiler begrijpt dit niet. Ik bedoelde wellicht de klasse “Console” met een hoofdletter. Onthoud dat C# hoofdlettergevoelig is.
- Op lijn 12 wordt er een puntkomma verwacht. Eigenlijk betekent dit dat ik op lijn 12 een puntkomma vergeten te plaatsen ben om het statement te beëindigen.

Probeer gerust zelf eens een fout te introduceren in de code zodat het compileren mislukt. Bestudeer het Error List venster en corrigeer de fouten alvorens je verder gaat.

## TEST + DEBUG

Om een project uit te voeren ga je naar **Debug → Start Debugging**. Je kan ook op de **F5** toets drukken of de  **Start** knop in de debug toolbar gebruiken.

Voer de applicatie uit via **Debug → Start Debugging**. Het consolescherm verschijnt met het verwachte resultaat.



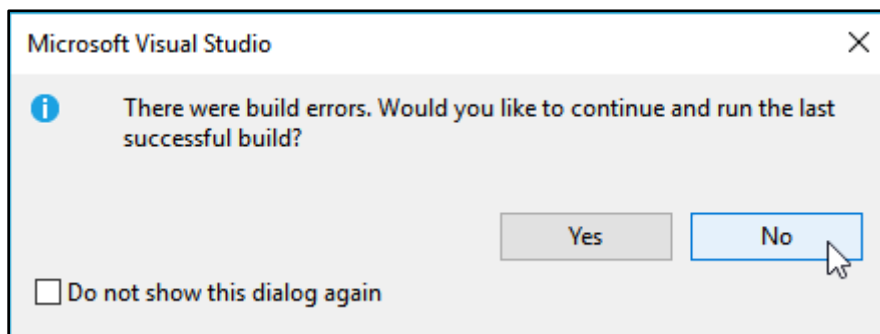
Tijdens de uitvoering met de Debugger kan je de broncode niet meer wijzigen. Let ook op de gewijzigde toolbar in Visual Studio:



Omdat de laatste instructie van het programma een `Console.ReadLine()` betreft moet je op de enter-toets drukken om verder te gaan. De applicatie wordt automatisch afgesloten na uitvoering van de laatste instructie.

Wanneer je de broncode wijzigt dan zal het starten van de applicatie automatisch een hercompilatie tot gevolg hebben. Je kan dus bij elke wijziging direct op **F5** drukken om het resultaat van je wijzigingen te zien.

Indien je code compilatiefouten bevat en je drukt op **F5**, dan krijg je het volgende dialoogvenster:

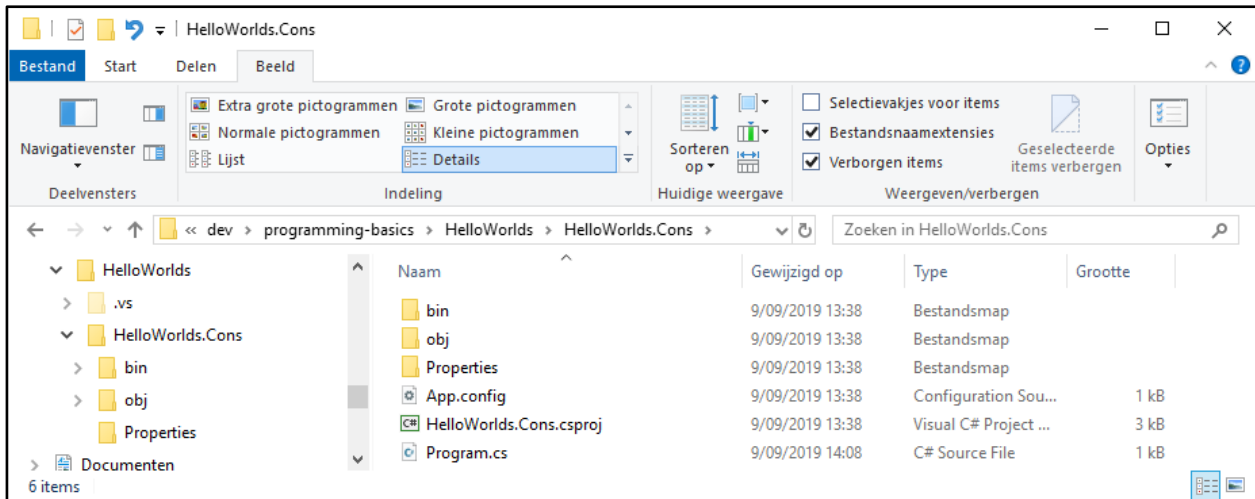


Kies bij dit scherm steeds voor **No**, want je wil immers de meest recente versie starten. Je moet dan eerst de fouten corrigeren zodat je project gecompileerd kan worden.

### 4.1.3 BUILD OUTPUT

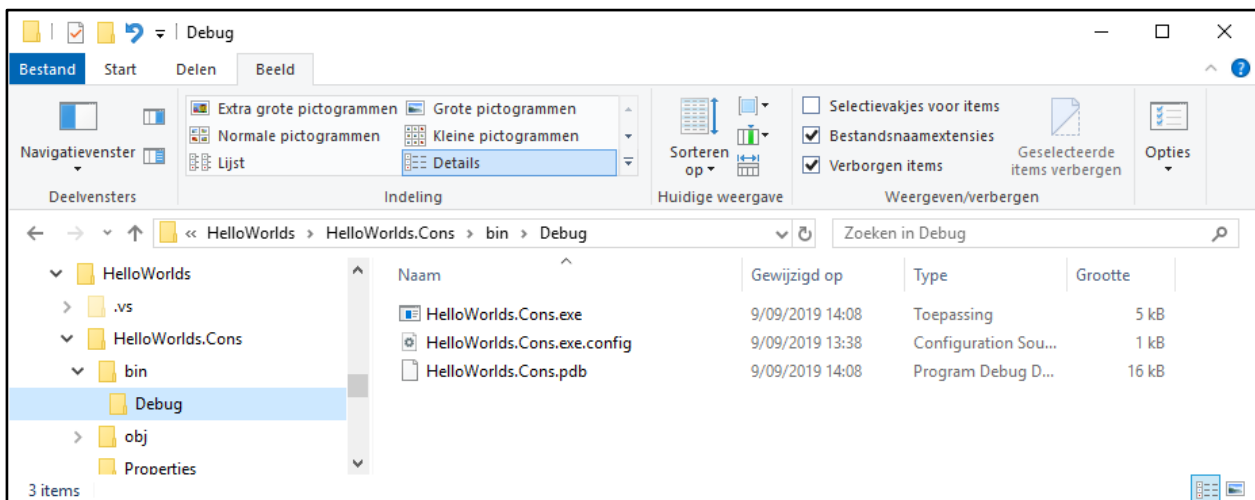
Wanneer je het project compileert, manueel met behulp van **CTRL+SHIFT+B** of automatisch bij uitvoering ervan, dan wordt het resultaat daarvan in de **/bin** map geplaatst. De bin map (staat voor "binaries") bevindt zich in de projectmap.

Om snel naar de projectmap in je Windows Verkenner zichtbaar te maken kan je rechtsklikken op de projectnaam in de Solution Explorer en kies je voor **Show Folders in File Explorer**.



Je bemerkt de **bin** en **obj** mappen. Deze bevatten elk een submap genaamd **Debug**. We hebben het project immers gecompileerd als ontwikkelingsversie. Een productieversie resulteert in een gelijkaardige map genaamd Release.

Open de **/bin/Debug** map en bekijk de inhoud. Je vindt er drie bestanden die dezelfde naam hebben als je project. Zet de weergave van bestandsnaamextensies in Windows Verkenner gemakshalve aan.



**.exe** Dit is de resulterende **assembly** van het project. Het bevat de IL bytecode die door de .NET Framework CLR kan worden uitgevoerd. Je kan jouw applicatie dus ook van hieruit starten. Wanneer je op **F5** drukt is dit het bestand dat wordt uitgevoerd.

**.config** Dit is het oorspronkelijke App.config bestand, dat meereist met de assembly.

**.pdb** Dit bestand wordt gebruikt om fouten op te kunnen sporen in Debug mode. We komen daar later op terug.

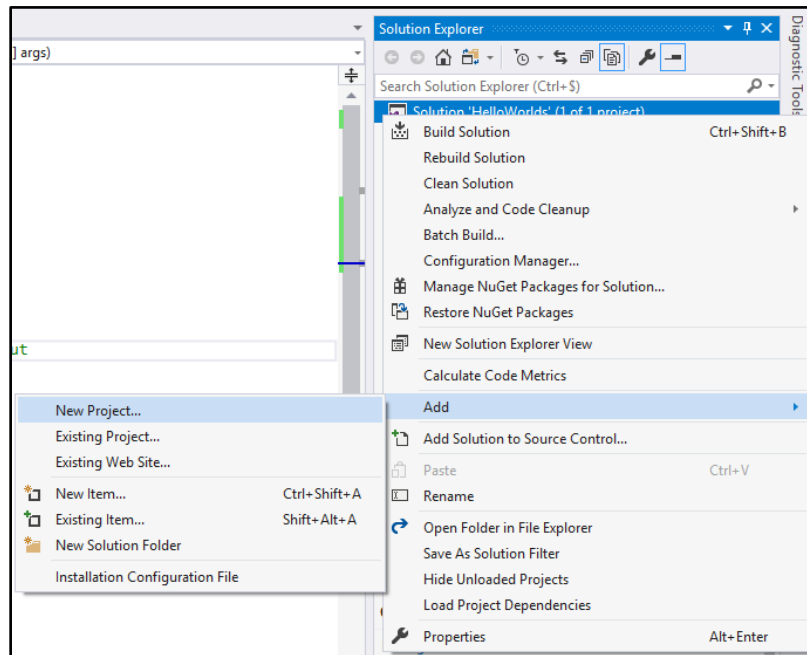


## 4.2 WINDOWS PRESENTATIONS FORMS (WPF)

Je maak nu een tweede applicatie met een Graphical User Interface (GUI) in de vorm van een venster. Het project dat deze applicatie bevat mag deel uitmaken van dezelfde solution.

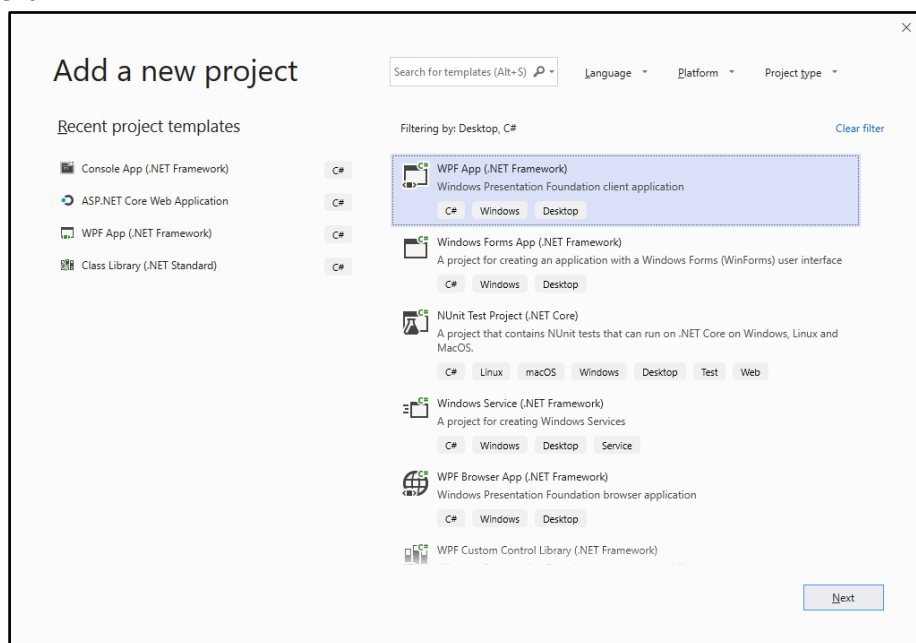
### 4.2.1 EEN PROJECT TOEVOEGEN AAN EEN SOLUTION

1. Zorg dat je solution genaamd **HelloWorlds** geopend is in Visual Studio.
2. Rechtsklik op de solution in de Solution Explorer en kies **Add → New Project**



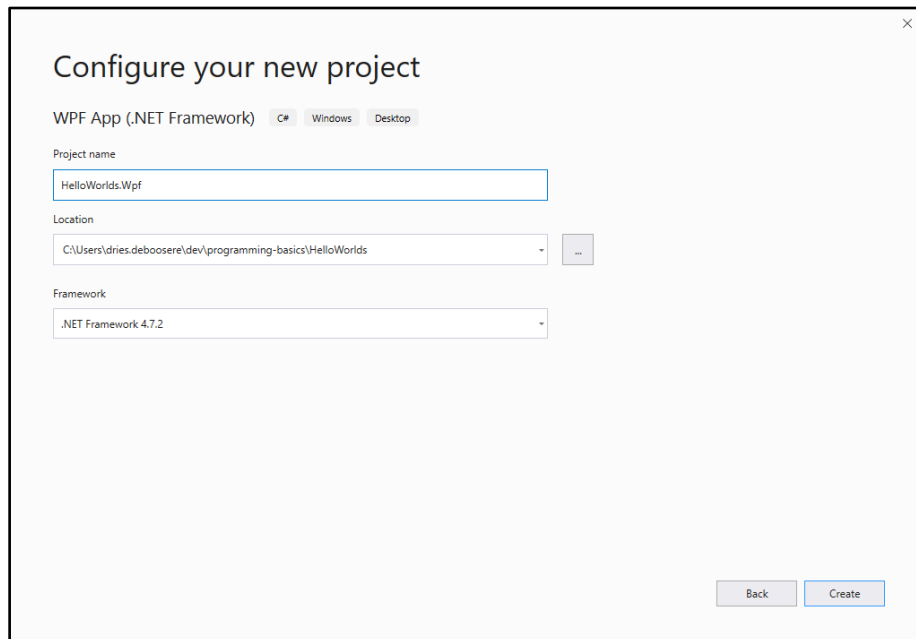
3. In het **New Project** dialogvenster kies je de volgende opties:

- Project type: Desktop
- Project template: WPF App (.NET Framework)
- Klik op Next



Kies een naam voor dit nieuw project:

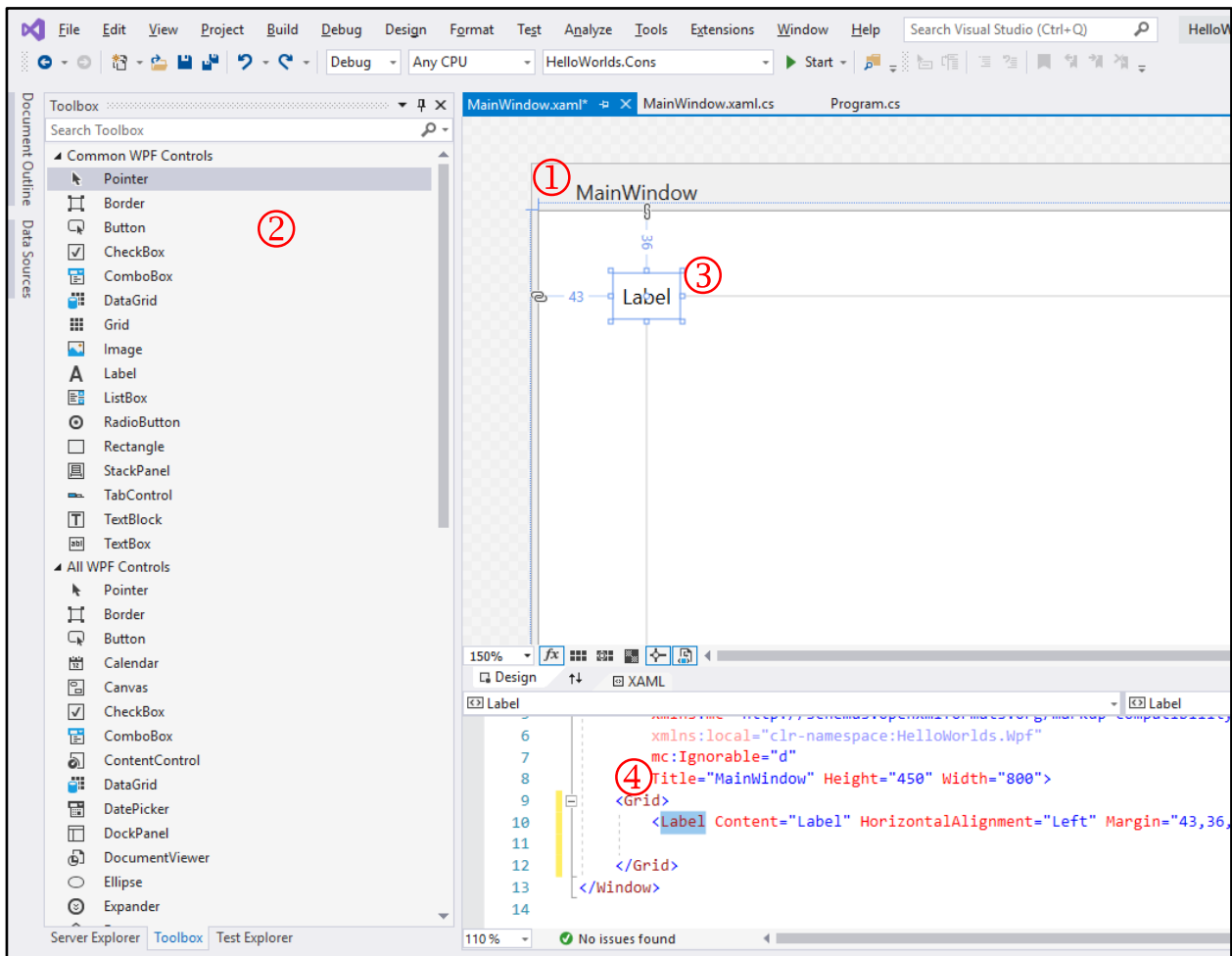
- Projectnaam: **HelloWorlds.Wpf**



Nadat Visual Studio het projectsjabloon genereert zie je een grafische omgeving van een typische Windows desktopapplicatie. Het hoofdscherm, MainWindow genaamd, wordt geopend in een nieuw Visual Studio venster. Je kan dit venster inrichten zoals je zelf wil: door middel van tekstvakken, knoppen, lijsten en vele andere **controls** stel je de gebruiker in staat om de applicatie te besturen. Je vindt deze controls in de **Toolbox**, wanneer het MainWindow venster zichtbaar hebt gemaakt in Visual Studio.

4. Open de **Toolbox** via het menu **View → Toolbox**.
5. Sleep een **Label** control vanuit de Toolbox naar de layout van het MainWindow.

Je zou nu de volgende situatie moeten hebben:



- ① Het MainWindow venster kan je openen via de Solution Explorer en stelt je in staat om jouw GUI te ontwerpen met behulp van **Controls**.
- ② De **Toolbox** bevat alle WPF Controls waarmee je jouw ontwerp kan inrichten.
- ③ Wanneer je een **Label** control op het MainWindow sleept, dan kan je allerlei eigenschappen veranderen, zoals de plaats, de grootte en de inhoud ervan. Probeer het Label gerust eens te verplaatsen en te wijzigen in grootte.
- ④ Het onderste scherm toont de XAML-code die de GUI definieert. Ervaren WPF-programmeurs kunnen het scherm opbouwen door middel van deze code in plaats van het klik-en-sleep systeem erboven.

### 4.2.2 CONTROL EIGENSCHAPPEN WIJZIGEN

Je wenst nu de inhoud van het Label te wijzigen.

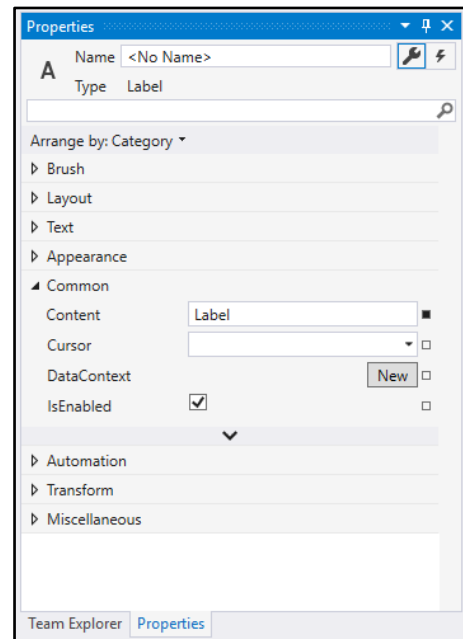
6. Selecteer het Label.
7. Open het **Properties** (eigenschappen) venster via **View → Properties Window** of via **F4**.  
Dit venster staat standaard rechts onder in je Visual Studio omgeving.

Via het **Properties Window** kan je talloze eigenschappen van de geselecteerde Control (of Window) instellen.

Het is niet de bedoeling om deze allemaal van buiten te kennen, maar het is belangrijk om er je weg in te vinden zodat je een idee hebt wat er allemaal kan worden ingesteld.

8. Zoek de eigenschap genaamd **Content** en wijzig het naar "Hello World!".

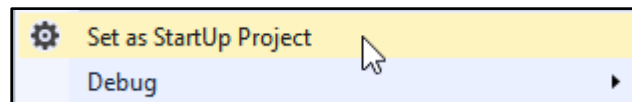
Merk op dat ook de tekst automatisch gewijzigd wordt in het MainWindow venster, en dat eigenschap eveneens in de XAML code wordt aangepast.



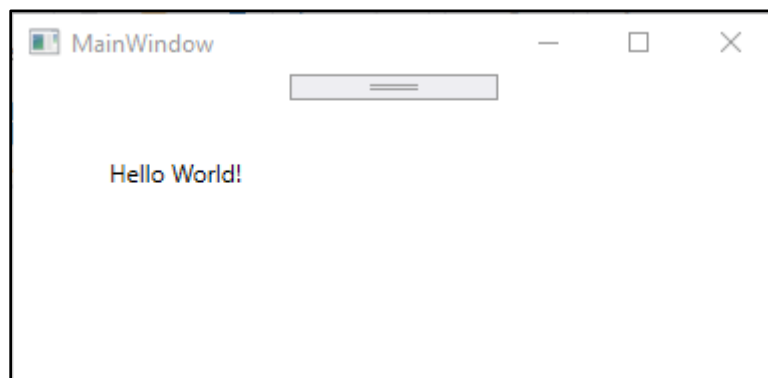
Je bent nu klaar om je GUI applicatie te testen.

Omdat de solution meerdere projecten bevat, moet je nog kiezen welke applicatie er moet worden gestart bij een druk op de **F5** knop.

9. Stel HelloWorlds.Wpf in als **start up project** door een rechtsklik op het project in de Solution Explorer. Kies in het contextmenu voor "**Set as Startup Project**"



10. Druk op **F5** om de applicatie uit te voeren.



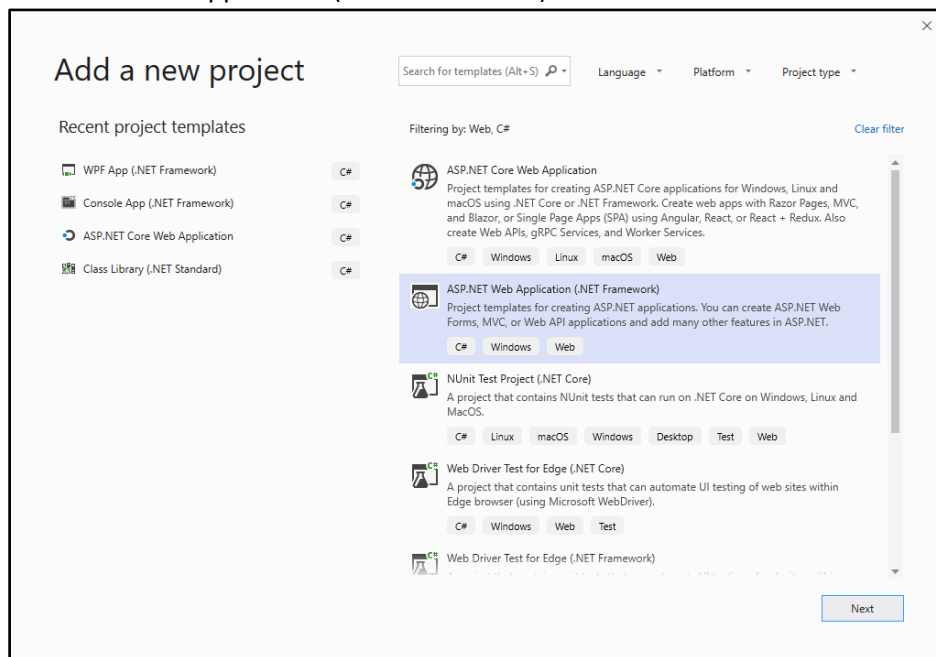
11. **Sluit** de applicatie zodat je **verder kan werken** in Visual Studio.

## 4.3 WEBAPPLICATIE

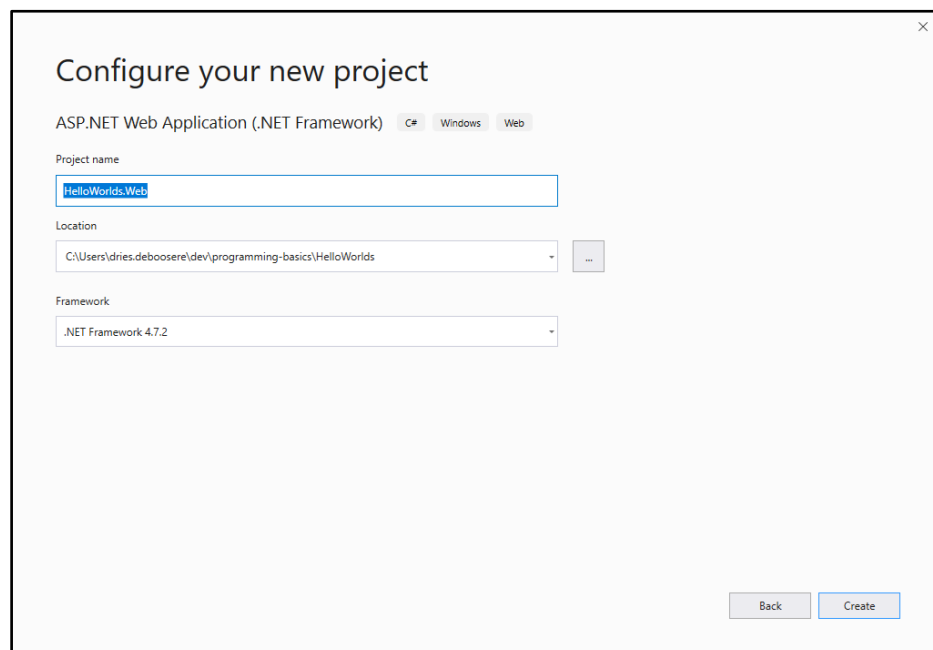
Als laatste voorbeeldapplicatie maak je een webapplicatie aan. Dit zijn applicaties die uitgevoerd worden door een Webserver. Wanneer een bezoeker een correcte URL intikt in zijn browser zal de webapplicatie de inhoud van de gevraagde webpagina terugsturen naar de browser.

### 4.3.1 EEN WEBAPPLICATIE PROJECT MAKEN

1. Zorg dat je solution genaamd **HelloWorlds** geopend is in Visual Studio.
2. Rechtsklik op de solution in de Solution Explorer en kies **Add → New Project**
3. In het **New Project dialoogvenster** kies de volgende opties:
  - Sjabloon: ASP.NET Web Application (.NET Framework)

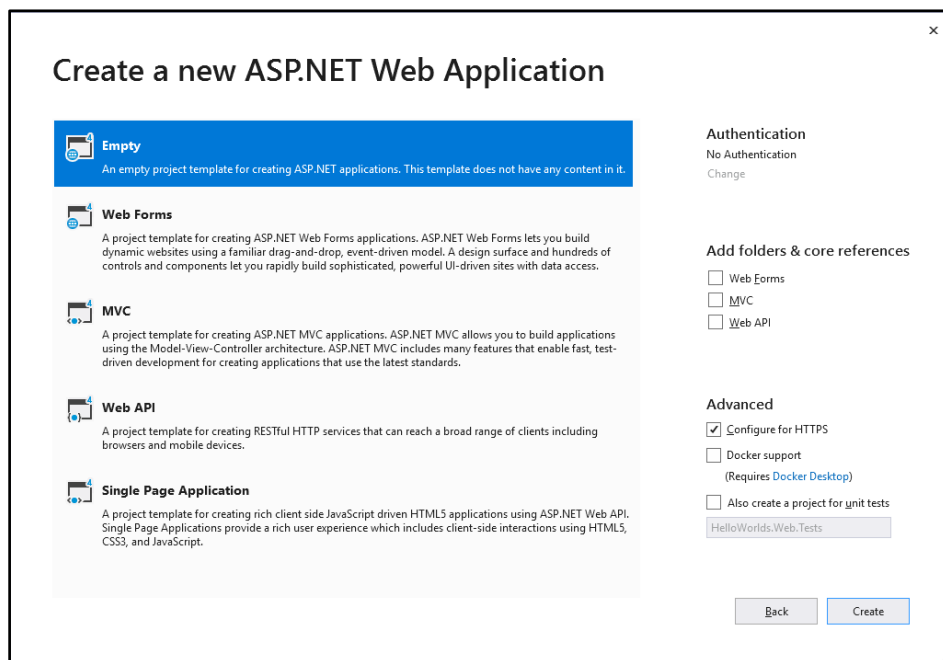


- Projectnaam: **HelloWorlds.Web**



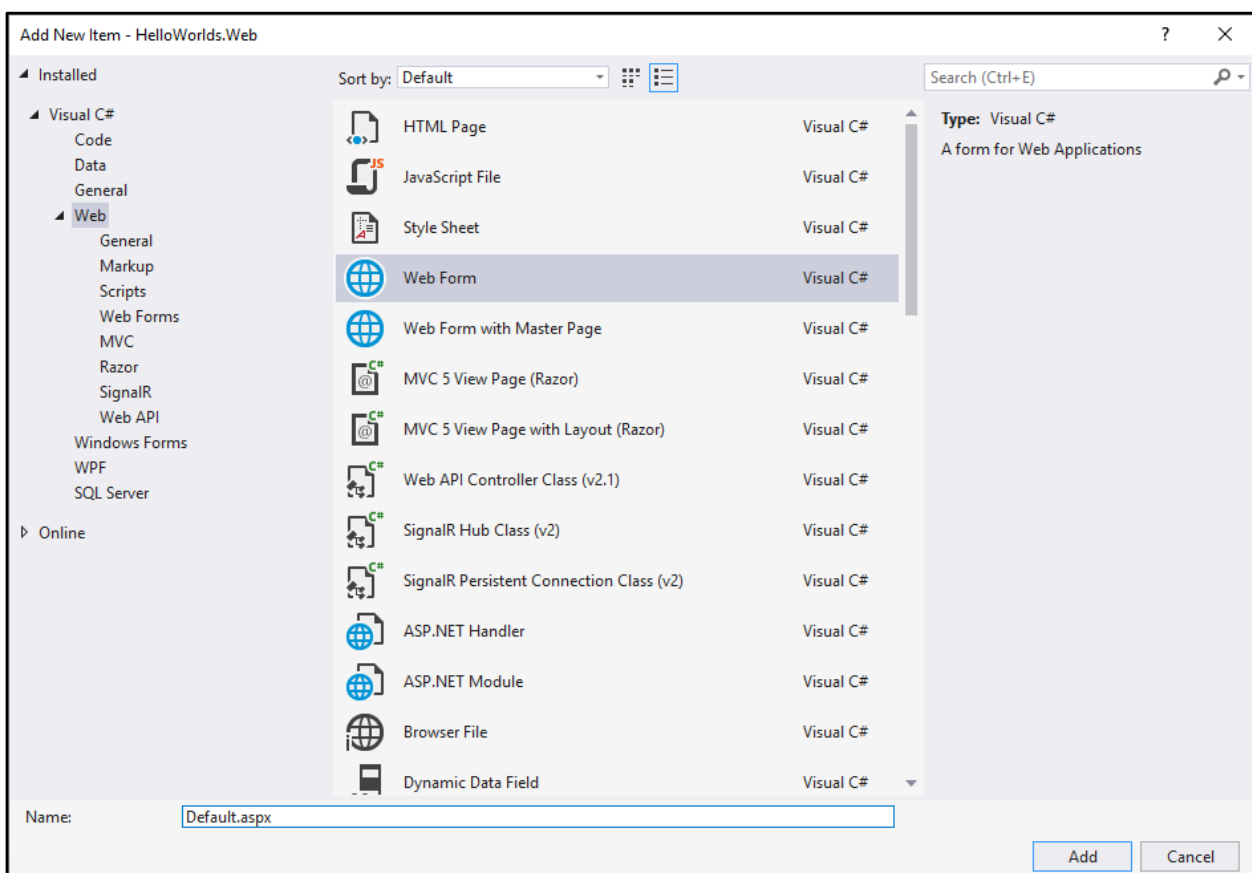
4. In het scherm dat hierop volgt kies je voor een **“Empty”** sjabloon.

## 5. Vink Configure for HTTPS uit



## 6. Rechtsklik op het project in de Solution Explorer en kies **Add → New Item**

## 7. Kies voor een **Web Form** en geef het de naam **Default.aspx**.

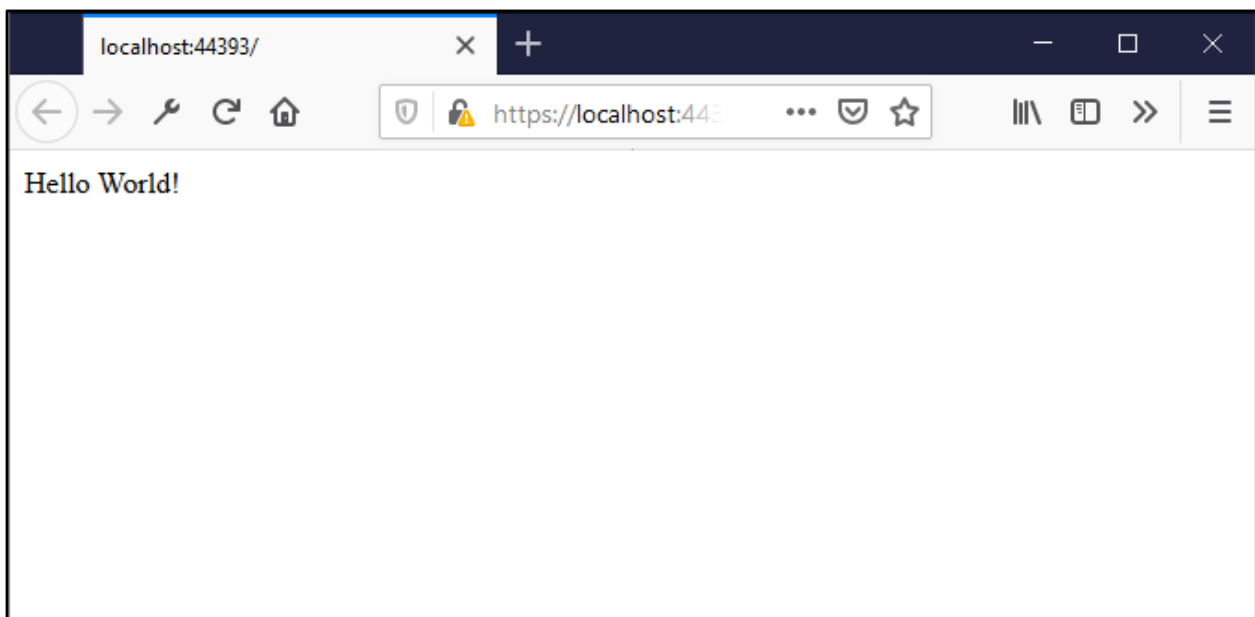


8. Met Default.aspx geopend in Visual Studio sleep je een Label control vanuit de toolbox naar de pagina. Plaats de control tussen de twee `<div></div>` elementen en geeft het als inhoud "Hello World!".

De html code ziet er nu als volgt uit:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label ID="Label1" runat="server" Text="Hello World!"></asp:Label>
    </div>
  </form>
</body>
</html>
```

9. Stel HelloWorlds.Web in als startup project via een rechtsklik in de Solution Explorer en de optie **Set as Startup Project**.
10. Voer de applicatie uit met **F5** of de startknop bovenaan.  
De inhoud van je webpagina wordt nu getoond in je standaardbrowser.

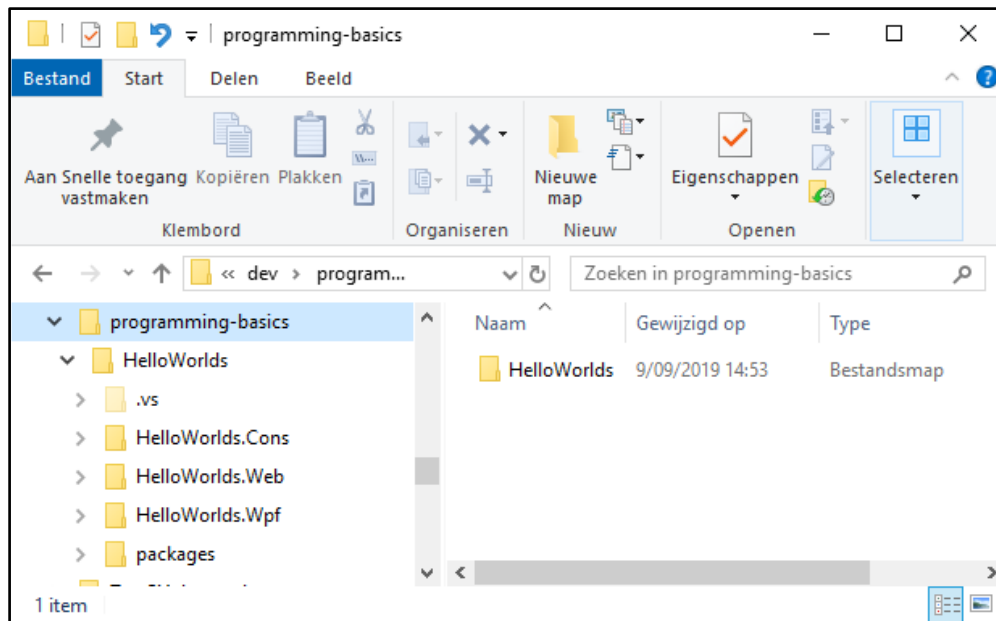


11. Stop de applicatie of sluit de Browser om verder te kunnen werken in Visual Studio.

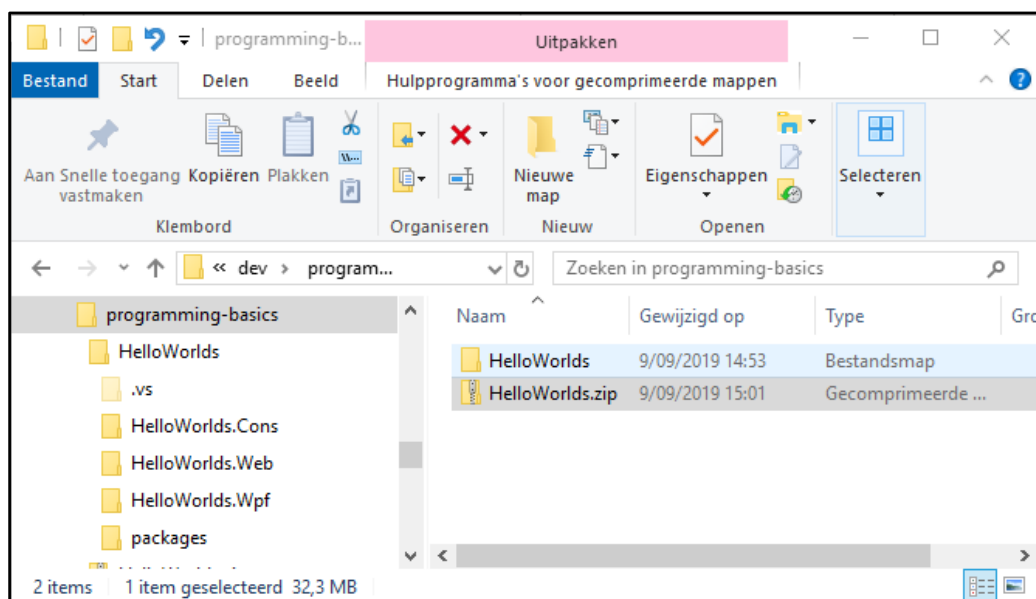
## 4.4 DE BRONCODE BUNDELEN

Wanneer je taken indient, wordt er van je verwacht dat je alle broncode die bij jouw applicatie(s) hoort bundelt in een zip bestand. Het is daarbij belangrijk dat je bij het zippen daadwerkelijk alle bestanden meeneemt. Ga daarom als volgt te werk:

- Zoek de map waarin het solutionbestand en de projectmappen zich bevinden.  
Ben je onzeker waar deze map zich bevindt? Dan kan je deze openen vanuit Visual Studio door een rechtsklik op de Solution en kies voor **Open folder in File Explorer**. Van hieruit ga je één niveau hoger om de rootmap van je solution te zien:



- Maak een gecomprimeerd ZIP-bestand van deze map: rechtsklik en kies **Verzenden naar → Gecomprimeerde (gezipte) map**. Je kan ook een alternatief ZIP programma gebruiken, zoals 7-Zip.




- Upload het ZIP-bestand naar de dropbox waar je de opdracht moet indienen.
- Indien je zeker wil zijn dat je programma blijft werken, probeer dan het volgende:
  - Pak het zip bestand op een andere locatie op je harde schijf.



- Open de solution en probeer je applicatie(s).

**Code repository**

De volledige broncode van deze applicatie is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-inleiding-hello-worlds.git>

Het werken met git en GitHub wordt verder uitgelegd in de module Continuous Integration Basics.

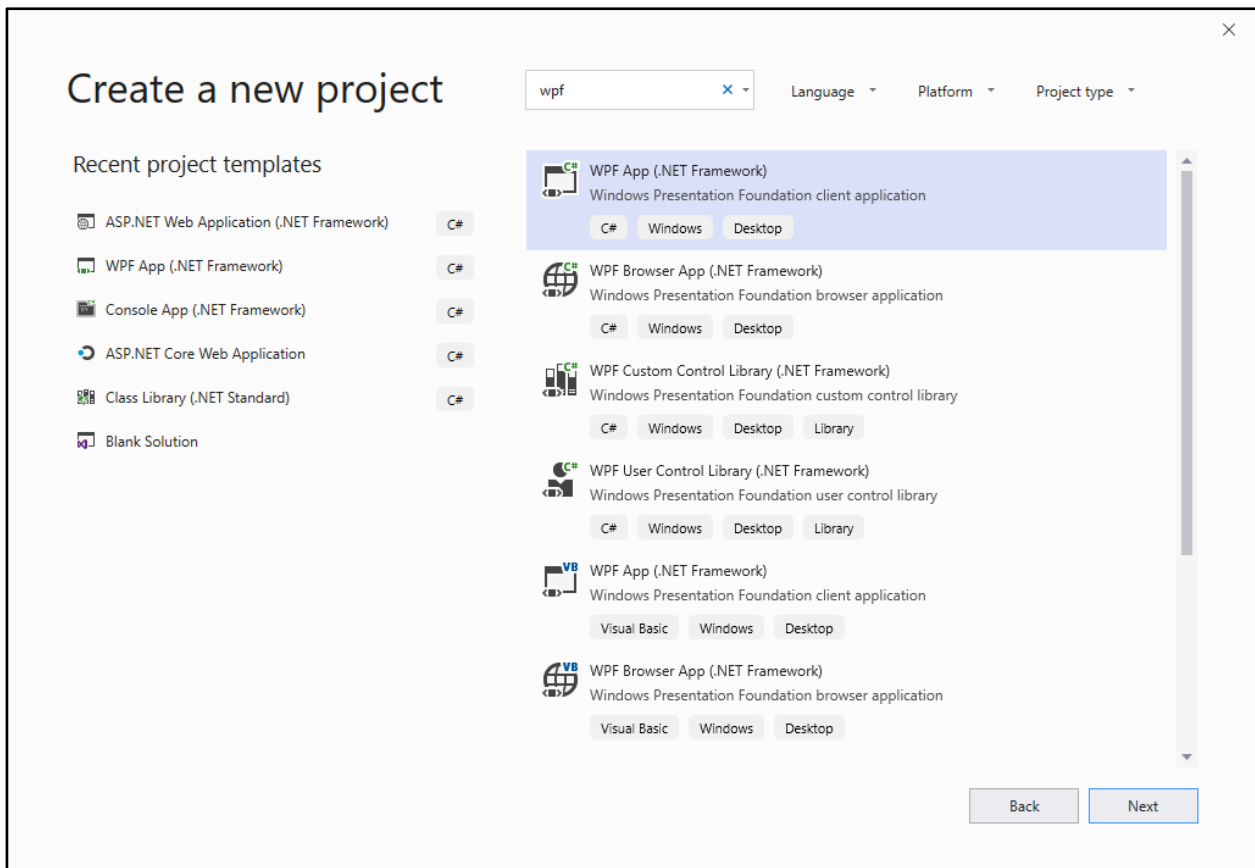
## 5 EEN EERSTE WPF APPLICATION

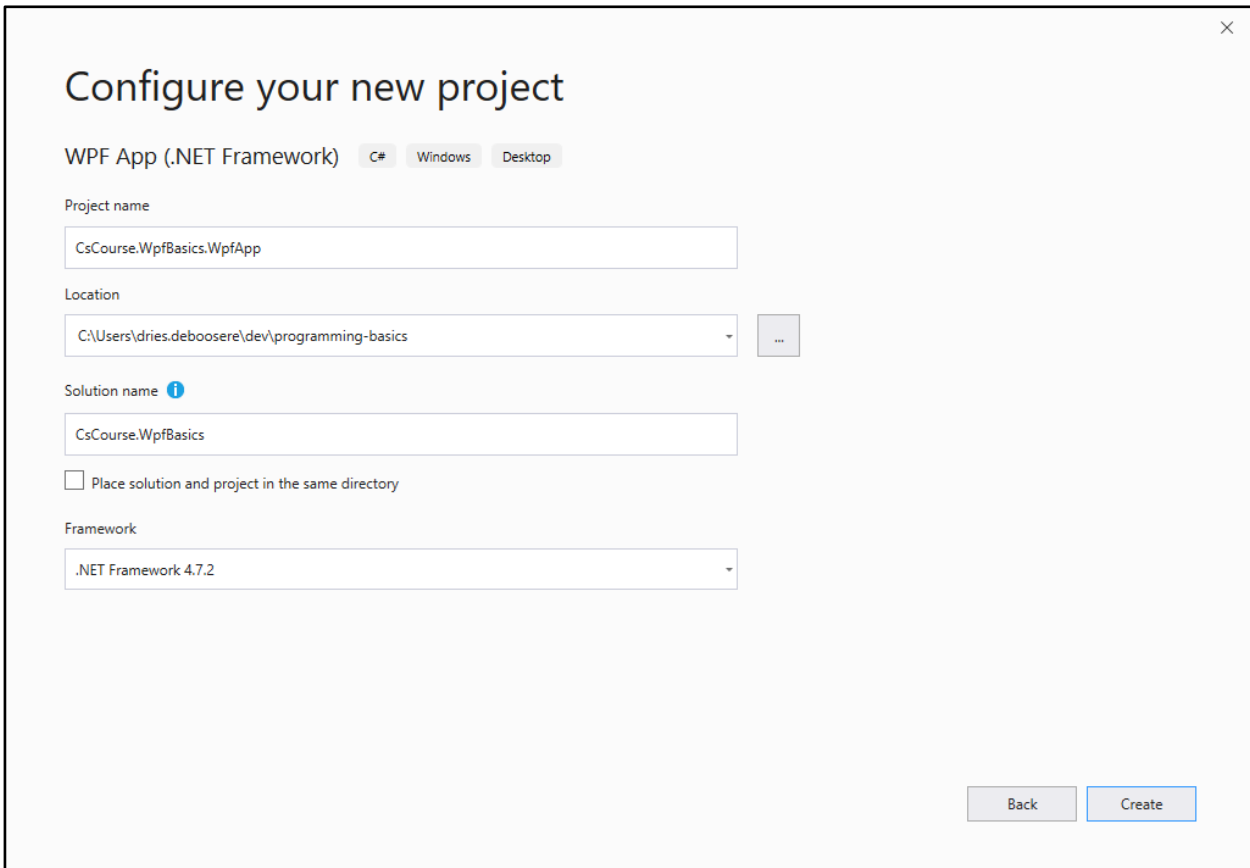
In deze syllabus leer je voornamelijk programmeren met C# aan de hand van WPF applicaties. Om WPF verder te verkennen maak je nu een uitgebreidere, interactieve GUI-applicatie.

In deze applicatie wordt het MainWindow het enige venster dat de gebruiker te zien krijgt. Het zal een tekstvak en een knop bevatten. De gebruiker moet zijn naam intikken in het tekstvak en op de knop klikken. De applicatie begroet de gebruiker dan met de ingevoerde naam.

### 5.1 PROJECT AANMAKEN

1. Start Visual Studio en kies **Create New Project**
2. In het **New Project** dialoogvenster kies je de volgende zaken:
  - 1 Runtime: **.NET Framework 4.7.2**
  - 2 Sjabloon: **WPF App (.NET Framework)**
  - 3 Project name: **CsCourse.WpfBasics.WpfApp**
  - 4 Solution name: **CsCourse.WpfBasics**






Configure your new project

WPF App (.NET Framework) C# Windows Desktop

Project name  
CsCourse.WpfBasics.WpfApp

Location  
C:\Users\dries.deboosere\dev\programming-basics

Solution name   
CsCourse.WpfBasics

☐ Place solution and project in the same directory

Framework  
.NET Framework 4.7.2

Back Create

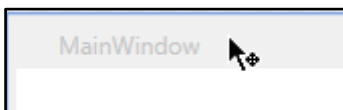
De namen voor het project en de solution zijn volledig vrij te kiezen. Door de naam van de solution algemener te maken, zorg je voor een meer logische indeling indien je zou beslissen om meerdere projecten aan de solution toe te voegen. De punten in de naam zorgen ervoor dat je code automatisch wordt ingedeeld in overeenkomstige namespaces, waarover later meer.

## 5.2 GUI ELEMENTEN TOEVOEGEN EN BEWERKEN

Voor je gaat programmeren verzorg je eerst de lay-out van je GUI-applicatie. Het venster dat getoond wordt bij het uitvoeren van een WPF applicatie is standaard de **MainWindow**.

1. Open **MainWindow.xaml** door erop te dubbelklikken in de Solution Explorer.
2. Wijzig de afmetingen van het Window met de muis:

- Selecteer het Window door een enkele klik in de titelbalk



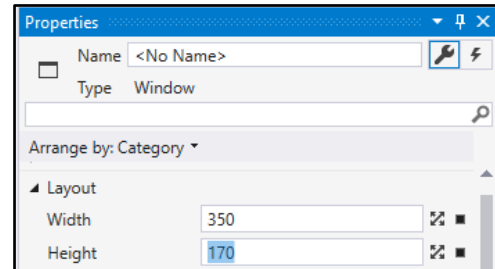
- Sleep het vierkante handvaatje in de hoek rechtsonder naar de gewenste positie.



- Zorg voor een breedte van 300 pixels en een hoogte van 160 pixels.

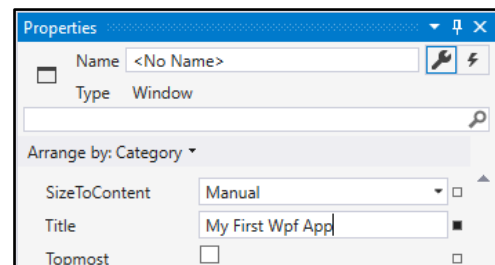
3. Wijzig de afmetingen van het Window via het **Properties** scherm:

- Selecteer het Window door een enkele klik in de titelbalk
- Zoek in het **properties** scherm, in de categorie **Layout**, de eigenschappen **Width** en **Height**.
- Tik de waarden in, Width: **350**, Height: **170** en druk enter om te bevestigen.

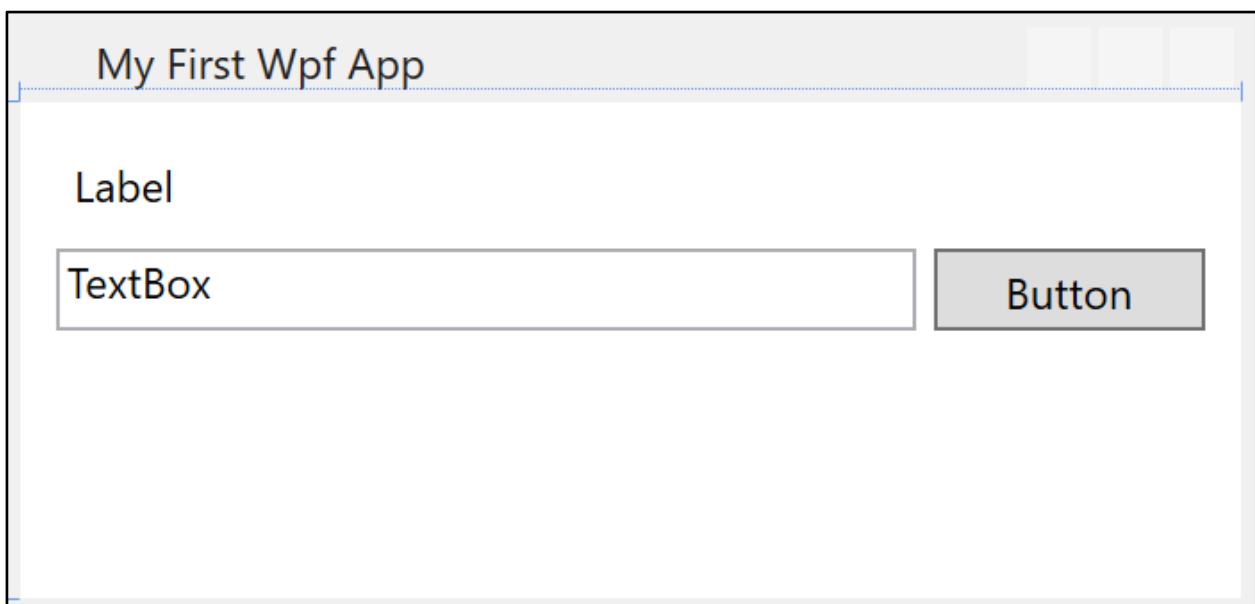


4. Wijzig de titel van het Window via het **Properties** scherm. De titel is zichtbaar in de titelbalk van de applicatie.

- Selecteer het Window door een enkele klik in de titelbalk
- Zoek in het **properties** scherm, in de categorie **Common**, de eigenschap **Title**.
- Geef het venster de titel "My First Wpf App" en druk **enter** om te bevestigen.

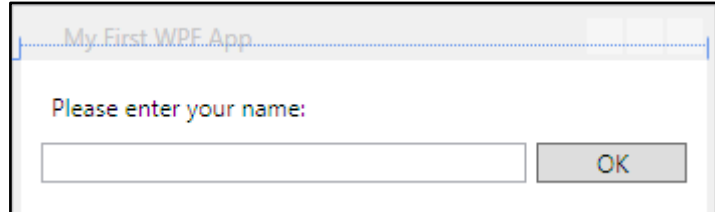


5. Plaats nu vanuit het **Toolbox** venster de volgende controls op het MainWindow: **Label**, **TextBox** en **Button**. Positioneer ze met behulp van de visuele hints om de elementen precies te plaatsen waar jij dat wil.



6. Wijzig de volgende eigenschappen van de Label, de TextBox en de Button door het betreffende element eerst te selecteren met een enkele klik en vervolgens de eigenschappen in het Properties venster aan te passen:

- **Label**
  - Content: "Voer je naam in"
- **TextBox**
  - Text: (blanco maken)
  - Name: txtName
- **Button**
  - Content: "OK"
  - Name: btnOk



Door elementen een **Name** te geven, kan je deze aanspreken vanuit C# code. Het is een goede gewoonte om elementen steeds een Name te geven zodra je ze hebt toegevoegd.

Voor de naamgeving van visuele elementen wordt er meestal de zogenaamde Hongaarse notatie gebruikt, waarbij het type van dat element de prefix vorm. Voor een TextBox is dat bijvoorbeeld **txt** en voor een Button is dat bijvoorbeeld **btn**.

Wij gebruiken de volgende prefixen:

- Label: **lbl**
- TextBox: **txt**
- TextBlock: **tbk**
- ComboBox: **cmb**
- Button: **btn**
- ListBox: **lst**
- StackPanel: **stp**
- Grid: **grd**

Door deze prefixen kunnen we gemakkelijk de namen van controls terugvinden via de intellisense. Door een prefix in te tikken krijgen we de lijst met de controls van een bepaalde soort.

Merk op dat alles wat je in het **Design** scherm en het Properties scherm hebt gedaan een rechtstreekse weerslag heeft in de XAML code van de `MainWindow`:

```
<Grid>
  <Label Content="Please enter your name:" HorizontalAlignment="Left" Margin="10,10,0,0"
    VerticalAlignment="Top"/>
  <Button x:Name="btnOk" Content="OK" HorizontalAlignment="Left" Margin="257,41,0,0"
    VerticalAlignment="Top" Width="75"/>
  <TextBox x:Name="txtName" HorizontalAlignment="Left" Height="20" Margin="10,41,0,0"
    TextWrapping="Wrap" VerticalAlignment="Top" Width="242"/>
</Grid>
```

Ervaren WPF programmeurs kunnen de lay-out dus ook rechtstreeks als XAML code intikken.

### 5.3 EVENTS AFHANDELEN

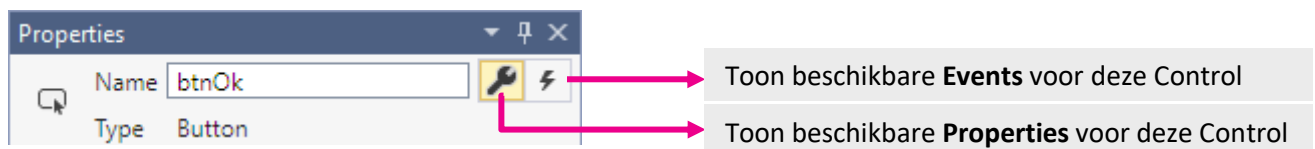
Alle WPF controls, inclusief `Window` hebben zogenaamde **Events** of gebeurtenissen waarop je kan inschrijven zodat jouw programmalogica wordt uitgevoerd.

Een `Window` heeft bijvoorbeeld een **Loaded** event, dat zich voordoet wanneer het venster volledig is ingeladen. Een `TextBox` heeft een **TextChanged** event, dat zich voordoet wanneer de gebruiker tekst heeft gewijzigd in het tekstvak en een `Button` heeft een **Click** event, dat zich voordoet wanneer de gebruiker op de knop klikt.

Om je eigen programmacode uit te voeren wanneer zo'n event zich voordoet moet je een stukje code, in de vorm van een **methode** koppelen aan de betreffende control. Die code wordt dan uitgevoerd zodra het event zich voordoet.

Je wenst nu wat code uit te voeren wanneer de gebruiker op de knop klikt. Je gaat nu als volgt te werk:

1. Selecteer de knop genaamd **btnOk** in de Designer met een enkele klik.
2. Ga naar het **Properties** venster en zoek de knop met bliksem symbool.



3. Klik op de knop met het **bliksempje** om alle beschikbare Events van de Button control weer te geven.
4. Zoek het Event genaamd **Click** en dubbelklik met de muis in het lege tekstvak ernaast.



Eens je dat hebt gedaan gebeuren er twee zaken:

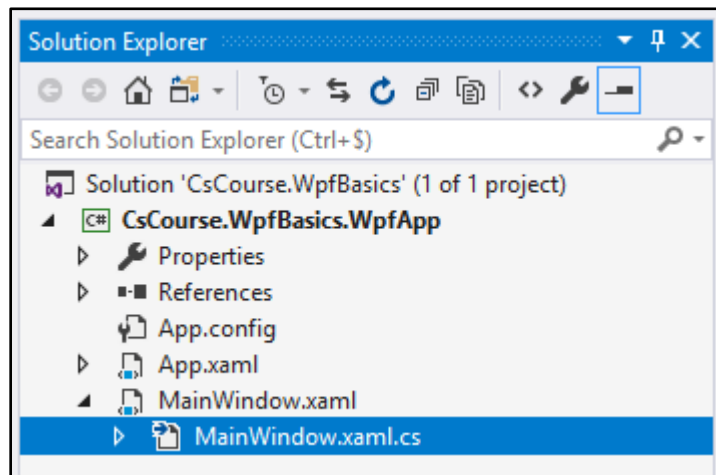
- Er wordt een methode gegenereerd, genaamd `btnOk_Click`
- De methode wordt gekoppeld aan het `Click` event van de knop.

Visual Studio opent nu automatisch het bestand genaamd **MainWindow.xaml.cs**, een C# codebestand waar je de (lege) methode ziet staan:

```
private void BtnOk_Click(object sender, RoutedEventArgs e)
{
}
}
```

Het codebestand, genaamd `MainWindow.xaml.cs` wordt ook wel de **CodeBehind** genoemd. Het bevat de programmacode die rechtstreeks aan de visuele layout van het `MainWindow` is gekoppeld.

Je kan het bestand steeds terugvinden in de Solution Explorer, wanneer je de `MainWindow.xaml` node open tikt. Dat zie je in het volgende screenshot.

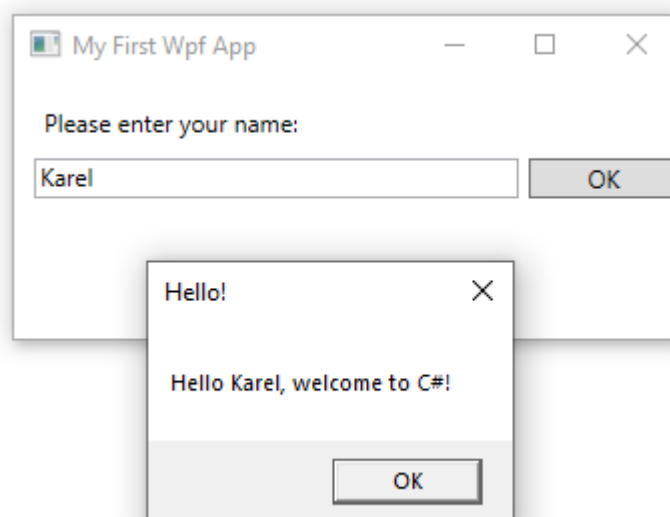


Je wijzigt nu de methode genaamd `BtnOk_Click` als volgt:

```
//Event Handler method for the Button Click event
private void BtnOk_Click(object sender, RoutedEventArgs e)
{
    //display a messagebox (popup dialog) with the user's name
    MessageBox.Show("Hello " + txtName.Text + ", welcome to C#!", "Hello!");
}
```

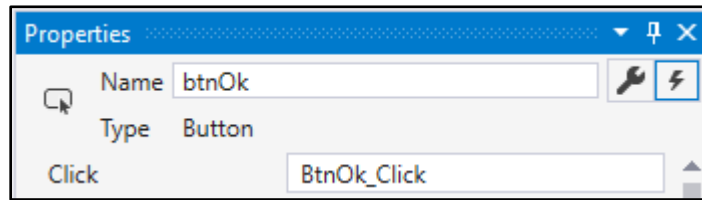
De toegevoegde code bevat een aantal commentaarlijnen om duidelijk te maken wat we doen, en een instructie die een `MessageBox` oproept. De inhoud van de `MessageBox` bevat wat tekst die onder meer de ingevoerde naam van de gebruiker toont.

5. Voer de applicatie uit met **F5** en test de functionaliteit.



Een dergelijke methode, gekoppeld aan een event, wordt ook wel een Event Handler genoemd. In het Design venster, `MainWindow.xaml` zie je nu ook de naam van die methode staan naast het Event in het Properties venster.






Je hebt nu een eerste basis applicatie in WPF gemaakt die gebruikersinteractie toestaat door middel van een GUI. De meeste voorbeelden en opdrachten in deze syllabus zullen deze basishandelingen vereisen. Zorg dus dat je dit goed onder de knie hebt.



### Code repository

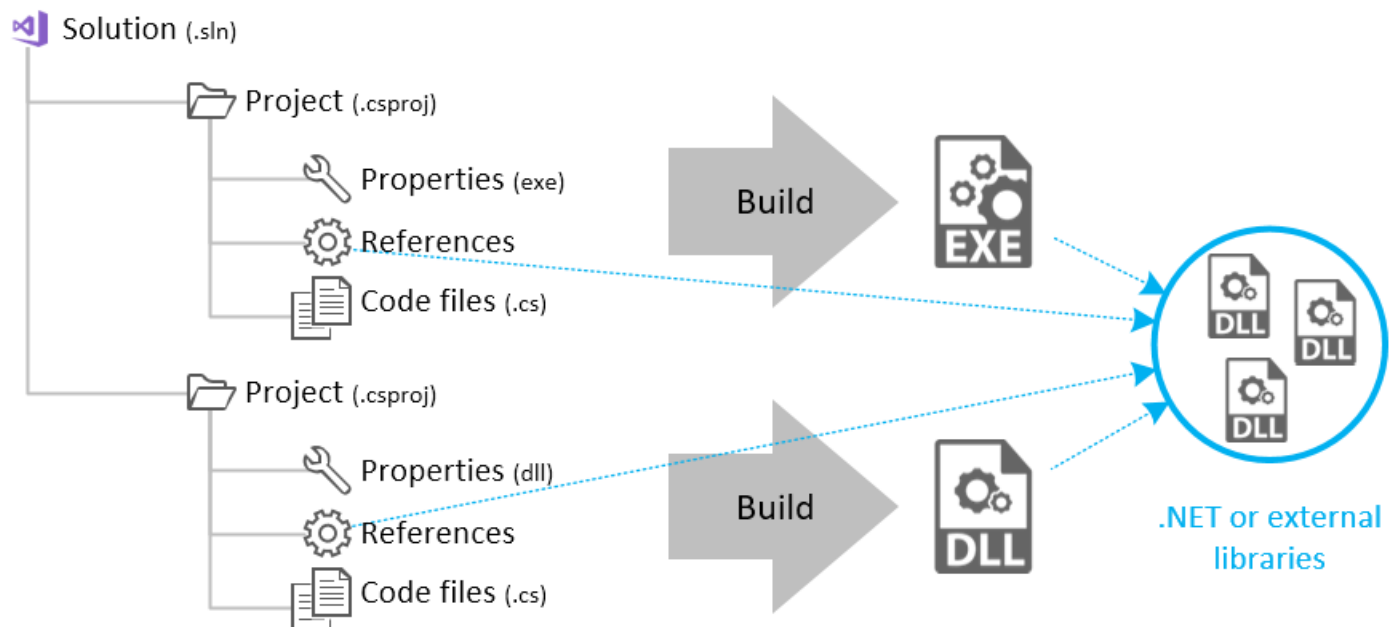
De volledige broncode van deze applicatie is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-inleiding-wpf-basics.git>

## 6 WPF REFERENTIEMATERIAAL

### 6.1 SOLUTIONS EN PROJECTEN

Om de relaties tussen Solutions, Projecten, assemblies en het .NET framework te herhalen kan je het volgende schema beschouwen:



### 6.2 REFERENTIEMATERIAAL

Deze syllabus bevat geen volledig overzicht van alle WPF Controls, hun properties en events. Het is ook zinloos om deze allemaal uit het hoofd te kennen. Een goede programmeur weet zich te behelpen met onderzoekwerk en experiment. De onderstaande bronnen vormen hier een goed startpunt voor, die je gedurende deze module kan blijven benutten.



#### Meer informatie

- [wpf-tutorial.com](http://wpf-tutorial.com)  
Een handleiding voor de meeste gebruikte WPF Controls
- [wpftutorial.net](http://wpftutorial.net)  
Een uitgebreide handleiding voor WPF
- [MSDN: System.Windows.Controls](https://msdn.microsoft.com/en-us/library/system.windows.controls.aspx)  
Microsoft's overzicht van alle WPF Controls
- 

Wie op een vlotte manier WPF layouts wenst te maken, moet zeker een kijkje nemen in het tutorial over WPF Panels. Dat is te vinden op <http://www.wpf-tutorial.com/panels/introduction-to-wpf-panels>

# **HOOFDSTUK 2 STATEMENTS, VARIABELEN EN OPERATOREN**



## INHOUDSOPGAVE

<b>1</b>	<b>STATEMENTS</b>	<b>47</b>
<b>2</b>	<b>IDENTIFIERS</b>	<b>48</b>
<b>3</b>	<b>VARIABELEN</b>	<b>49</b>
<b>3.1</b>	<b>Filmpje</b>	<b>49</b>
<b>3.2</b>	<b>Variabelen gebruiken</b>	<b>49</b>
<b>3.3</b>	<b>Variabelen benoemen</b>	<b>49</b>
<b>3.4</b>	<b>Variabelen declareren</b>	<b>49</b>
3.4.1	Voorbeeld:	50
<b>3.5</b>	<b>Scope van een variabelen</b>	<b>53</b>
<b>4</b>	<b>PRIMITIEVE DATATYPES</b>	<b>55</b>
<b>4.1</b>	<b>Omzetting van het ene datatype/object naar het andere</b>	<b>57</b>
4.1.1	Van ... naar ...	57
4.1.2	Van ... naar string.	58
4.1.3	Van een string naar een getal	58
<b>4.2</b>	<b>Impliciet getypeerde lokale variabelen</b>	<b>58</b>
<b>5</b>	<b>WISKUNDIGE OPERATOREN</b>	<b>59</b>
<b>5.1</b>	<b>Overzicht</b>	<b>59</b>
<b>5.2</b>	<b>Toepassing</b>	<b>59</b>
<b>6</b>	<b>SAMENGESTELDE TOEKENNING</b>	<b>64</b>



## 1 STATEMENTS

Een statement is een opdracht die een actie uitvoert. Meerdere statements kunnen gegroepeerd worden in methoden.

Statements volgen in C# een set strikte regels die hun opbouw beschrijft. Deze regels worden algemeen **syntax** genoemd. Een fout tegen de syntactische regels van een programmeertaal is dan een '**syntax error**'. Eén van deze syntaxregels is het feit dat een statement steeds gevolgd wordt door een **;** (puntkomma / semicolon).

Voorbeeld van een statement:

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

Voorbeeld van een methode met twee statements:

```
public void VerdubbelTwee()
{
    Console.WriteLine("Het dubbele van 2 is 4");
    Console.ReadLine();
}
```

## 2 IDENTIFIERS

Elementen zoals namespaces, klassen, methoden, variabelen (zie later), controls krijgen in C# een naam. De naam die je aan een element geeft, noemen we de **identifier**.

Een identifier moet aan volgende regels voldoen

- Enkel letters (kleine en hoofdletters), cijfers en het teken \_ (underscore) zijn toegelaten
- Het eerste teken mag geen cijfer zijn
- C# is hoofdlettergevoelig: **resultaat** is niet gelijk aan **Resultaat**
- C# reserveert een aantal woorden (**reserved identifiers**) voor intern gebruik, deze kan je niet opnieuw gebruiken als identifier.

Een lijst van dergelijke sleutelwoorden vind je bij [MSDN](#). (Microsoft Developers Network)



### Identifiers

Een lijst van dergelijke sleutelwoorden vind je bij:

- [MSDN \(Microsoft Developers Network\)](#)



## 3 VARIABELEN

### 3.1 FILMPJE

Op Udacity.com vind je een filmpje waarin heel eenvoudig wordt uitgelegd wat een variabele is. Je vindt het in de cursus 'Android Basics: User Input', Making an App Interactive: Part 1, 9. Quiz: The Need for Variables. Eens je je aangemeld hebt voor deze gratis cursus, kun je deze [link](#) gebruiken.

### 3.2 VARIABELEN GEBRUIKEN

Een variabele is een opslagruimte die een waarde kan bevatten. Je kan een variabele zien als een plekje geheugen dat je een naam hebt gegeven. Dit geheugenplekje bevat informatie die gewijzigd kan worden in de loop van het programma. Een variabele kan tekst, getallen, datums ... bevatten.

Tijdens het programmeren zal je veelvuldig gebruik maken van variabelen.

### 3.3 VARIABELEN BENOEMEN

Een variabele wordt steeds aangesproken met de naam. Het is belangrijk dat je voor jezelf een goede manier ontwikkelt om variabelen een naam te geven. Niets is zo vervelend als programmacode te moeten lezen die vol staat van variabelennamen zoals a, b, c, ... of getal1, getal2, getal3, ... .

Bij het lezen van dergelijke code moet je je als programmeur steeds gaan afvragen welke informatie de variabele bevat. Veel duidelijker zijn variabelennamen zoals aankoopBedrag, verkoopBedrag, winst, btwBedrag, totaal, ...

Vaak worden bij naamgeving van variabelen volgende regeltjes toegepast (dit is geen verplichting, veeleer een afspraak):

- Begin een variabelennaam niet met een hoofdletter
- Gebruik **camelCase** notatie: start bij namen die bestaan uit meerdere woorden elk woord met een hoofdletter (niet het eerste), alle andere letters zijn kleine letters.
- Gebruik **geen** Hongaarse notatie: een (vroeger) veel gebruikt manier om variabelen een naam te geven, waarbij aan de naam een prefix werd geschreven die duidt op het type van de variabele.

### 3.4 VARIABELEN DECLAREREN

Variabelen bewaren waarden. In C# kunnen veel verschillende types van waarden opgeslagen en verwerkt worden: gehele getallen (integer), kommagetallen (floating-point number), tekst (string), tekens (character), ...

Bij de declaratie van een variabele geef je aan welk type gegevens de variabele zal gaan bijhouden. In het declaratie-statement bepaal je het type en de naam van een variabele:

```
int leeftijd;
string familieNaam;
```

Hier declareer je een variabele met de naam 'leeftijd' van het type integer (geheel getal) en een variabele met de naam 'familieNaam' van het type string (tekenreeks/tekst).

- Merk nogmaals op dat elk statement eindigt met een komma-punt.
- Het type `int` is één van de primitieve datatypes binnen C# (meer hierover verderop).
- In C# ben je verplicht elke variabele te declareren vooraleer je deze gebruikt!
- Nadat je een variabele hebt gedeclareerd of bij de declaratie kan je deze een waarde toewijzen:

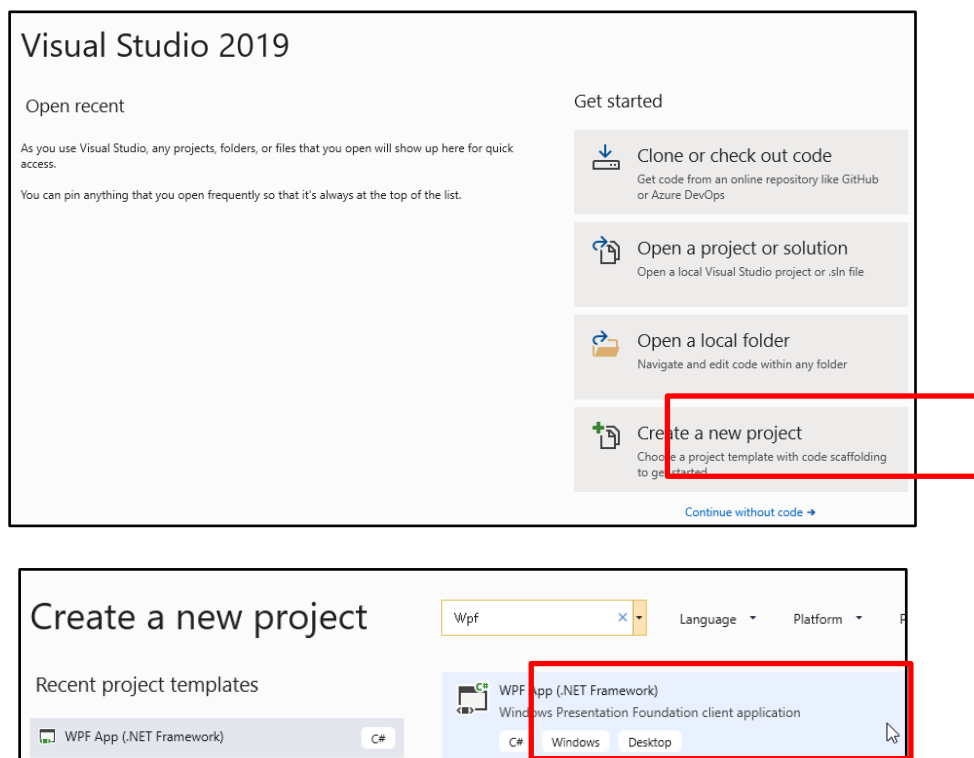
```
leeftijd = 42;
int prijs = 15;
```

- Het gelijkheidsteken is in C# de toekenningoperator: je kent de waarde 42 toe aan de variabele `leeftijd`.
- Je kan in C# een variabele declareren en initialiseren (een initiële waarde toekennen) in één coderegel:

```
int age = 42;
```

### 3.4.1 VOORBEELD:

Maak een nieuw project aan met de naam `WpfHello2`.




## Configure your new project

WPF App (.NET Framework) C# Windows Desktop

Project name  
WpfHello2

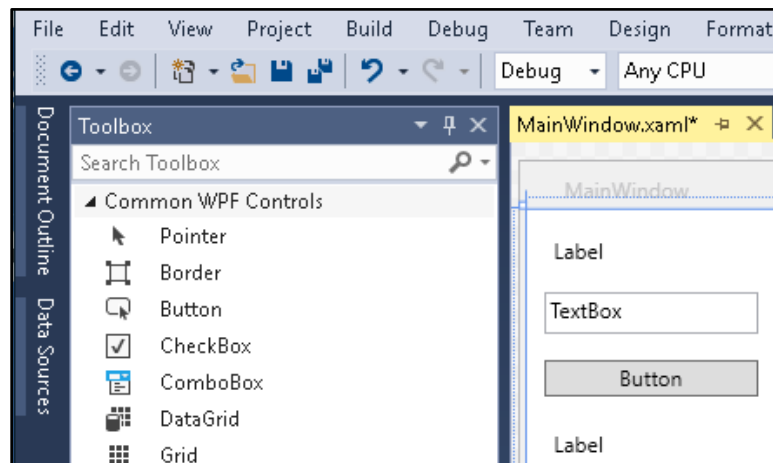
Location  
D:\PB\

Solution name   
WpfHello2

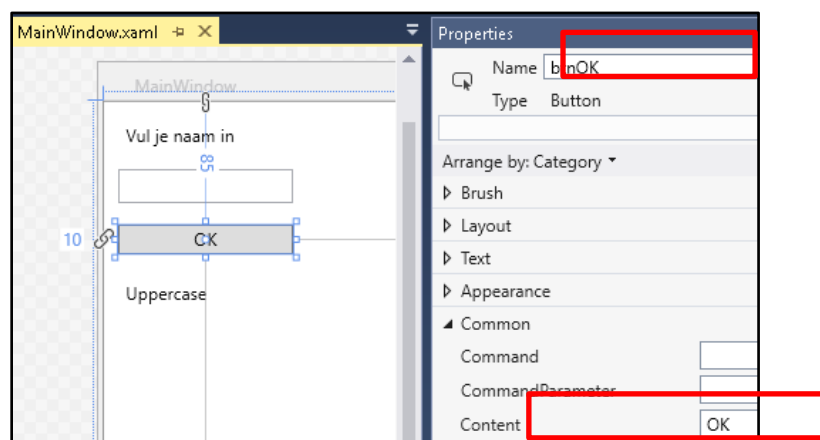
☒ Place solution and project in the same directory

Framework  
.NET Framework 4.6.1

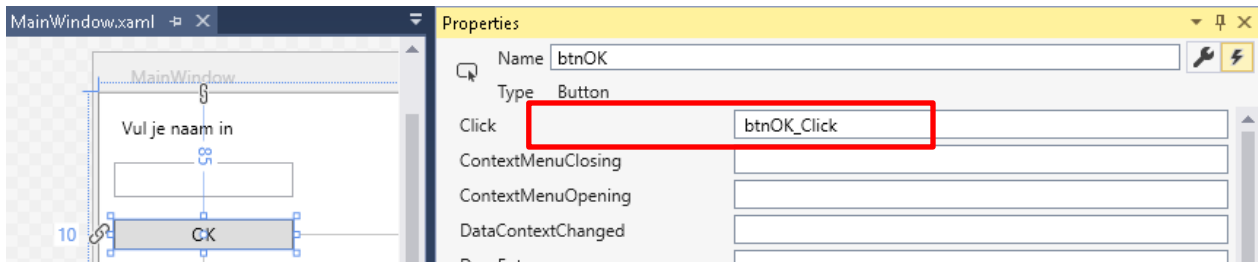
Sleep een textbox, een button en twee labels naar het formulier.



Pas de naam (txtUsername, btnOK, lblHoofdletters) en de tekst van de objecten aan.



Pas de methode BtnOK\_Click aan:



```
private void BtnOK_Click(object sender, RoutedEventArgs e)
{
    string name = txtUsername.Text;
    name = name.ToUpper();
    lblHoofdletters.Content = name;
}
```

Declareer een string met de naam *name*. Initialiseer deze string met de waarde van de eigenschap `Text` van de control `txtUserName`: de `TextBox`.

```
string name = txtUserName.Text;
```

Ken aan de variabele *name* de waarde toe van de variabele *name* waarop de methode `ToUpper()` werd toegepast: alle letters worden hoofdletters:

```
name = name.ToUpper();
```

### 3.5 SCOPE VAN EEN VARIABLEN

Als we een variabele declareren is die geldig voor gans het stuk code waarin ze gedeclareerd wordt. Een variabele kan geldig zijn binnen een codeblok (bvb. een lus of selectieblok waarover later), een methode, een klasse of een project. Eens het codeblok, de procedure ... uitgevoerd zijn, verdwijnt de variabele uit het geheugen.

De volgende codevoorbeelden maken dit duidelijk.

**Voeg nog een label toe aan je MainWindow en noem dit lblFeedback.**

Hieronder wordt een variabele naam gedeclareerd in de methode `TxtUsername_TextChanged`.

```
namespace CsCourse.WpfBasics.WpfApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void TxtUsername_TextChanged(object sender, TextChangedEventArgs e)
        {
            string naam;
            naam = txtUsername.Text;
            lblFeedback.Content = "Je naam is " + txtUsername.Text;
        }
    }
}
```

Een de methode doorlopen, is de informatie die in *naam* zat terug verdwenen. Willen we iets met die variabele aanvangen onder een button bvb., dan moeten we terug de tekst uit `txtUsername` gaan oproepen.

```
private void BtnOK_Click(object sender, RoutedEventArgs e)
{
    string naam;
    naam = txtUsername.Text;
    MessageBox.Show("Ben jij werkelijk dé " + naam);
}
```

Eigenlijk hebben we nu twee variabelen met dezelfde naam, die een onafhankelijk leven leiden van elkaar. Als we de variabele op het niveau van de partial class `MainWindow` declareren, blijft ze

beschikbaar zolang de MainWindow actief is. Vanuit elk codeonderdeel kan de waarde van *naam* dan opgevraagd of aangepast worden.

```
public partial class MainWindow : Window
{
    string naam;

    public MainWindow()
    {
        InitializeComponent();
    }

    private void TxtUsername_TextChanged(object sender, TextChangedEventArgs e)
    {
        naam = txtUsername.Text;
        lblFeedback.Content = "Je naam is " + txtUsername.Text;
    }

    private void BtnOK_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Ben jij werkelijk dé " + naam);
    }
}
```

In de bovenstaande code kunnen we de variabele naam overal binnen MainWindow gebruiken. In txtUsername\_TextChanged kennen we een waarde toe, in BtnOK\_Click vragen we de waarde op.



#### Code repository

De volledige broncode van deze oefening is te vinden op

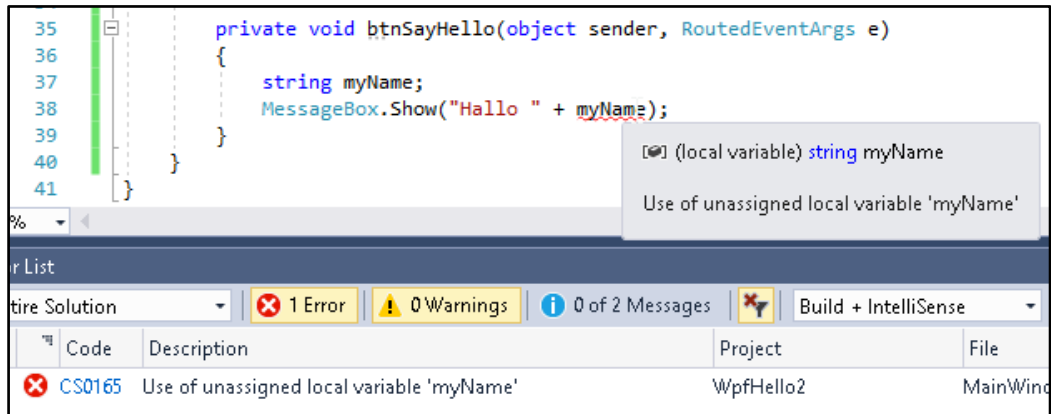
git clone <https://github.com/howest-gp-prb/cu-variabelen-wpf-hello-2.git>

## 4 PRIMITIEVE DATATYPES

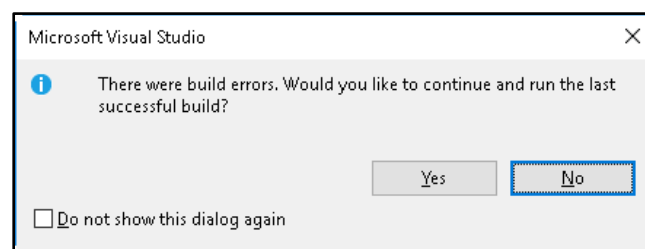
C# heeft een aantal ingebouwde types die we primitieve datatypes noemen:

data type	omschrijving	grootte (bits)	bereik	voorbeeld
byte	positieve gehele getallen	8	0 – 255	byte wielen; wielen = 4;
sbyte	gehele getallen	8	-128 - 127	sbyte verdieping; sbyte = -2;
short	gehele getallen	16	-32 768 to 32 767	short loon loon = 1500
int	gehele getallen	32	$-2^{31}$ tot $2^{31}$	int count; count = 42;
long	gehele getallen	64	$-2^{63}$ tot $2^{63}$	long wait; wait = 42L;
float	kommagetallen (pos / neg)	32	$1,5 \times 10^{-45}$ tot $3,4 \times 10^{38}$	float test; test = 0.42F;
double	precieze kkommagetallen (pos / neg)	64	$5 \times 10^{-324}$ tot $1,7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	geldbedragen	128	28 betekenisvolle cijfers	decimal coin; coin = 0.42M;
string	tekenreeks	16 bits / teken	nvt	string auto; auto = "Honda";
char	1 teken	16	0 tot $2^{16} - 1$	char geslacht; geslacht = 'V';
bool	booleaanse waarde: waar / onwaar	8	Waar / onwaar ( 0 / 1)	bool gevonden; gevonden = false;

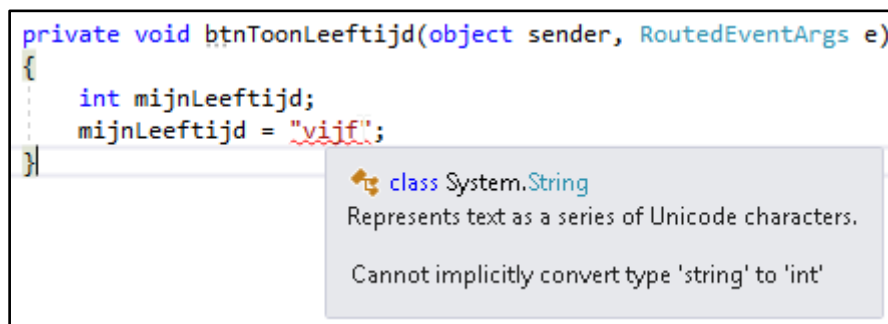
In C# krijg je een error wanneer je een niet toegekende variabele wenst te gebruiken:



Als je toch probeert om de code te bouwen, krijg je een foutboodschap:

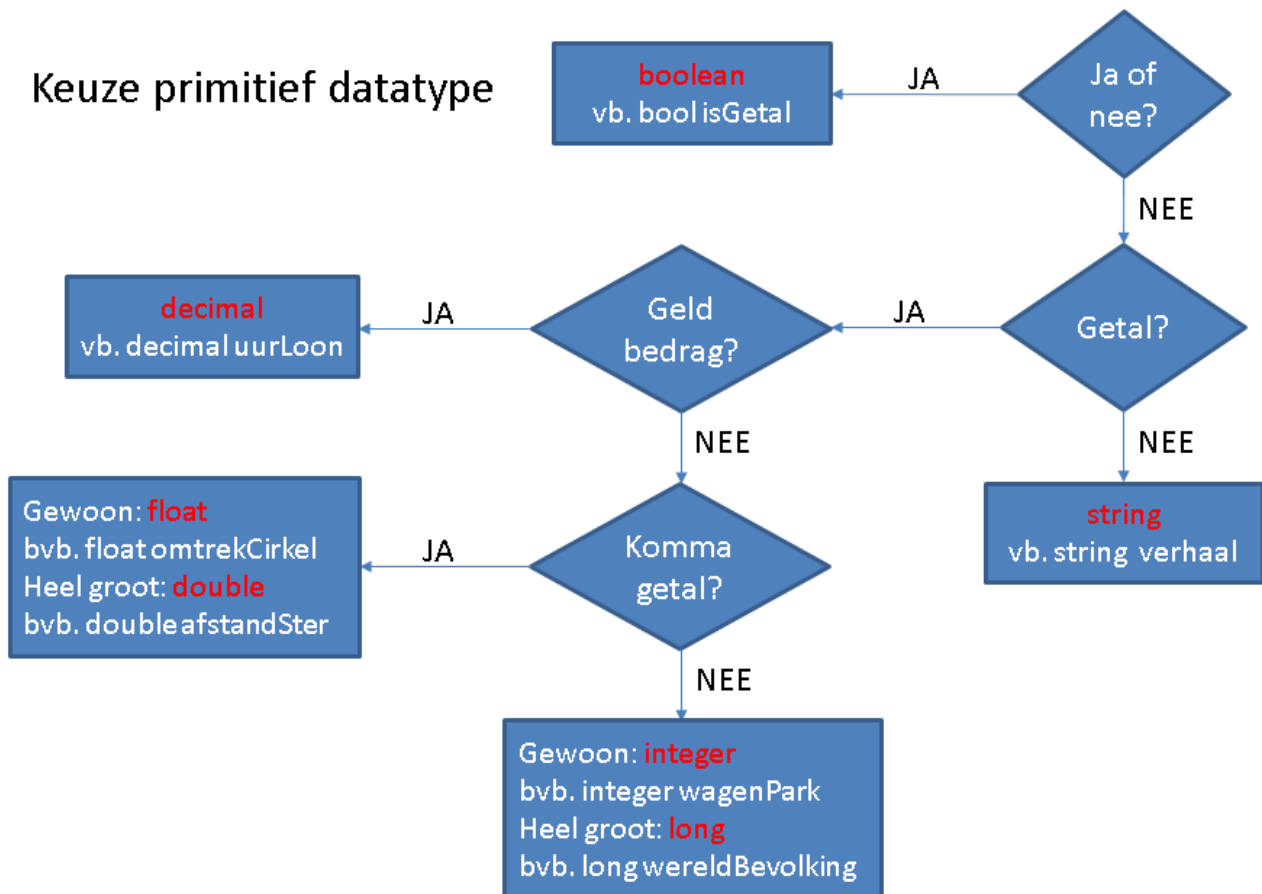


Ook als je een ongeldige waarde aan een variabele wil toekennen, wijst Visual Studio je terecht:





## Keuze primitief datatype



## 4.1 OMZETTING VAN HET ENE DATATYPE/OBJECT NAAR HET ANDERE

### 4.1.1 VAN ... NAAR ...

#### CONVERT

Met de Convert-functie kun je allerlei objecten omzetten naar allerlei andere types, zoals boolean, string, getallen...

Hier converteren we bijvoorbeeld een getal naar een boolean. Nul wordt dan false, alle andere getallen true. Was alles maar zo simpel in het leven...

```
int getal = 0;
bool trueOrFalse = Convert.ToBoolean(getal);
Console.WriteLine("Getal = " + trueOrFalse);
```

#### CASTING

Casting is een conversie van het ene type naar het andere door vóór het te converteren object het gewenste type tussen haakjes te plaatsen.

```
long getal = 0;
int nummer = (int)getal;
```

Net hetzelfde resultaat zou je bekomen met

```
long getal = 0;
int nummer = Convert.ToInt32(getal);
```

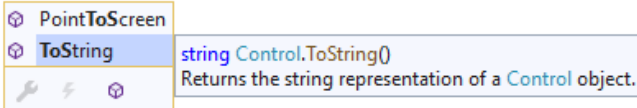
Met casting kunnen we ook meer complexe conversies uitvoeren.

```
string beginVanDeZin;
ListBoxItem stad;
stad = (ListBoxItem)lstSteden.SelectedValue;
```

#### 4.1.2 VAN ... NAAR STRING.

Aan de datatypes die we hierboven leerden kennen, zijn mogelijke functies gekoppeld. Zo kun je in een string karakters opsporen en veranderen, de lengte bepalen... Heb je een object waarop je die functies wil uitoefenen, dan kun je de functie ToString() er op toepassen.

```
Button test = new Button();
string testString = test.tos
```



In bovenstaand voorbeeld kun je op testString alle functies van een string uitoefenen.

#### 4.1.3 VAN EEN STRING NAAR EEN GETAL

##### PARSE

Soms krijg je in je code een numerieke waarde binnen in string-formaat. Wil je daarop wiskundige bewerkingen uitvoeren, dan moet je die tekst omzetten naar een getal, parsen in het jargon.

```
string nummer = "1";
int getal = int.Parse(nummer);
Console.WriteLine("Getal = " + getal);
```



##### Meer informatie

- [Meer info bij Microsoft](#)

## 4.2 IMPLICIET GETYPEERDE LOKALE VARIABELEN

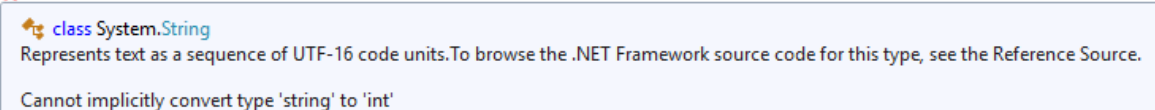
C# ondersteunt ook het gebruik van het sleutelwoord var voor de declaratie van variabelen:

```
var mijnIntVar = 5;
var mijnStringVar = "Hallo";
```

Nu zal het .Net Framework na initialisatie zelf het type toekennen.

Een variabele gedeclareerd met var moet onmiddellijk geïnitieerd worden. Het type kan achteraf niet meer gewijzigd worden.

```
var mijnLeeftijd = 5;
mijnLeeftijd = "vijf";
```



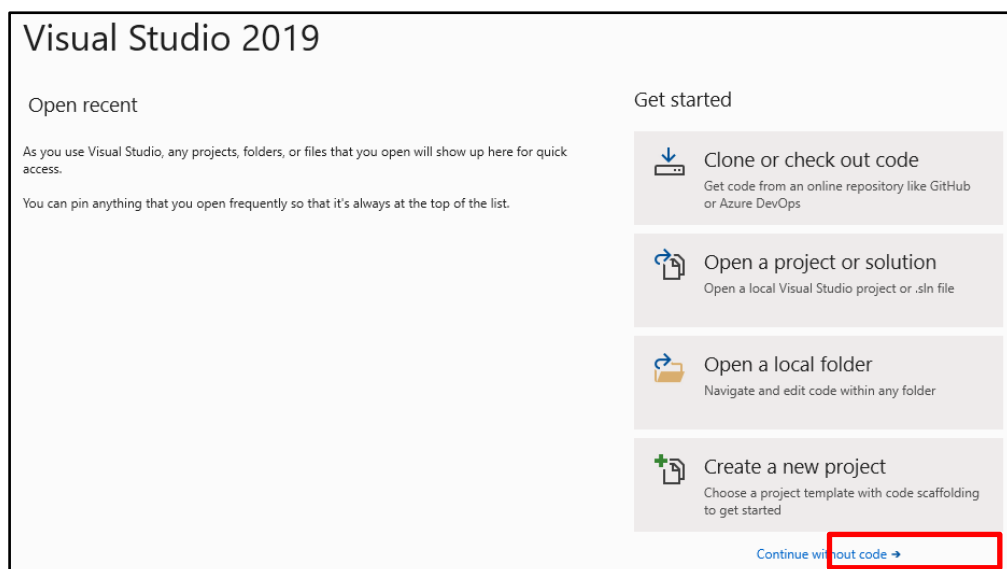
## 5 WISKUNDIGE OPERATOREN

### 5.1 OVERZICHT

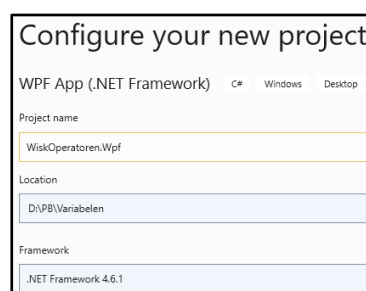
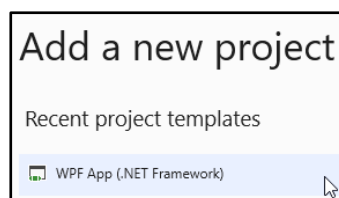
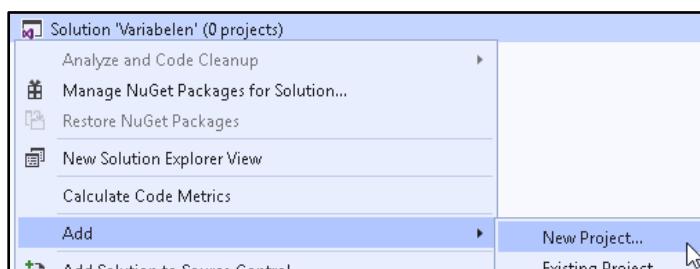
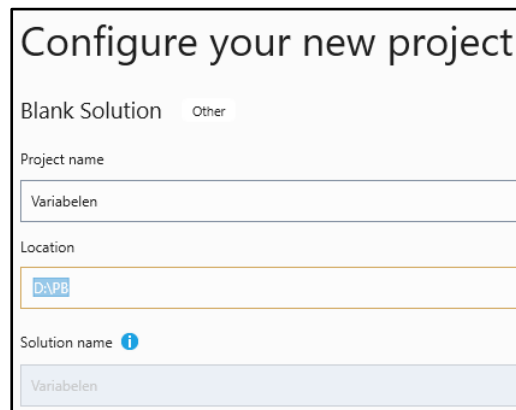
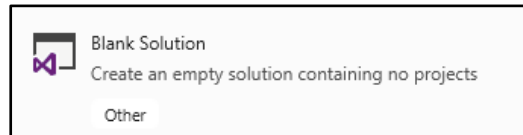
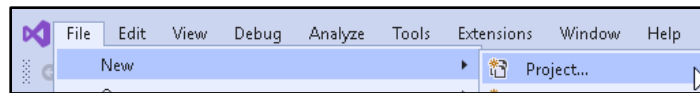
operator	beschrijving	voorbeeld
+	optellen	int som; som = 5 + 3;
-	afrekken	int verschil; verschil = 5 - 3;
*	vermenigvuldigen	int product; product = 5 * 3;
/	delen	double quotient; quotient = 5 / 3;
%	rest bij gehele deling (modulo)	int rest; rest = 5 % 3;
++	verhoog met 1	int i; i = 1; i++;
--	verminder met 1	int i; i = 10; i--;

### 5.2 TOEPASSING

#### 7. Maak een nieuwe Solution Variabelen aan in Visual Studio



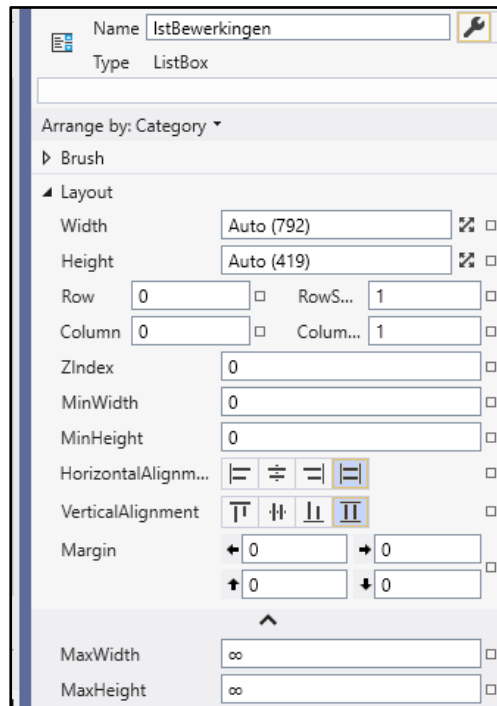
8. Voeg een nieuw project toe met de naam WiskOperatoren.Wpf



9. Sleep vanuit de ToolBox een ListBox binnen de Grid van het WPF-Window en geef hem de naam 1stBewerkingen.

Een ListBox kan je in WPF gebruiken om een lijst van waarden weer te geven. Het doel van de toepassing is een aantal zinnen onder elkaar op het scherm te tonen.

10. Sleep de randen van de listbox tegen de randen van de grid
11. Pas eventueel de waarden voor de eigenschappen Margin, HorizontalAlignment, VerticalAlignment, Width en Height aan bij de properties (**F4**) zoals hieronder getoond.

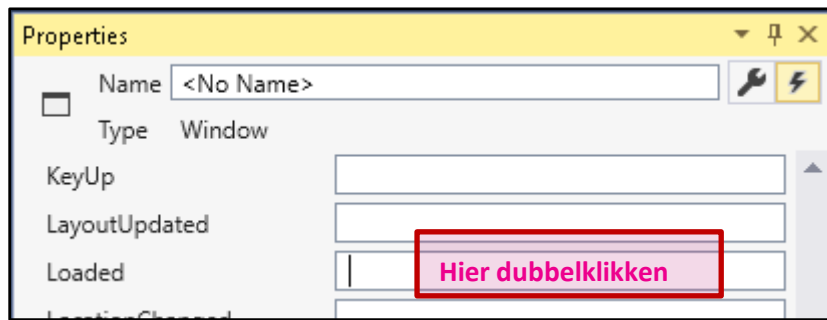


De ListBox vult nu het volledige Window.

In de XAML-code zien we dit:

```
<Window x:Class="WiskOperatoren.Wpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <ListBox x:Name="lstBewerkingen"/>
    </Grid>
</Window>
```

1. Selecteer het Window MainWindow en klik het tabbladje **Events**.
2. Dubbelklik naast het event Loaded:



De methode `Window_Loaded` zal uitgevoerd worden wanneer het Window volledig geladen is. De programmacode die we schrijven voor het event `Loaded` van een Window zal dus meteen uitgevoerd worden als het Window geladen (opgestart) wordt.

3. In `MainWindow.xaml.cs` werd nu volgende code automatisch voorzien:

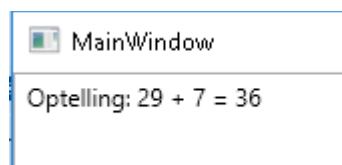
```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
}
```

De methode `Window_Loaded`.

4. Voeg volgende code toe, gebruik waar mogelijk intellisense van Visual Studio:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    lstBewerkingen.Items.Add("Optelling: 29 + 7 = " + (29 + 7));
}
```

5. Voer de toepassing uit:

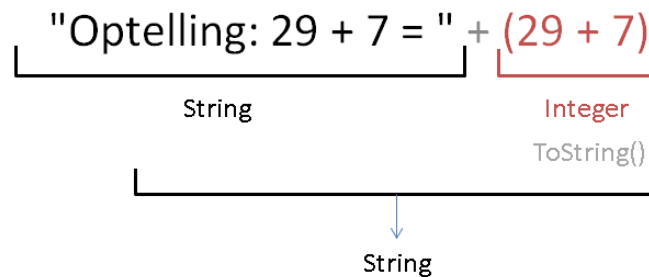


De naam van de `ListBox` die je in het Window hebt geplaatst, is `lstBewerkingen`. Een `ListBox` kan een lijst van elementen op het scherm tonen. In de code kan je hem aanspreken/manipuleren met zijn naam.

De eigenschap `Items` van de `ListBox` `lstBewerkingen` bevat de elementen die in de `ListBox` aanwezig zijn (`ItemCollection`). Met de methode `Add` van de `ItemCollection` kun je een element aan de `ListBox` toevoegen. Dit is hetgeen gebeurt met de opdracht `bewerkingen.Items.Add(...)`.

Uit dit voorbeeld kan je leren dat de operator + voor getallen voorziet in de som, en voor strings leidt tot een samenvoeging van de tekenreeksen (concatenering):

- "Optelling: 29 + 7 = " is een string
- (29 + 7) is de optelling van twee gehele getallen 29 en 7 en levert 36.
- De eerste tekenreeks wordt nu met de operator + samengevoegd met het gehele getal 36; hiervoor wordt het gehele getal achter de schermen omgezet in een string (methode ToString()-zie later), de operator + voor twee tekenreeksen leidt tot een samenvoeging van de twee tekenreeksen.



#### 6. Voeg nog een reeks strings toe aan de ListBox:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    lstBewerkingen.Items.Add("Optelling: 29 + 7 = " + (29 + 7));
    lstBewerkingen.Items.Add("Aftrekking: 29 - 7 = " + (29 - 7));
    lstBewerkingen.Items.Add("Vermenigvuldiging: 29 * 7 = " + (29 * 7));
    lstBewerkingen.Items.Add("Deling: 29 / 7 = " + (29F / 7F));
    lstBewerkingen.Items.Add("Modulo : 29 % 7 = " + (29 % 7));
    int getal = 0;
    lstBewerkingen.Items.Add("Waarde van getal: " + getal);
    lstBewerkingen.Items.Add("Verhogen met 1 : ++getal = " + ++getal);
    lstBewerkingen.Items.Add("Verlagen met 1 : --getal = " + --getal);
    lstBewerkingen.Items.Add("2 tot de 3de macht: " + Math.Pow(2, 3));
    lstBewerkingen.Items.Add("De vierkantswortel van 16: " + Math.Sqrt(16));
}
```

Bij de deling is het interessant op te merken dat minstens één van de leden van de bewerking een kommagetal moet zijn (**suffix F**), anders wordt de gehele deling uitgevoerd en gaan cijfers achter de komma dus verloren.

Wanneer meerdere bewerkingen in één statement worden opgenomen, worden deze als volgt uitgevoerd:

- Rekening houden met eventuele haakjes
- Rekening houden met de volgorde der bewerkingen (vb. eerst vermenigvuldigen / delen, daarna optellen of aftrekken)
- Uitvoering van links naar rechts

## 6 SAMENGESTELDE TOEKENNING

Je weet reeds dat je een waarde aan een variabele kan toekennen met behulp van de toekenningsoperator `=`

```
int aantalLeerlingen;
aantalLeerlingen = 10;
```

Je kunt de wiskundige hoofdbewerkingen uitvoeren:

```
aantalLeerlingen = 10 + 5;
```

Op de onderstaande manier kan je de originele waarde van een variabele aanpassen:

```
aantalLeerlingen = aantalLeerlingen + 8;
aantalLeerlingen = aantalLeerlingen - 2;
aantalLeerlingen = aantalLeerlingen * 3;
aantalLeerlingen = aantalLeerlingen / 3;
aantalLeerlingen = aantalLeerlingen % 4;
```

Dit kan korter door gebruik te maken van een samengestelde toekenningsoperator:

```
aantalLeerlingen += 8;
aantalLeerlingen -= 2;
aantalLeerlingen *= 3;
aantalLeerlingen /= 3;
aantalLeerlingen %= 4;
```

Je kunt dit ook toepassen op andere types variabelen dan getallen.

```
string naam = "Kenji";
string voornaam = "Minogue";
naam += " " + voornaam;
```



### Code repository

De volledige broncode van deze applicatie is te vinden op

`git clone` <https://github.com/howest-gp-prb/cu-variabelen-wiskundige-operatoren.git>



# HOOFDSTUK 3

## METHODEN



## INHOUDSOPGAVE

<b>1</b>	<b>METHODEN DECLAREREN</b>	<b>69</b>
<b>1.1</b>	<b>Voorbeeld: methode ToonLand</b>	<b>69</b>
<b>2</b>	<b>EEN METHODE AANROEPEN</b>	<b>70</b>
<b>3</b>	<b>PARAMETERS</b>	<b>72</b>
<b>4</b>	<b>RETURNS</b>	<b>73</b>
<b>4.1</b>	<b>Parameter arrays</b>	<b>76</b>
<b>4.2</b>	<b>Toepassing parameter arrays</b>	<b>76</b>
<b>5</b>	<b>SCOPE</b>	<b>78</b>
<b>6</b>	<b>OVERLOADING</b>	<b>80</b>
<b>7</b>	<b>EVENT HANDLER METHODEN ⇔ EIGEN METHODEN</b>	<b>82</b>
<b>7.1</b>	<b>Event handler methods</b>	<b>82</b>
<b>7.2</b>	<b>Check je eigen methode</b>	<b>84</b>



## 1 METHODEN DECLAREREN

Maak een nieuwe Visual Studio solution **Methoden**.

Maak in de solution Methoden een nieuwe WPF Application **WpfMethoden**.

Methoden worden steeds gedeclareerd binnen een klasse, voor ons dus binnen de accolades.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

De algemene syntax om een methode te declareren gaat als volgt:

```
returnType methodeNaam (parameterLijst)
{
    //statements binnen de methode
}
```

### 1.1 VOORBEELD: METHODE TOONLAND

Maak een methode die het land “België” in een berichtvenster toont.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    void ToonLand()
    {
        string land = "België";
        MessageBox.Show("Het land is " + land);
    }
}
```

- de nieuwe methode heeft de naam ToonLand;
- ToonLand retourneert niks: sleutelwoord void. Bij andere methode wordt als resultaat van de statements een string, getal of welk object ook als ‘uitkomst’ teruggestuurd.
- ToonLand ontvangt geen parameters: lege ronde haken na de naam van de methode.

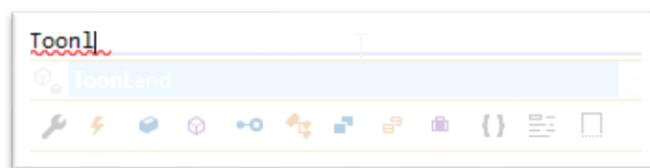
## 2 EEN METHODE AANROEPEN

Een methode aanroepen (laten uitvoeren) gebeurt door de naam van de methode te noteren gevolgd door ronde haakjes met daarbinnen de eventuele parameters (zie verder):

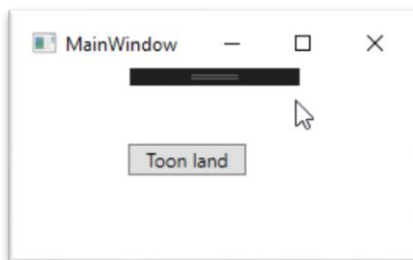
```
public MainWindow()
{
    InitializeComponent();
    ToonLand();
}
```

Hier roep je ToonLand aan vanuit de constructor (zie later) van Window1. Voorlopig is het voldoende te weten dat de code binnen dit codeblok steeds wordt uitgevoerd bij het starten van de toepassing.

Merk op dat je eigen methode onmiddellijk ter beschikking is met Visual Studio Intellisense.



Daar we een grafische toepassing maken in WPF is het natuurlijk leuker de methode te laten aanroepen via bv. een druk op een knop.



- Plaats een button op je Window
- Geef de button de naam "btnToonLand" en de content verander je naar "Toon land"
- Dubbelklik op deze button, een event-handlermethode wordt automatisch aangemaakt:

```
private void BtnToonLand_Click(object sender, RoutedEventArgs e)
{
}
}
```

- Vanuit deze methode kan je je eigen methode ToonLand aanroepen:

```
private void BtnToonLand_Click(object sender, RoutedEventArgs e)
{
    ToonLand();
}
```

- Verwijder de aanroep van ToonLand uit de constructor voor MainWindow
- Onze volledige code ziet er nu zo uit:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void BtnToonLand_Click(object sender, RoutedEventArgs e)
    {
        ToonLand();
    }
    void ToonLand(string land)
    {
        string land = "België";
        MessageBox.Show("Het land is " + land);
    }
}
```

### 3 PARAMETERS

In de methode ToonLand kunnen we maar één land laten zien via de messagebox. Om ook andere landen te kunnen tonen, declareren we de variabele niet in de methode, maar maken er een parameter van.

```
void ToonLand(string land)
{
    MessageBox.Show("Het land is " + land);
}
```

Een call (oproep) naar deze methode ziet er dan als volgt uit:

```
string teTonenLand = "Frankrijk";
ToonLand(teTonenLand);
```

Onze volledige code ziet er nu zo uit:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void BtnToonLand_Click(object sender, RoutedEventArgs e)
    {
        string teTonenLand = "Frankrijk";
        ToonLand(teTonenLand);
    }
    void ToonLand(string land)
    {
        MessageBox.Show("Het land is " + land);
    }
}
```



## 4 RETURNS

Hierboven zagen we methodes die niets retourneerden. De code in de methode werd uitgevoerd, maar er werd geen resultaat van de bewerkingen teruggegeven aan de statement die de methode aangeroepen had.

- Maak een nieuw project aan in de bestaande solution Methoden en noem dit nieuw project **WpfBerekenSom**.
- Voeg in de **code-behind** een nieuwe methode BerekenSom toe:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
        return getalLinks + getalRechts;
    }
}
```

- Deze methode zullen we nu even ontleden:
- De methode heeft als naam BerekenSom:

```
int BerekenSom(int getalLinks, int getalRechts)
```

- BerekenSom retourneert een geheel getal naar de plaats van aanroep:

```
int BerekenSom(int getalLinks, int getalRechts)
```

- BerekenSom ontvangt twee gehele getallen als parameters:

```
int BerekenSom(int getalLinks, int getalRechts)
```

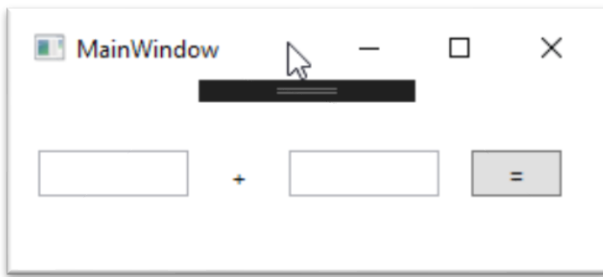
- Binnen de methode worden twee gehele getallen opgeteld

```
return getalLinks + getalRechts;
```

- BerekenSom retourneert het resultaat van de optelling naar de plaats van aanroep

```
return getalLinks + getalRechts;
```

- We zullen deze methode nu gebruiken in onze applicatie:



- Voeg twee textboxes toe
  - De linker textbox geef je de naam "txtGetalLinks" en je verwijdert de content
  - De rechter textbox geef je de naam "txtGetalRechts" en je verwijdert de content
- Voeg een label toe met content "+" en plaats het tussen de twee textboxes
- Voeg een button toe, geef deze de naam "btnBerekenSom" en plaats deze na de tweede textbox
- Dubbelklik nu op de button zodat er automatisch een eventhandlermethode aangemaakt wordt:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
        return getalLinks + getalRechts;
    }
    private void BtnBerekenSom_Click(object sender, RoutedEventArgs e)
    {
    }
}
```

- Voorzie de methode-statements:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
        return getalLinks + getalRechts;
    }
    private void BtnBerekenSom_Click(object sender, RoutedEventArgs e)
    {
        int links = int.Parse(txtGetalLinks.Text);
        int rechts = int.Parse(txtGetalRechts.Text);
        int som = BerekenSom(links, rechts);
        MessageBox.Show("De som van " + links + " en " + rechts + " is " + som,
"Som berekenen");
    }
}
```

```
}
}
```

- Deze statements doen het volgende:

We lezen de waarde van de "txtGetalLinks" uit. Omdat de inhoud van onze textbox altijd van het type tekst (string) is moeten we deze omzetten (parsen) naar een geheel getal (int) vooraleer we wiskundige operaties ermee kunnen uitvoeren. In ons geval een optelling.

Parse de inhoud van de "txtGetalLinks" naar het data type int en plaats deze in een nieuwe variabele "int links":

```
int links = int.Parse(txtGetalLinks.Text);
```

- Doe hetzelfde met "txtGetalRechts", parse de inhoud van "txtGetalRechts" naar het data type int en plaats deze in een nieuwe variabele "int rechts":

```
int rechts = int.Parse(txtGetalRechts.Text);
```

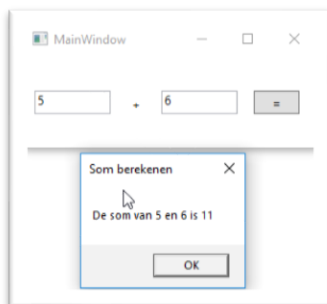
- We declareren een nieuwe variabele van data type int met de naam "som". Deze variabele ontvangt de geretourneerde waarde van onze methode "BerekenSom" die variabelen "links" en "rechts" meekregen als parameters. Met andere woorden; we geven het linker- en het rechtergetal mee aan onze methode BerekenSom. In deze methode worden de getallen "links" en "rechts" opgeteld en het resultaat wordt geretourneerd naar deze plaats van aanroep en wordt in de nieuwe variabele "int som" gestopt:

```
int som = BerekenSom(links, rechts);
```

- Via een MessageBox tonen we deze optelling met onze drie variabelen "links", "rechts" en "som":

```
MessageBox.Show("De som van " + links + " en " + rechts + " is " + som, "Som berekenen");
```

- Druk op F5 op je applicatie te starten en test je applicatie:



## 4.1 PARAMETER ARRAYS

In het bovenstaande hoofdstuk wisten we op voorhand hoeveel parameters/argumenten we moesten doorgeven bij de call naar de method. Dit is niet altijd het geval. Parameter arrays kan je handig aanwenden wanneer je niet op voorhand weet hoeveel argumenten je bij een methode-aanroep zal meegeven.

Met het sleutelwoord `params` kan je in een methode waarden in een array ontvangen. Bij de call van deze methode kunnen de waarden één voor één worden opgegeven.

## 4.2 TOEPASSING PARAMETER ARRAYS

- Maak een nieuw project aan met de naam `ParamArray.Wpf`
- Voorzie volgende lay-out:



Naam van de controls:

- `btnToonBerichten`
- `lblAantal`
- `lstBerichten`

- Voorzie volgende methode in `MainWindow.xaml.cs`:

```
private void ToonBerichten(params string[] berichten)
{
    lstBerichten.Items.Clear();
    lblAantal.Content = "Aantal berichten: " + berichten.Length;
    foreach (string bericht in berichten)
    {
        lstBerichten.Items.Add(bericht);
    }
}
```

- Voorzie een Button `btnToonBerichten` met volgende click-handler:

```
private void BtnToonBerichten_Click(object sender, RoutedEventArgs e)
{
    ToonBerichten(
        "C# is leuk",
        "Soep is lekker",
        "Sneeuw is koud",
        "Binair begint met de letter b");
}
```

Belangrijk bij het gebruik van params is dat enkel het laatste argument binnen een methode het sleutelwoord **params** kan bevatten.

Binnen vele .Net-methoden wordt van dit principe gebruik gemaakt: String.Format, Console.WriteLine, ...

## 5 SCOPE

Binnen de methode `berekenSom_Click` werden drie variabelen `links`, `rechts` en `som` gedeclareerd.

Variabelen die gedeclareerd werden binnen een methode hebben deze methode als bereik of scope.

Dit wil zeggen dat deze variabelen buiten deze methode totaal ongekend zijn en dus onbruikbaar.

- Voorzie een nieuwe methode `ScopeTest`:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
        return getalLinks + getalRechts;
    }
    private void BtnBerekenSom_Click(object sender, RoutedEventArgs e)
    {
        int links = int.Parse(txtGetalLinks.Text);
        int rechts = int.Parse(txtGetalRechts.Text);
        int som = BerekenSom(links, rechts);
        MessageBox.Show("De som van " + links + " en " + rechts + " is " + som,
"Som berekenen");
    }
    void ScopeTest()
    {
        MessageBox.Show("De som was: " + som);
    }
}
```

Wanneer je de toepassing wil uitvoeren krijg je een compileerfout:

The name 'som' does not exist in the current context

“som” is dus ongekend binnen de methode “ScopeTest”.

Je kan “som” natuurlijk opnieuw declareren en de berekening opnieuw doen, maar dit kan natuurlijk niet de bedoeling zijn.

Wanneer je de variabele “som” wenst te gebruiken buiten de methode “`berekenSom_Click`” kan je deze declareren op klasseniveau: buiten elke methode, maar binnen de accolades van de klasse.

- Pas de toepassing als volgt aan:

```
public partial class MainWindow : Window
{
    int som;

    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
```

```

        return getalLinks + getalRechts;
    }
    private void BtnBerekenSom_Click(object sender, RoutedEventArgs e)
    {
        int links = int.Parse(txtGetalLinks.Text);
        int rechts = int.Parse(txtGetalRechts.Text);
        som = BerekenSom(links, rechts);
        MessageBox.Show("De som van " + links + " en " + rechts + " is " + som,
"Som berekenen");
    }
    void ScopeText()
    {
        MessageBox.Show("De som was: " + som);
    }
}

```

- De variabele “som” is nu gekend in de ganse klasse MainWindow

## 6 OVERLOADING

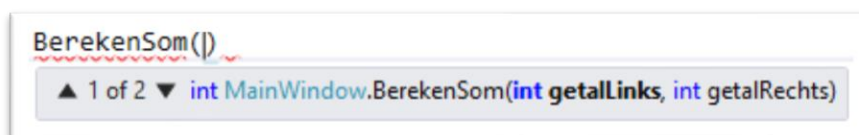
Als twee identifiers (variabelen, methoden, ...) binnen dezelfde scope, dezelfde naam hebben zeggen we dat ze **overloaded** zijn. Dikwijls is overloading niet de bedoeling (zoals twee keer dezelfde variabele declareren binnen dezelfde methode) en krijg je een compile-time error (een fout tijdens het compileren). Soms is overloading echter extreem nuttig, dit fenomeen zien we vaak bij het gebruik van methoden.

We willen de tekstwaarden uit de tekstvakken onmiddellijk kunnen doorgeven aan "BerekenSom". De omzetting in een geheel getal moet nu dus gebeuren in de methode "BerekenSom". Vanzelfsprekend willen we de methode "BerekenSom" die twee gehele getallen ontvangt niet verliezen, we maken dus een bijkomende methode "BerekenSom":

```
public partial class MainWindow : Window
{
    int som;

    public MainWindow()
    {
        InitializeComponent();
    }
    int BerekenSom(int getalLinks, int getalRechts)
    {
        return getalLinks + getalRechts;
    }
    int BerekenSom(string getalLinks, string getalRechts)
    {
        int links = int.Parse(getalLinks);
        int rechts = int.Parse(getalRechts);
        return BerekenSom(links, rechts);
    }
    private void BtnBerekenSom_Click(object sender, RoutedEventArgs e)
    {
        string links = txtGetalLinks.Text;
        string rechts = txtGetalRechts.Text;
        som = BerekenSom(links, rechts);
        MessageBox.Show("De som van " + links + " en " + rechts + " is " + som,
"Som berekenen");
    }
    void ScopeText()
    {
        MessageBox.Show("De som was: " + som);
    }
}
```

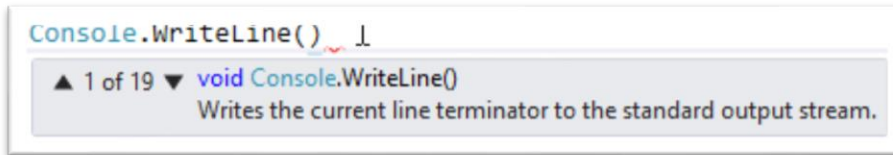
Let op de Visual Studio Intellisense bij het aanroepen van de methode BerekenSom:







Bekijk eens het aantal overloads voor `Console.WriteLine()`:



## 7 EVENT HANDLER METHODEN ↔ EIGEN METHODEN

### 7.1 EVENT HANDLER METHODS

Deze methodes worden uitgevoerd op het moment dat er een gebeurtenis plaatsvindt op een control, bvb. een klik op een button, een item dat aangeklikt wordt in een listbox ...

In een event handler method mogen er in principe enkel de volgende zaken voorkomen:

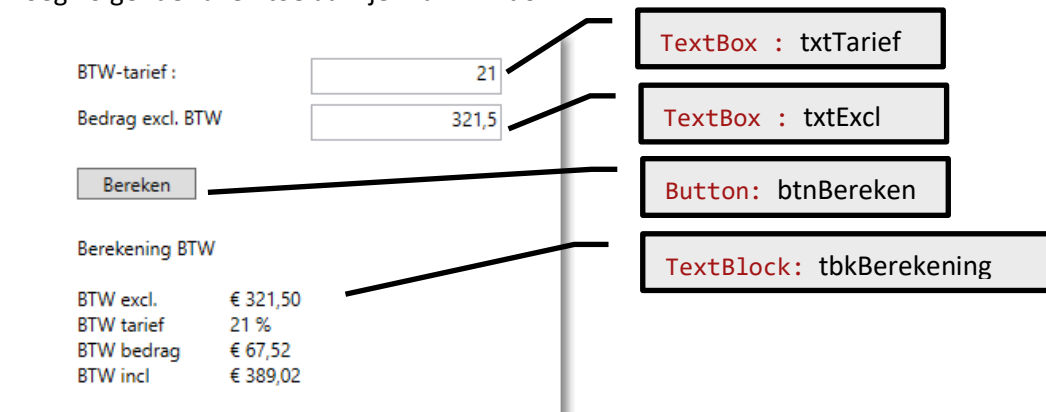
- Declaratie van lokale variabelen
- Inlezen van de input van de gebruiker
- Toewijzen van waarden aan de variabelen (indien eenvoudig statement)
- Call(s) naar eigen methoden
- Feedback naar de gebruiker
- Eventueel code om de applicatie gebruiksvriendelijk te maken (focus, isEnabled, Visibility...).

Probeer bovenstaande volgorde ook aan te houden in je andere methods. Op die manier is het later makkelijker om er zaken in op te zoeken/aan te passen.

Event handler methods moeten altijd simpel blijven om te lezen. De echte intelligentie ligt bij de methoden en later bij de klassen (zie hoofdstuk over objecten en klassen).

Maak binnen je solution een nieuw project aan met de naam **BerekenTotaal.Wpf**.

Voeg volgende zaken toe aan je MainWindow :



```
private void BtnBereken_Click(object sender, RoutedEventArgs e)
{
    decimal prijsExcl;
    decimal prijsIncl;
    float tarief;
    decimal btwBedrag;
    string samenvatting;

    tarief = float.Parse(txtTarief.Text);
    prijsExcl = decimal.Parse(txtExcl.Text);

    btwBedrag = prijsExcl * (decimal)tarief / 100;
    prijsIncl = prijsExcl + btwBedrag;

    samenvatting = "Berekening BTW\n\n" +
        $"BTW excl.\t€ {prijsExcl.ToString("0.00")}\n" +
```

```

        $"BTW tarief\t{tarief} %\n" +
        $"BTW bedrag\t€ {btwBedrag.ToString("0.00")}\n" +
        $"BTW incl\t\t€ {prijsIncl.ToString("0.00")}\n";

        tbkBerekening.Text = samenvatting;
    }

```

In bovenstaand voorbeeld is het geen goed idee om **berekeningen** binnen de event handler te houden.

We kunnen hier beter een methode schrijven om het btwbedrag en de prijs incl. BTW te berekenen. Deze methoden retourneren dan de berekende bedragen op basis van de meegegeven parameters.

Op die manier houden we de code binnen de event handler method simpeler. We kunnen de geschreven methodes ook opnieuw gebruiken vanuit andere event handlers en eventueel ook andere projecten.

```

private void BtnBereken_Click(object sender, RoutedEventArgs e)
{
    decimal prijsExcl;
    decimal prijsIncl;
    float tarief;
    decimal btwBedrag;
    string samenvatting;

    tarief = float.Parse(txtTarief.Text);
    prijsExcl = decimal.Parse(txtExcl.Text);

    btwBedrag = BtwBedrag(tarief, prijsExcl);
    prijsIncl = BtwInclusief(tarief, prijsExcl);

    samenvatting = "Berekening BTW\n\n" +
        $"BTW excl.\t€ {prijsExcl.ToString("0.00")}\n" +
        $"BTW tarief\t{tarief} %\n" +
        $"BTW bedrag\t€ {btwBedrag.ToString("0.00")}\n" +
        $"BTW incl\t\t€ {prijsIncl.ToString("0.00")}\n";

    tbkBerekening.Text = samenvatting;
}

decimal BtwBedrag(float tarief, decimal prijsExcl)
{
    decimal btwBedrag;
    btwBedrag = prijsExcl * (decimal)tarief / 100;
    return btwBedrag;
}

decimal BtwInclusief(float tarief, decimal prijsExcl)
{
    decimal prijsIncl;
    decimal btwBedrag;
    btwBedrag = BtwBedrag(tarief, prijsExcl);
    prijsIncl = prijsExcl + btwBedrag;
}

```

```

        return prijsIncl;
    }

```

Ook het samenstellen van feedback aan de gebruiker op basis van verscheidene stukken informatie, kan beter in een methode ondergebracht worden.

```

private void BtnBereken_Click(object sender, RoutedEventArgs e)
{
    float tarief;
    decimal prijsExcl;
    tarief = float.Parse(txtTarief.Text);
    prijsExcl = decimal.Parse(txtExcl.Text);
    tbkBerekening.Text = ToonBtwBerekening(tarief, prijsExcl);
}
decimal BtwBedrag(float tarief, decimal prijsExcl)
{
    decimal btwBedrag;
    btwBedrag = prijsExcl * (decimal)tarief / 100;
    return btwBedrag;
}
string ToonBtwBerekening(float tarief, decimal prijsExcl)
{
    string samenvatting = "";
    decimal btwBedrag;
    decimal prijsIncl;
    btwBedrag = BtwBedrag(tarief, prijsExcl);
    prijsIncl = prijsExcl + btwBedrag;
    samenvatting = "Berekening BTW\n\n" +
        $"BTW excl.\t€ {prijsExcl.ToString("0.00")}\n" +
        $"BTW tarief\t{tarief} %\n" +
        $"BTW bedrag\t€ {btwBedrag.ToString("0.00")}\n" +
        $"BTW incl\t€ {prijsIncl.ToString("0.00")}\n";
    return samenvatting;
}

```

## 7.2 CHECK JE EIGEN METHODE

Stel dat je de volgende code hebt geschreven om het controlegetal van een rijksregisternummer te checken:

```

private void BtnCheckControleGetal_Click(object sender, RoutedEventArgs e)
{
    CheckControleGetal();
}

void CheckControleGetal()
{
    string rijksNummer;
    int rijksRegister9chars;
    int teCheckenControlGetal;
    int controleGetal;
}

```

```

bool controleGetalKlopt;

rijksNummer = txtRijksregisterNummer.Text;

rijksRegister9chars = int.Parse(rijksNummer.Substring(0, 9));
teCheckenControlGetal = int.Parse(rijksNummer.Substring(9, 2));
controleGetal = 97 - (rijksRegister9chars % 97);

controleGetalKlopt = teCheckenControlGetal == controleGetal;

chkCheckControleGetal.IsChecked = controleGetalKlopt;
}

```

Op het eerste gezicht een goede oplossing. De code in de event handler is heel simpel.

Bekijken we de methode CheckControleGetal ivm hergebruik in andere omstandigheden, dan is er wel een probleem.

Er wordt informatie opgehaald uit `txtRijksregisterNummer`. In een andere omgeving is het mogelijk dat deze control niet bestaat. In dergelijke gevallen geven we de info die we ophalen mee als **parameter**. We moeten de info wel ophalen in de event handler methode die deze methode callt.

```

private void BtnCheckControleGetal_Click(object sender, RoutedEventArgs e)
{
    string rijksNummer;
    rijksNummer = txtRijksregisterNummer.Text;
    CheckControleGetal(rijksNummer);
}

void CheckControleGetal(string rijksNummer)
{
    ...

    controleGetalKlopt = teCheckenControlGetal == controleGetal;

    chkCheckControleGetal.IsChecked = controleGetalKlopt;
}

```

Een tweede probleem is dat we een property van een specifiek control willen aanpassen. Ook hier zijn we niet zeker of die bij een volgend gebruik aanwezig zal zijn.

De oplossing hier is de waarde van de aan te passen property te **retourneren**. Het return type wordt dan het datatype van het resultaat dat geretourneerd moet worden. Ook hier moet de call aangepast worden.

```

private void BtnCheckControleGetal_Click(object sender, RoutedEventArgs e)
{
    string rijksNummer;
    rijksNummer = txtRijksregisterNummer.Text;
    chkCheckControleGetal.IsChecked = CheckControleGetal(rijksNummer);
}

```

```
bool CheckControleGetal(string rijksNummer)
{
    ...
    controleGetalKlopt = teCheckenControlGetal == controleGetal;


    return controleGetalKlopt;
}
```

Ander voordeel is dat de gebruiker van de methode de geretourneerde info kan gebruiken zoals hij wil. Een control kan aangepast worden, de waarde kan verder gebruikt worden in het programma, een messagebox kan getoond worden...



### Code repository

De volledige broncode van dit hoofdstuk is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-methoden-code-uit-voorbeelden.git>

# HOOFDSTUK 4

# DEBUGGING





## INHOUDSOPGAVE

<b>1</b>	<b>JE APPLICATIE DEBUGGEN</b>	<b>91</b>
<b>1.1</b>	<b>Breakpoints</b>	<b>91</b>
1.1.1	Breakpoints plaatsen	91
1.1.2	Breakpoints verfijnen	92
<b>1.2</b>	<b>Step Into, Step Over en Step Out</b>	<b>93</b>
1.2.1	Variabelen inspecteren	94



# 1 JE APPLICATIE DEBUGGEN

Tijdens het ontwikkelen van programma's zal je tal van problemen moeten oplossen. Eerst en vooral moet je correcte code leren schrijven zodat je geen **compile-time errors** veroorzaakt. Anderzijds moet je **run-time fouten** kunnen oplossen. Het maken van een try/catch blok (zie hfst 8) is niet altijd de oplossing.

Stel dat je een try/catch blok schrijft waarin een netwerkverbinding wordt geopend die een webpagina download. Het wil maar niet lukken; je krijgt steeds een `NullReferenceException`. Zo'n run-time fout wordt meestal veroorzaakt door jezelf en moet door jou worden opgelost zonder nood aan een try/catch blok.

In zo'n geval moet je gaan debuggen.

Statistieken wijzen uit dat een programmeur voor elke 1 000 lijnen code gemiddeld 70 bugs introduceert. Het oplossen van een bug duurt gemiddeld 30 maal langer dan het schrijven van die code. Een programmeur zal gemiddeld meer dan 60% van zijn tijd doorbrengen aan het debuggen van een applicatie. Efficiënt leren debuggen is dus onontbeerlijk!

## 1.1 BREAKPOINTS

Een breakpoint is een door jou bepaalde locatie in de code waar de uitvoering van de applicatie zal pauzeren. Tijdens zo'n *break* kan je heel wat hulpmiddelen inzetten om fouten op te sporen.

### 1.1.1 BREAKPOINTS PLAATSEN

Je plaatst een breakpoint door in de donkere kantlijn van je codevenster te klikken zodat er een bordeauxrode bol verschijnt. De uitvoering zal dan stoppen vlak voor die instructie, die eveneens rood wordt ingekleurd.



#### Code repository

Download de oefening Debugging.Wpf via

`git clone https://github.com/howest-gp-prb/cu-debugging.git`

6. Plaats enkele breakpoints, op de volgende plaatsen:

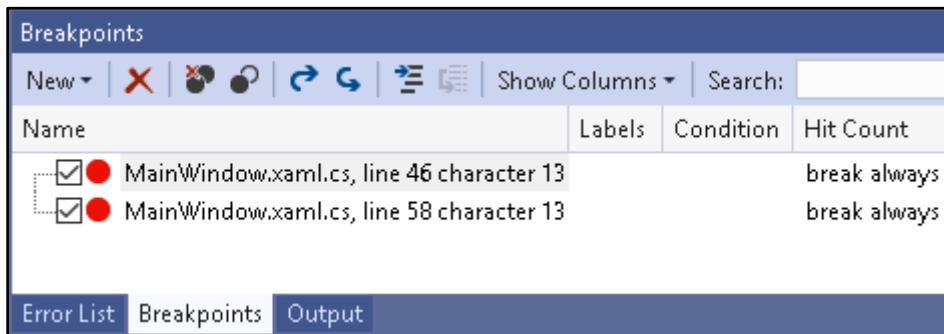
```

44  string ToonBewerkingen(int getal1, int getal2)
45  {
46      string resultaten;
47
48      resultaten = $"{getal1} + {getal2} = {TelOp(getal1, getal2).ToString("0.00")}\n";
49      resultaten += $"{getal1} / {getal2} = {Deel(getal1, getal2).ToString("0.00")}\n";
50      resultaten += "-----";
51      return resultaten;
52  }
53
54  1 reference
55  private void BtnBewerkingen_Click(object sender, RoutedEventArgs e)
56  {
57      int getal1, getal2;
58      string bewerkingen;
59      getal1 = int.Parse(txtGetal1.Text);
60      getal2 = int.Parse(txtGetal2.Text);
61
62      bewerkingen = ToonBewerkingen(getal1, getal2);
63      tbkResult.Text = bewerkingen;
64  }

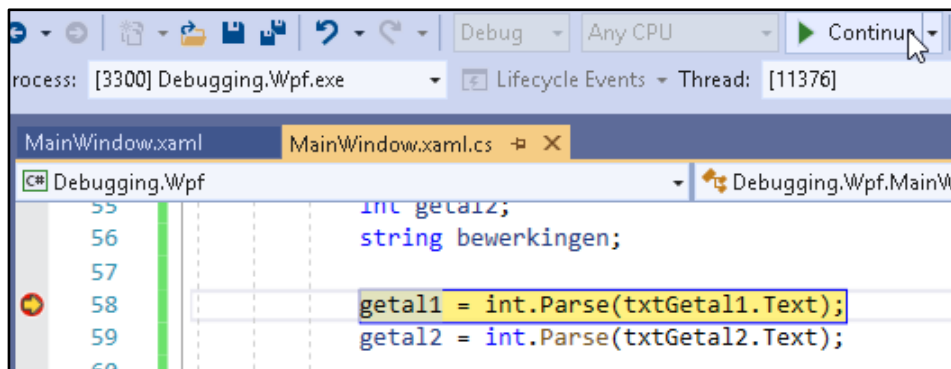
```

Om een breakpoint te verwijderen die je niet langer nodig hebt klik je nogmaals in de kantlijn, op de bol.

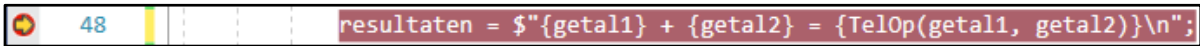
7. Open het Breakpoints venster in Visual Studio, via **Debug → Windows → Breakpoints**.  
Je vindt er alle breakpoints in de broncode.



- Om een breakpoint tijdelijk te **deactiveren** zet je het betreffende vinkje uit.
  - Om een breakpoint te **verwijderen** rechtsklik je erop en kies je voor **Delete**.
8. Voer de applicatie uit in debug mode, via F5, de Start knop of via **Debug → Start Debugging**.  
Wanneer je nu een 'Bewerkingen' klikt, merk je dat je terug naar het codescherm wordt geleid.  
Visual Studio stopt vlak vóór de code van het breakpoint wordt uitgevoerd en geeft de locatie aan met een gele pijl en arcering.



9. Klik op **Continue** of druk op **F5** om de uitvoering verder te zetten.  
De gele cursor springt nu naar het volgende breakpoint dat geraakt wordt.

10. 

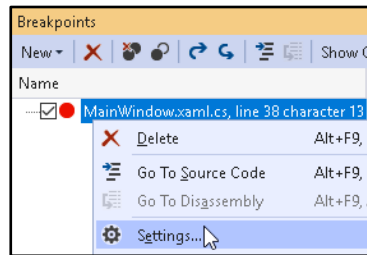
11. Druk nogmaals op **Continue** of **F5** om de uitvoering verder te zetten. Er worden geen breakpoints meer geraakt en de applicatie wordt opnieuw actief en bestuurbaar.

### 1.1.2 BREAKPOINTS VERFIJNEN

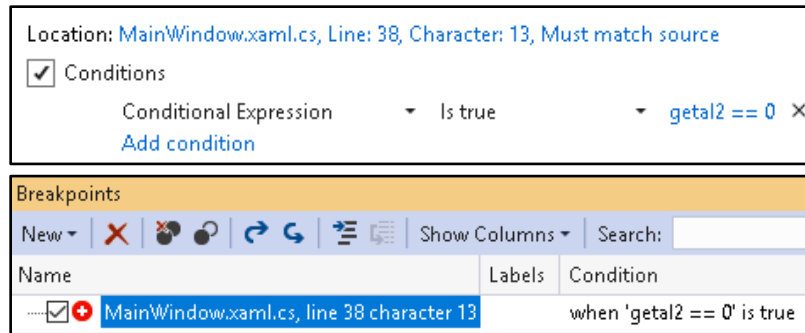
Het kan gebeuren dat de onderbreking van de code enkel wil laten gebeuren in bepaalde gevallen. Stel dat we willen volgen wat er gebeurt als getal 2 gelijk is aan 0 bij het volgende breakpoint:

```
float Deel(int getal1, int getal2)
{
    float quotient;
    quotient = getal1 / (float)getal2;
    return quotient;
}
```

In het snelmenu van het Breakpoints window kunnen we kiezen voor 'Settings':



Daarna kun je een voorwaarde ingeven om de code te onderbreken.



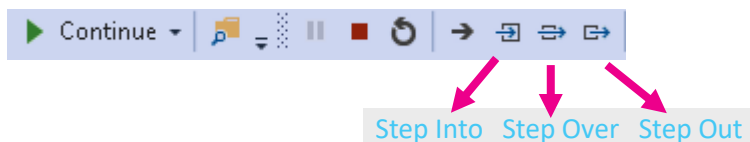
De code wordt nu inderdaad enkel onderbroken als getal2 gelijk is aan 0.

## 1.2 STEP INTO, STEP OVER EN STEP OUT

Tijdens de break van een applicatie kan je bepalen waarheen de gele cursor moet springen. Er zijn verschillende opties:

- Doorgaan tot aan het volgende breakpoint: **Continue** of **F5**
- De huidige instructie uitvoeren en wachten: **Step Into**, **Step Over** en **Step Out**.

Je vindt deze opties bovenaan het hoofdvenster van Visual Studio in debug modus:



12. Plaats een breakpoint in lijn 46

13.

14. Klik op de knop en wacht tot breakpoint geraakt wordt.

15. Druk op **Step Over** of **F10**. Dit is de meeste gebruikte optie.

De gele cursor voert de huidige instructie uit en pauzeert op de instructie eronder.

Druk op F5 om de code verder te laten uitvoeren.

16. Klik nogmaals op de knop en wacht tot breakpoint geraakt wordt.

17. Druk op **Step Into** of **F11**.

De gele cursor sprint naar de methode '*Optelling*' en pauzeert op de eerste accolade.

Met F10 of F11 kun je nu de code binnen deze methode *Optelling* stap voor stap laten uitvoeren.

18. Klik nu op de **Step Out** knop of **Shift + F11**. De cursor springt terug naar het statement dat de call naar de methode heeft uitgevoerd.

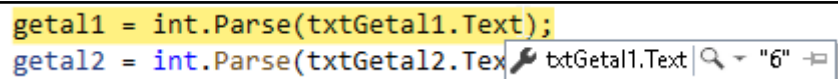
### 1.2.1 VARIABELEN INSPECTEREN

Tijdens het navigeren van een gepauzeerde applicatie kan je de waarden van variabelen inspecteren. Dit is een bijzonder krachtig hulpmiddel voor het opsporen van fouten. Er zijn twee manieren om dit te doen; door er met de muiscursor over te zweven, of met hulpvenster.

19. Voer een som uit met een niet-numerieke waarde en wacht tot het eerste breakpoint geraakt wordt.

#### MET DE MUISCURSOR

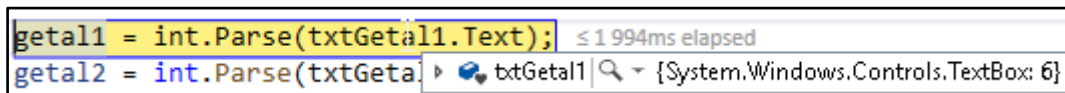
20. Zweef met de muiscursor over de Text property van de Textbox instanties.  
Je merkt telkens een tooltip die de waarde van de variabele weergeeft.



```
getal1 = int.Parse(txtGeta1.Text);
getal2 = int.Parse(txtGeta2.Text);
```

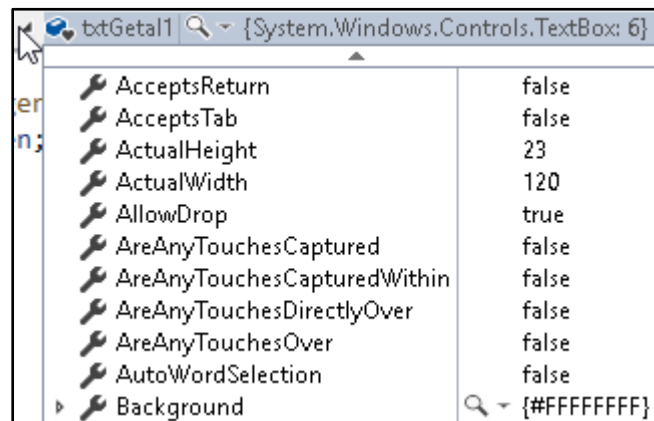
Bij een variabele die een complex datatype heeft wordt de volledige naam van het type getoond. Dat is het geval voor bijvoorbeeld de variabele txtGeta1, dat van het type TextBox is.

21. Zweef met de muiscursor over de variabele txtGeta1 en bekijk de tooltip.



```
getal1 = int.Parse(txtGeta1.Text);
getal2 = int.Parse(txtGeta2.Text);
```

22. Bekijk de Properties van deze variabele door het pijltje in de tooltip open te klappen.

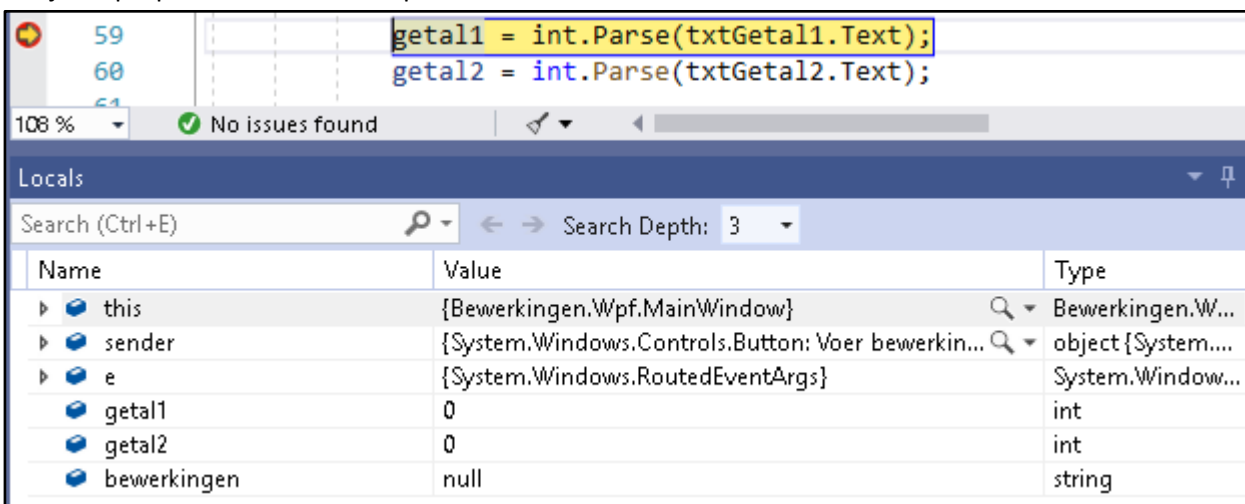


Property	Value
AcceptsReturn	false
AcceptsTab	false
ActualHeight	23
ActualWidth	120
AllowDrop	true
AreAnyTouchesCaptured	false
AreAnyTouchesCapturedWithin	false
AreAnyTouchesDirectlyOver	false
AreAnyTouchesOver	false
AutoWordSelection	false
Background	{#FFFFFF}

## HET LOCALS VENSTER

In het Locals venster worden alle variabelen in de huidige scope (van de methode) getoond. Je kan het venster zichtbaar maken via **Debug → Windows → Locals**.

23. Maak een som met correcte numerieke waarden en gebruik **Step Over** tot je op de regel terecht komt waar de Text property van de TextBox wordt opgevuld met de waarde uit de variabele som.
24. Open het **Locals** venster en bekijk de waarden.  
Je vindt er drie kolommen: de naam van de variabele, de waarde ervan en het type. De meest recent aangepaste variabele heeft een rode kleur gekregen. Complexe typen zijn ook hier uitklapbaar zodat je de properties ervan kan inspecteren.



## DE WATCH VENSTERS

Er zijn meerdere **Watch** vensters beschikbaar waarin je zelf variabelen kan plaatsen om die in het oog te houden. Dit zijn veruit de meest gebruikte Debug vensters die er zijn. Er zijn 4 watch vensters, respectievelijk Watch 1, Watch 2, Watch 3 en Watch 4 genaamd. Tijdens het debuggen zou er al eentje zichtbaar moet zijn in.

Om een Watch venster te openen, ga je naar **Debug → Windows → Watch → Watch 1 ... 4**

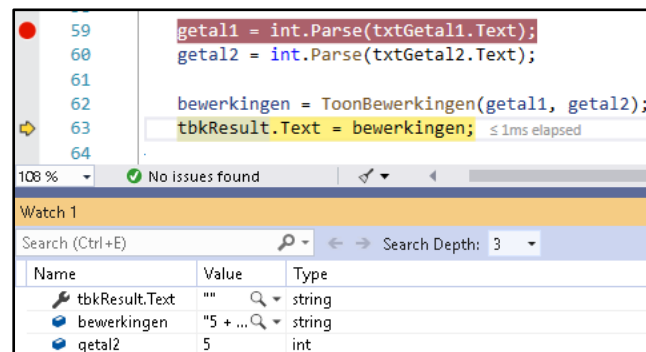
25. Maak een som met correcte numerieke waarden en wacht tot de eerste breakpoint geraakt wordt.
26. Open het **Watch 1** venster. Deze is standaard leeg.

Je kan een te inspecteren variabele toevoegen op twee manieren:

- Tik de naam van de variabele in de Name kolom en druk op enter.
  - Rechtsklik op de variabele in het codevenster en kies **Add Watch**.
27. Door tijdens het debuggen met de rechtmuisknop op een variabele of object te klikken, kun je in het snelmenu kiezen voor Add Watch. Je kunt ook gewoon in de Name-kolom tikken wat je wil 'wachten'.
  28. Voeg de volgende variabelen toe aan het Watch 1 venster:
    - tbkResult.Text
    - bewerkingen
    - getal2

Name	Value	Type
tbkResult.Text	""	string
bewerkingen	null	string
getal2	0	int

29. Gebruik **Step Over** tot je op de regel terechtkomt waar de Text property van de TextBox wordt opgevuld met de waarde uit de variabele som. Merk op hoe de waarden wijzigen tijdens de stapsgewijze uitvoering van je code. Hier kan je zeer veel uit leren om de oorzaak van fouten te achterhalen.



Je kunt de waarden in de kolom 'Value' ook overtikken. Dit kan interessant zijn om de effecten na te gaan bij een andere waarde dan de actuele.

Debuggen is een taak die evenwaardig is aan het schrijven van correcte code. Maak er een goede gewoonte van tijdens de ontwikkeling van je applicatie zodat je efficiënt met de hulpmiddelen leert werken. Op deze manier moet je niet gissen wat de oorzaak van een fout is, maar kan je dit zonder meer achterhalen.



### Meer informatie

[MS: Visual Studio - Getting started with the debugger](#)



# HOOFDSTUK 5

## SELECTIE



## INHOUDSOPGAVE

<b>1</b>	<b>BOOLEAANSE VARIABELEN</b>	<b>101</b>
<b>2</b>	<b>BOOLEAANSE OPERATOREN</b>	<b>102</b>
<b>3</b>	<b>VOORWAARDELIJKE OPERATOREN</b>	<b>103</b>
<b>4</b>	<b>VOLGORDE VAN BEWERKINGEN</b>	<b>104</b>
<b>5</b>	<b>SELECTIE: IF</b>	<b>105</b>
<b>5.1</b>	<b>if: als de bewering klopt...</b>	<b>105</b>
<b>5.2</b>	<b>... en als de bewering niet klopt: else</b>	<b>105</b>
<b>5.3</b>	<b>Statements groeperen: codeblokken</b>	<b>105</b>
<b>5.4</b>	<b>if statements nesten: else if</b>	<b>107</b>
<b>6</b>	<b>SWITCH</b>	<b>110</b>



## 1 BOOLEAANSE VARIABLEN

In de wereld van het programmeren wordt gewerkt met bits. Een bit heeft de waarde 1 of 0. Dit zijn de twee toestanden van een bit.

In het programmeren moet een expressie altijd als waar of onwaar geëvalueerd worden: 1 of 0. In het dagelijkse leven kan een antwoord op een vraag vaak nogal vaag zijn: misschien, waarschijnlijk, ...

Wanneer we programmeren, moet het antwoord steeds exact gekend zijn.

We wijzen een variabele *x* de waarde 99 toe. Laten we even kijken hoe we dan statements in verband met *x* kunnen evalueren:

- *x* bevat de waarde 99  $\Rightarrow$  evalueert als waar (true)
- *x* is kleiner dan 10  $\Rightarrow$  evalueert als onwaar (false)

C# gebruikt een type *bool* dat een van de twee statussen waar of onwaar kan bevatten. De naam *bool* is afkomstig van de Booleaanse algebra en is een tak uit de exacte wiskunde.

- Maak een nieuwe Visual Studio solution **Selectie**.
- Maak in de solution *Selectie* een nieuwe WPF Application **Selectie.Wpf**.
- Voorzie in *MainWindow* een Button **btnTestBool**
- Wanneer op de knop wordt geklikt wordt volgende code uitgevoerd:

```
private void BtnTestBool_Click(object sender, RoutedEventArgs e)
{
    bool benjeklaar = false;
    MessageBox.Show(benjeklaar.ToString(), "Ben je klaar?");
}
```



## 2 BOOLEAANSE OPERATOREN

Een Booleaanse operator voert een berekening uit die resulteert in waar of onwaar, true / false. De eenvoudigste operator is ! (het uitroepteken): dit is de operator **NOT** - true wordt false en false wordt true.

- We voegen een knop toe aan de applicatie met de naam **btnTestNot** als opschrift 'Test NOT'.

```
private void BtnTestNot_Click(object sender, RoutedEventArgs e)
{
    bool benjeklaar = false;
    benjeklaar = !benjeklaar;
    MessageBox.Show(benjeklaar.ToString(), "Ben je klaar?");
}
```

Het antwoord is nu *true*.

In c# heb je verschillende operatoren om na te gaan of zaken al dan niet gelijk zijn en kleiner/groter.

Operator	Betekenis	Voorbeeld	Uitkomst als x = 99
==	Gelijk aan	x == 100	false
!=	Niet gelijk aan	x != 100	true
<	Kleiner dan	x < 21	false
>	Groter dan	x > 21	true
<=	Kleiner dan of gelijk	x <= 21	false
>=	Groter dan of gelijk	x >= 21	true

### 3 VOORWAARDELIJKE OPERATOREN

C# kent ook twee operatoren waarmee voorwaarden aan elkaar gekoppeld kunnen worden:

- De logische EN (AND) voorgesteld door **&&**
- De logische OF (OR) voorgesteld door **||**

Voorbeelden:

```
void IsAvond()
{
    int uur = 19;
    bool avond = (uur >= 18) && (uur <= 22);
    MessageBox.Show("Om " + uur + " uur: " + avond.ToString(), "Is het al avond?");
}
```

De variabele *avond* bevat *true* wanneer uur groter of gelijk is aan 18 **EN** kleiner of gelijk is aan 22.

```
void IsNacht()
{
    int uur = 23;
    bool nacht = (uur > 22) || (uur < 6);
    MessageBox.Show("Om " + uur + " uur: " + nacht.ToString(), "Is het nacht?");
}
```

De variabele *nacht* bevat *true* wanneer uur groter is aan 22 **OF** kleiner dan 6.

## 4 VOLGORDE VAN BEWERKINGEN

Volgende tabel geeft aan welke prioriteit een bewerking heeft ten opzichte van een andere. Hoe hoger de sectie in de tabel, hoe hoger de prioriteit. Een bewerking met een **hogere prioriteit** wordt in een opdracht uitgevoerd vóór een bewerking met lagere prioriteit. Bewerkingen met een gelijke prioriteit worden **van links naar rechts** uitgevoerd.

operator	beschrijving
()	prioriteit aanpassen
++	verhogen als suffix (vb. i++)
--	verlagen als suffix (vb. i--)
!	NOT
++	verhogen als prefix (vb ++i)
--	verlagen als prefix (vb. --i)
*	vermenigvuldigen
/	delen
%	modulo: rest bij deling
+	optellen
-	afrekken
<	kleiner dan
>	groter dan
<=	kleiner dan of gelijk aan
>=	groter dan of gelijk aan
==	gelijk aan
!=	niet gelijk aan
&&	en
	of
=	toekenning



## 5 SELECTIE: IF

Een selectie is een moment in een programma waarop een beslissing moet worden genomen over hoe het programma verder zal worden uitgevoerd. Enkel ALS aan een voorwaarde is voldaan, worden bepaalde statements uitgevoerd.

Het sleutelwoord voor het nemen van een beslissing is **if**.

### 5.1 IF: ALS DE BEWERING KLOPT...

De algemene syntax voor het nemen van een beslissing is als volgt:

```
if (booleaanse expressie)
    //statement als booleaanse expressie waar is;
```

Voorbeeld:

```
int leeftijd = 19;
if (leeftijd >= 18 ) MessageBox.Show("Op de leeftijd van " + leeftijd + " ben je
meerderjarig");
if (leeftijd < 18) MessageBox.Show("Op de leeftijd van " + leeftijd + " ben je nog
niet meerderjarig");
```

### 5.2 ... EN ALS DE BEWERING NIET KLOPT: ELSE

Algemene syntax:

```
if (booleaanse expressie)
    //statement als booleaanse expressie waar is;
else
    //statement als booleaanse expressie niet waar is;
```

In het vorige voorbeeld hadden we met **else** kunnen werken:

```
if (leeftijd >= 18 ) MessageBox.Show("Op de leeftijd van " + leeftijd + " ben je
meerderjarig");
else MessageBox.Show("Op de leeftijd van " + leeftijd + " ben je nog niet
meerderjarig");
```

### 5.3 STATEMENTS GROEPEREN: CODEBLOKKEN

Gebruik accolades { ... } om statements te groeperen: meer dan 1 statement na if / else

```
if (booleaanse expressie)
{
    //statements als booleaanse expressie waar is;
```

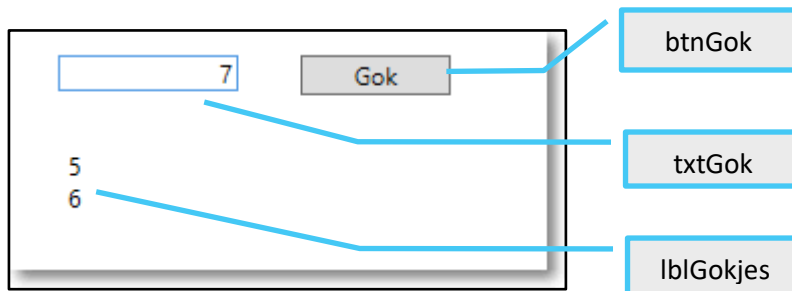
```

}
else
{
    //statements als booleaanse expressie niet waar is;
}

```

Voorbeeld: raadspeel

Maak binnen je solution een nieuw project **Raadspeel.Wpf** aan.



Wanneer het formulier geladen wordt, laten we de computer een getal van 1 – 10 kiezen via de Random-klasse.

*Declareer een random altijd als een instance-variabele (dwz op klasseniveau). Op die manier is de kans veel groter dat je echt willekeurige getallen krijgt.*

```

int teRaden;
int pogingen;
Random rnd = new Random();
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    teRaden = rnd.Next(1, 11);
    Console.WriteLine("Te raden: " + teRaden);
    pogingen = 0;
}

```

Bij een klik op de knop gok wordt het ingegeven getal gelezen en zijn er 2 mogelijkheden:

- het getal is geraden: de speler krijgt feedback over het geraden getal en het aantal pogingen.
- het getal is niet geraden: het geraden getal wordt getoond in een label. De inhoud van txtGok wordt gewist en de focus gaat naar deze textbox.

```

private void BtnGok_Click(object sender, RoutedEventArgs e)
{
    int gok = int.Parse(txtGok.Text);
    if (gok == teRaden)
    {
        pogingen++;
        lblGokjes.Content = "Je hebt het getal " + teRaden + " geraden na " +

```

```

pogingen + " pogingen";
    }
    else
    {
        lblGokjes.Content += gok.ToString() + Environment.NewLine;
        txtGok.Text = "";
        txtGok.Focus();
        pogingen++;
    }
}

```

## 5.4 IF STATEMENTS NESTEN: ELSE IF

Bij een if ... else – selectie ben je als programmeur beperkt tot twee mogelijkheden: ofwel is het if-statement waar, ofwel niet. De realiteit is echter vaak ingewikkelder dan dat.

```

string kleur = "oranje";
string actie;
if (kleur == "rood")
    actie = "stop";
else if (kleur == "groen")
    actie = "doorrijden";
else
    actie = "stop indien mogelijk";

```

Een ander voorbeeld: voeg een button **btnWeekDag** toe aan het MainWindow van WpfSelectie. Het click-event ziet er als volgt uit:

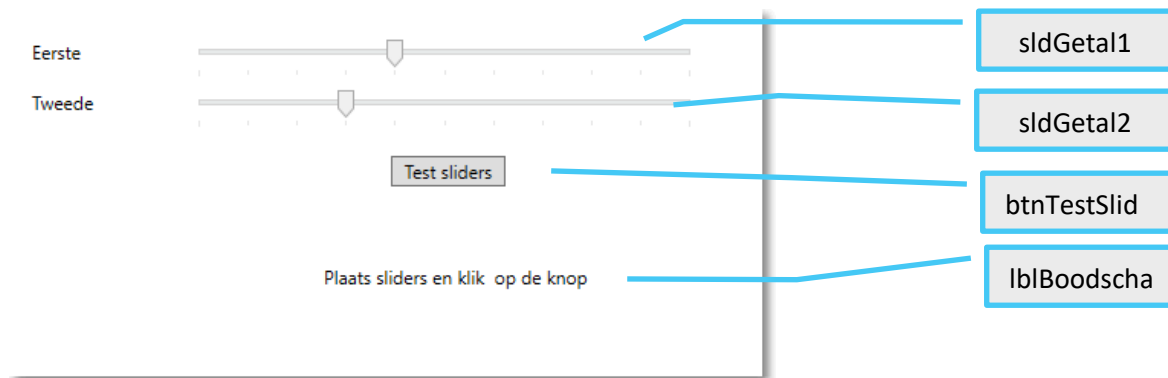
```

string dagNaam;
DateTime momenteel = DateTime.Now;
DayOfWeek dag = momenteel.DayOfWeek;
if (dag == DayOfWeek.Sunday)
    dagNaam = "zondag";
else if (dag == DayOfWeek.Monday)
    dagNaam = "maandag";
else if (dag == DayOfWeek.Tuesday)
    dagNaam = "dinsdag";
else if (dag == DayOfWeek.Wednesday)
    dagNaam = "woensdag";
else if (dag == DayOfWeek.Thursday)
    dagNaam = "donderdag";
else if (dag == DayOfWeek.Friday)
    dagNaam = "vrijdag";
else
    dagNaam = "zaterdag";
MessageBox.Show("Vandaag is het " + dagNaam, "Dag van de week");

```

## Toepassing: waarden vergelijken

Maak binnen je solution een nieuw project aan met de naam **Sliders.Wpf**



De sliders hebben o.a. volgende instellingen:

Property	Value	Omschrijving
AutoToolTipPlacement	BottomRight	Waarde wordt getoond tijdens slepen
TickPlacement	BottomRight	Intervalstreepjes worden getoond
IsSnapToTickEnabled	True	Enkel een waarde op de intervalstreepjes kan geselecteerd worden

Als op de knop wordt geklikt wordt een melding getoond:

- eerste = tweede
- eerste < tweede
- eerste > tweede



### Extra

- verander de tekstkleur van de melding
- gebruik het event ValueChanged van de Sliders om de melding te plaatsen, verwijder de knop.

```
int getal1 = (int)sldGetal1.Value;
int getal2 = (int)sldGetal2.Value;
string boodschap = "";
if (getal1 == getal2) {
    boodschap = "eerste = tweede";
}
else if (getal1 < getal2) {
    boodschap = "eerste < tweede";
}
```

```
else {
    boodschap = "eerste > tweede";
}
lblBoodschap.Content = boodschap;
```

Een if-statement nesten kan ook betekenen dat er een if-statement binnen een ander selectieblok geplaatst wordt.

Dit zien we in het volgende voorbeeld:

```
int leeftijd = 30;
int ticketPrijs;
string beroep = "ambtenaar";

if (beroep == "werkloos" || beroep == "gepensioneerd")
{
    ticketPrijs = 0;
}
else
{
    if (leeftijd < 12)
    {
        ticketPrijs = 0;
    }
    else
    {
        ticketPrijs = 12;
    }
}

MessageBox.Show("De prijs van je ticket bedraagt: " + ticketPrijs + " euro");
```

## 6 SWITCH

In het voorbeeld over de weekdays wordt bij elke else if 'dag ==' herhaald. Om de code in dergelijke gevallen leesbaarder te maken, kun je beter een switch statement gebruiken.

```
switch(controleExpressie)
{
    case constanteExpressie :
        statements;
        break;
    case constanteExpressie :
        statements;
        break;
    ...
    default :
        statements;
        break;
}
```

Opgepast, het sleutelwoord **break** is heel belangrijk als laatste statement van elke **case**. Dit statement beëindigt de switch-constructie. Zonder break wordt de volgende case ook uitgevoerd, ook al is de voorwaarde niet vervuld. In Visual Studio krijg je een foutmelding als de break vergeten wordt. I.p.v. break kan ook **return** of **throw** (zie hoofdstuk over exceptions) gebruikt worden.

Toegepast op ons concreet voorbeeld:

```
string dagNaam;
DateTime momenteel = DateTime.Now;
DayOfWeek dag = momenteel.DayOfWeek;
switch(dag)
{
    case DayOfWeek.Sunday:
        dagNaam = "zondag";
        break;
    case DayOfWeek.Monday:
        dagNaam = "maandag";
        break;
    case DayOfWeek.Tuesday:
        dagNaam = "dinsdag";
        break;
    case DayOfWeek.Wednesday:
        dagNaam = "woensdag";
        break;
    case DayOfWeek.Thursday:
        dagNaam = "donderdag";
        break;
    case DayOfWeek.Friday:
        dagNaam = "vrijdag";
        break;
    default:
        dagNaam = "zaterdag";
}
```

```
        break;  
    }  
    MessageBox.Show("Vandaag is het " + dagNaam, "Dag van de week");
```

**Code repository**

De volledige broncode van dit hoofdstuk is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-selectie.git>





# **HOOFDSTUK 6**

## **ENUMS, ARRAY'S & COLLECTIONS**



## INHOUDSOPGAVE

<b>1</b>	<b>INLEIDING</b>	<b>117</b>
<b>2</b>	<b>ENUMS</b>	<b>118</b>
<b>2.1</b>	<b>Wat zijn enums?</b>	<b>118</b>
<b>2.2</b>	<b>Gebruik van enums</b>	<b>118</b>
<b>2.3</b>	<b>Index van enums</b>	<b>119</b>
<b>2.4</b>	<b>Nut van Enums</b>	<b>121</b>
<b>3</b>	<b>ARRAYS</b>	<b>122</b>
<b>3.1</b>	<b>Wat zijn arrays?</b>	<b>122</b>
<b>3.2</b>	<b>Een Array declareren</b>	<b>122</b>
<b>3.3</b>	<b>Een array instantiëren</b>	<b>123</b>
3.3.1	Declareren, de elementen afzonderlijk instantiëren	123
3.3.2	Verkorte schrijfwijze: direct instantiëren	123
3.3.3	Nog korter	124
<b>3.4</b>	<b>Toepassing</b>	<b>124</b>
<b>3.5</b>	<b>De klasse Array</b>	<b>124</b>
<b>4</b>	<b>COLLECTION KLASSEN</b>	<b>126</b>
<b>4.1</b>	<b>List</b>	<b>126</b>
<b>4.2</b>	<b>Dictionary</b>	<b>127</b>
<b>5</b>	<b>SAMENVATTING</b>	<b>128</b>



## 1 INLEIDING

In programma's werkten we tot hiertoe met variabelen. Die konden wisselende waarden bevatten, maar slecht **één terzelfdertijd**. We konden wel meerdere waarden van dezelfde soort opslaan, maar dan enkel in een control zoals een listbox of een combobox.

Om dergelijke lijsten/reeksen van waarden in het geheugen op te slaan, hebben we in C# meerdere oplossingen. Ze variëren van een eenvoudige, vaststaande lijst met waarden naar complexere objecten.

We kunnen hiervoor gebruik maken van Enums, array's en collections welke we verder in deze syllabus zullen bespreken.

## 2 ENUMS

### 2.1 WAT ZIJN ENUMS?

Enums worden gebruikt om een vaststaande reeks gerelateerde waarden in op te slaan. Je kunt ze bv. gebruiken voor:

- de dagen van de week
- de maanden
- de sterrenbeelden van de horoscoop
- gebruikersstatus
- Operaties van een rekenmachine (optellen, aftrekken, delen en vermenigvuldigen)
- ...

De enige manier om de enum aan te passen, is door de broncode te veranderen.

**Het moet dus echt wel gaan om zaken die nooit zullen veranderen.**

### 2.2 GEBRUIK VAN ENUMS

Aangezien we nog steeds vier seizoenen hebben en er geen seizoenen zullen wegvallen of bijkomen zullen we even de vier seizoenen gebruiken als voorbeeld voor een enum.

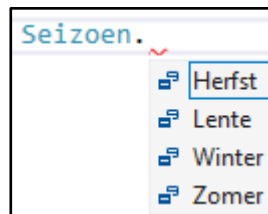
7. Maak een nieuwe WPF applicatie (.NET Framework)
8. Voorzie één button op je MainWindow met volgende properties:

- **Name:** btnToonEnumWaarde
- **Content:** Toon Enum waarde
- **Event:** Click

9. We declareren een enum als volgt:

```
enum Seizoen { Herfst, Winter, Lente, Zomer }
```

10. Om één van de waardes van de enum Seizoen op te halen moeten de naam van de enum typen, gevolgd door een punt. Door de intellisense die aanwezig is in Visual Studio worden alle waardes aanwezig in de enum Seizoen getoond in een lijst:



11. In ons voorbeeld willen we een waarde vanuit de enum Seizoen tonen in een MessageBox. Dit kunnen we doen met onderstaande code. We kiezen vanuit de enum Seizoen de waarde Herfst en zetten dit d.m.v. ToString() om naar een string die getoond kan worden in een MessageBox.

```
private void BtnToonEnumWaarde_Click(object sender, RoutedEventArgs e)
```

```
{
    MessageBox.Show(Seizoen.Herfst.ToString());
}
```

12. Indien we een andere waarde uit de enum `Seizoen` willen tonen moeten we `Herfst` vervangen door een andere waarde van de enum `Seizoen`. Bijvoorbeeld `Zomer`. Onze code ziet er dan zo uit:

```
private void BtnToonEnumWaarde_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(Seizoen.Zomer.ToString());
}
```

## 2.3 INDEX VAN ENUMS

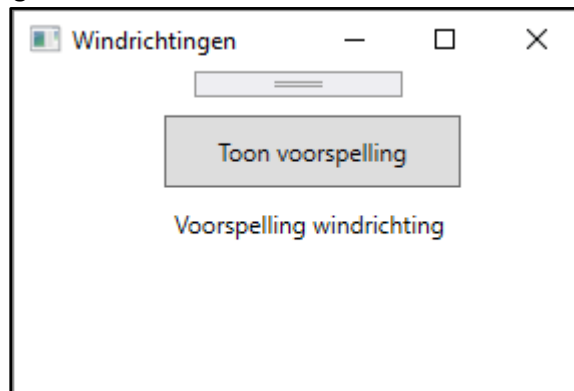
Alle elementen in een enum hebben elk een index. Hierdoor is het mogelijk om eventuele bewerkingen uit te voeren met enums.

We tonen dit even voor aan de hand van een klein voorbeeld met een enum van windrichtingen.

13. We maken een nieuwe WPF project in onze bestaande solution en geven dit de naam **Windrichtingen.WPF**.

14. We voorzien twee controls op ons MainWindow:

- **Button:** btnToonVoorspelling
- **Label:** lblVoorspelling



15. In onze code-behind definiëren we een enum met de windrichtingen:

```
enum WindRichting { Oost, Zuid, Noord, West };
```

16. We creëren een click-event op onze button en voorzien volgende code:

```
private void BtnToonVoorspelling_Click(object sender, RoutedEventArgs e)
{
    int huidigeWindNr = (int)WindRichting.Oost;
```

```

string huidigeWind = WindRichting.Oost.ToString();
lblVoorspelling.Content = "Huidige wind = " + huidigeWind + " (nr. " + huidigeWindNr +
")\n";

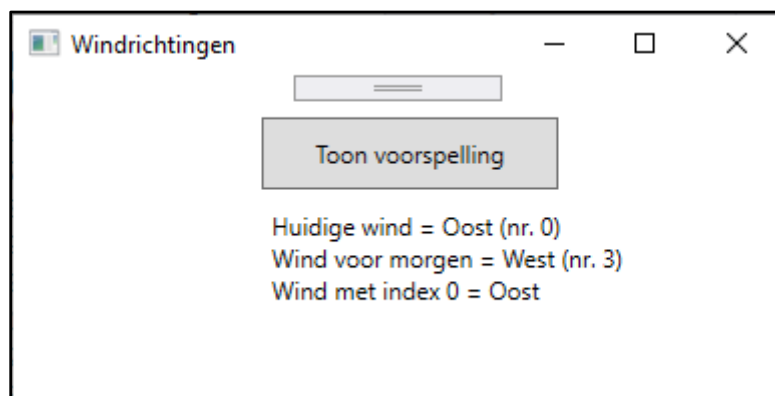
int windVoorMorgenNr = (int)WindRichting.West;
string windVoorMorgen = WindRichting.West.ToString();
lblVoorspelling.Content += "Wind voor morgen = " + windVoorMorgen + " (nr. " +
windVoorMorgenNr + ")\n";

WindRichting richtingNul = (WindRichting)0;
lblVoorspelling.Content += "Wind met index 0 = " + richtingNul.ToString();
}

```

17. Om onze applicatie te testen veranderen we ons startup project door rechts te klikken op ons project **Windrichtingen.WPF** en kiezen voor **Set as startup project**.

18. Na een klik op de knop "Toon voorspelling" zien we het label veranderen.



Met deze applicatie vragen we dus de index van een waarde van een enum op, alsook de waarde zelf.

- Willen we de index weten van een bepaalde waarde van een enum: `(int)EnumNaam.Waarde`

```
int huidigeWindNr = (int)WindRichting.Oost;
```

- Willen we de waarde opvragen van een enum: `EnumNaam.Waarde.ToString()`

```
string huidigeWind = WindRichting.Oost.ToString();
```

- Willen we de waarde weten van een bepaalde index van een enum: `(EnumNaam)index`

```
WindRichting richtingNul = (WindRichting)0;
```

We kunnen de index van een element ook zelf gaan bepalen. In de definitie geven we dan een waarde.

```
Bewering
```

```
{
    onwaar = -1,
    onbepaald,
```



```
    waar
}
```

In bovenstaand voorbeeld bepalen we de waarde van *onwaar*. Bij de andere elementen hebben we hier geen index geplaatst. Daardoor begint het systeem vanaf het volgende element te indexeren vanaf 0. *Onbepaald* is hier dus 0, *waar* is 1.

In de bovenstaande gevallen wordt er geen gegevenstype bepaald voor de enum.

Bovenstaand voorbeeld zouden we als volgt kunnen aanpassen:

```
enum Bewering: int
{
    onwaar = -1,
    onbepaald,
    waar
}
```

Het gekozen gegevenstype moet er wel één met gehele getallen zijn.

```
enum WindRichting:string { Oost, Zuid, Noord, West };
```



class System.String

Represents text as a series of Unicode characters.

Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

## 2.4 NUT VAN ENUMS

Het nut van Enums is beperkt. Je voorkomt er in elk geval tikfouten mee. Anderzijds kan de index van de enum gebruikt worden om andere zaken aan de enum te koppelen.

## 3 ARRAYS

### 3.1 WAT ZIJN ARRAYS?

Arrays worden vaak ook gegevensvelden of tabellen genoemd.

Aan een **variabele** kan je altijd maar **één waarde tegelijk** toewijzen .

De variabele *maand* kan bv. enkel de naam van één maand bevatten.

Een **array** daarentegen kan verschillende waarden bevatten. Men spreekt daarbij van **verschillende elementen**, waarbij elk element een waarde heeft. De afzonderlijke elementen gedragen zich bijgevolg als variabelen. In plaats van tien variabelen kan je dus ook een array van tien elementen gebruiken.

De array `maandenVanHetJaar` zou dus twaalf variabelen van het type `string` bevatten.

De elementen van een Array behoren allen tot hetzelfde type:

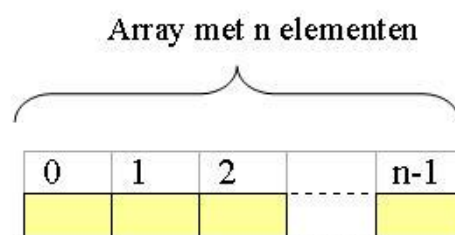
- een array van `ints`
- een array van `strings`
- een array van `Buttons`
- een array van arrays (die op hun beurt van het type `bool` zijn)
- ...

Arrays hebben nog een groot voordeel: de verschillende elementen worden doorlopend genummerd (index), waarbij het **eerste element altijd de index 0** krijgt. Als programmeur kunnen we deze nummering niet aanpassen, zoals bij enums wel het geval is.

Via deze index kun je waarden ophalen of bepalen:

```
string[] maand = new string[12];
maand[0] = "januari"; //waarde toekennen
MessageBox.Show(string.Format("De eerste maand is {0} ", maand[0])); //waarde ophalen
```

Met een lus (zie volgende hoofdstuk) kan je de elementen van de array doorlopen.



Een Array met n elementen: de elementen hebben een index 0 tot n-1.

De indexen van een array met 10 elementen lopen dus van 0 tot en met 9.



#### Verschil tussen arrays en enums

- Bij **enums** gaan we de waardes van de enum niet veranderen, maar gaan we deze **enkel opvragen**.
- Bij **arrays** kunnen we elk element gaan **opvragen en wijzigen**, zolang we binnen het **zelfde datatype** blijven

### 3.2 EEN ARRAY DECLAREREN

```
int[] leeftijden;
```

Hier werd een array-variabele met de naam **leeftijden** gedeclareerd; de elementen zullen van het type **int** zijn. De **vierkante haken** na het variabele-type geven dus aan dat de variabele een array van elementen van het gekozen type (in het voorbeeld int) zal zijn.

### 3.3 EEN ARRAY INSTANTIËREN

Een array is een referentietype: een arrayvariabele is dus een variabele die op de stack wordt bewaard en daar enkel het geheugenadres bevat naar het geheugenblok dat op de heap de eigenlijke elementen van de array bevat (zie hoofdstuk over 'value en reference').

Bij de instantiëring van een array wordt aangegeven hoeveel elementen de array zal bevatten. De elementen kunnen nu ook opgevuld worden. Deze elementen moeten van het type zijn dat je hebt aangegeven bij declaratie.

We kunnen op verschillende manier instantiëren.

#### 3.3.1 DECLAREREN, DE ELEMENTEN AFZONDERLIJK INSTANTIËREN

```
int[] leeftijden = new int[4];
//De array met de naam leeftijden zal 4 elementen bevatten van het type int.
//Deze elementen worden genummerd 0 - 3

leeftijden[0] = 18;
leeftijden[1] = 63;
leeftijden[2] = 42;
leeftijden[3] = 7;
```

Opmerking: het aantal elementen kan ook de waarde van een int-variabele zijn:

```
int aantalElementen = 4;
int[] leeftijden = new int[aantalElementen];
```

#### 3.3.2 VERKORTE SCHRIJFWIJZE: DIRECT INSTANTIËREN

Als je al onmiddellijk de beginwaarden van de elementen kent, kan je in C# een verkorte schrijfwijze gebruiken door de elementen tussen accolades te noteren:

```
int[] leeftijden = new int[4] { 18, 63, 42, 7 };
```

### 3.3.3 NOG KORTER

Het is zelfs toegestaan om het sleutelwoord `new` gevolgd door het type en het aantal elementen gewoon weg te laten:

```
int[] leeftijden = { 18, 63, 42, 7 };
```

## 3.4 TOEPASSING

Voeg een nieuw project toe aan je bestaande solution.

- Maak een event aan wanneer `MainWindow` geladen is
- In dit event declareren we een array met leeftijden
- In dit event tonen we een `MessageBox` die de derde waarde/leeftijd uit onze array `leeftijden` zal tonen.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        int[] leeftijden = { 18, 63, 42, 7 };
        MessageBox.Show("De derde leeftijd is: " + leeftijden[2].ToString());
    }
}
```



### Oefening

- Voeg vijf knoppen toe aan je WPF applicatie.
- Elke knop toont één waarde uit de array `leeftijden`:
  - Eerste knop toont de eerste waarde uit de array
  - Tweede knop toont de tweede waarde uit de array
  - ...
- Denk aan de scope van de array `leeftijden`, indien nodig mag deze scope gewijzigd worden.

## 3.5 DE KLASSE ARRAY

`System.Array` is een klasse binnen `.Net`.

Deze klasse bevat een aantal methoden (`Copy`, `Reverse`, ...) en eigenschappen (`Length`, ...), die we bij `Enums` niet vinden.

Arrays hebben echter ook een **belangrijk nadeel**: ze zijn fixed-length: om elementen toe te voegen of te verwijderen ben je dus verplicht een nieuwe `Array` te maken!

Het alternatief is om een array aan te maken die zeker genoeg elementen bevat, maar dan ga je niet zuinig om met het geheugen van de computer.

Arrays zijn dus pas echt interessant als het aantal elementen in een reeks nooit of heel zelden verandert.

Een oplossing hiervoor zijn parameter arrays (meer hierover in het hoofdstuk over lussen) en Collections.

```
void kaaprenVaarders()
{
    string[] vaarders = { "Jan", "Piet", "Joris" };
    string[] vaardersCopy = new string[3];
    vaarders.CopyTo(vardersCopy, 0);
    vaarders = new string[4];
    vaardersCopy.CopyTo(varders, 0);
    vaarders[3] = "Korneel";
    string kaaprenVaarders = vaarders[0] + ", " + vaarders[1] + ", " + vaarders[2] +
    " en " + vaarders[3] + "\nDie hebben baarden, zij varen mee.";

    MessageBox.Show(kaaprenVaarders);
}
```

## 4 COLLECTION KLASSEN

.Net bevat naast Arrays nog andere klassen om reeksen elementen te hanteren. We noemen ze de Collection klassen. Je vindt ze in de namespace `System.Collections`.

Om ze te gebruiken, dien je de namespace te vermelden in de code of een `using` statement te voorzien bovenaan.

```
using System.Collections;
...
List<string> personen;
```

Indien we geen gebruik maken een `using` statement moeten we onze declaratie voluit schrijven:

```
System.Collections.List<string> personen;
```

In de volgende onderdelen van deze cursus geven we wat uitleg over enkele van deze collections.

### 4.1 LIST

Een oplossing voor het probleem met de vaste lengte van arrays en enums vinden bij de List-klasse.

Een list bevat namelijk de methode `Add`.

```
void KaaprenVaarders()
{
    string kaaprenVaarders;
    List<string> vaarders = new List<string>();
    vaarders.Add("Jan");
    vaarders.Add("Piet");
    vaarders.Add("Joris");
    vaarders.Add("Korneel");

    kaaprenVaarders = vaarders[0] + ", " + vaarders[1] + ", " + vaarders[2] + " en " +
    vaarders[3] + "\nDie hebben baarden, zij varen mee.";
    MessageBox.Show(kaaprenVaarders, "Niet gesorteerd");

    vaarders.Sort();
    kaaprenVaarders = vaarders[0] + ", " + vaarders[1] + ", " + vaarders[2] + " en " +
    vaarders[3] + "\nDie hebben baarden, zij varen mee.";
    MessageBox.Show(kaaprenVaarders, "Gesorteerd");
}
```

In het voorbeeld hierboven zien we hoe eenvoudig het is om elementen aan een list toe te voegen. Dit lijkt zeer sterk op de manier om items aan een list- of combobox toe te voegen.

Ander voordeel van lists is dat je er meer methoden op kan toepassen zoals `Remove`, `Contains`, `Find`, `Insert`, `Reverse`, `Remove` en nog véél andere methodes.



**Meer informatie over de List klasse**

- [List<T> klasse](#)

## 4.2 DICTIONARY

Een Dictionary is een collection waarin paren van key/value (sleutel / waarde) bewaard kunnen worden. In tegenstelling tot andere collections kan een element van een Dictionary niet alleen opgezocht worden aan de hand van een index, maar ook van een sleutel.

```
void Vrienden()
{
    Dictionary<string, int> vrienden = new Dictionary<string, int>();

    vrienden.Add("Emmanuel", 40);
    vrienden.Add("Angela", 42);
    vrienden.Add("Donald", 25);

    string output = "";
    output += "De leeftijd van Emmanuel is " + vrienden["Emmanuel"] + "\n";
    output += "De tweede vriend(in) in de rij is " + vrienden.Keys.ElementAt(1) + "\n";
    lblOutput.Content = output;
}
```

De dictionary is zeer handig als je snel zaken wil opzoeken en vlot elementen wil verwijderen of toevoegen.

## 5 SAMENVATTING

	Index	Lengte	Waarden	Referentie
<i>Enum</i>	Aanpasbaar	Vast	Vast	Naam van de waarde Index
<i>Array</i>	Vast	Vast	Variabel	Index
<i>List</i>	Vast	Variabel	Variabel	Index
<i>Dictionary</i>	Vast	Variabel	Variabel	Index + veldnaam



### Code repository

De volledige broncode van dit voorbeeld is te vinden op

`git clone` <https://github.com/howest-gp-prb/cu-reeksen-arrays-collections.git>



# HOOFDSTUK 7

# LUSSEN



## INHOUDSOPGAVE

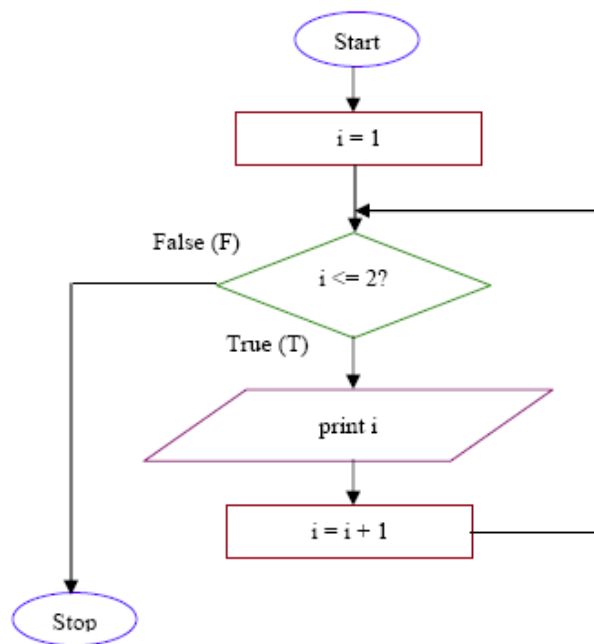
<b>1</b>	<b>LUSSEN</b>	<b>FOUT! BLADWIJZER NIET GEDEFINIEERD.</b>
<b>1.1</b>	<b>Inleiding</b>	<b>133</b>
<b>1.2</b>	<b>Soorten lussen</b>	<b>134</b>
1.2.1	While	134
1.2.2	For	135
1.2.3	Foreach	136
1.2.4	Do	137
1.2.5	Break	139
1.2.6	Continue	139
1.2.7	Scope van variabelen binnen en lus	140



# 1 INLEIDING

- Maak een nieuwe Visual Studio solution Lussen.
- Maak in de solution Lussen een nieuwe WPF App Lussen.Wpf.

Een programmalus wordt gebruikt wanneer een statement of een groep statements een aantal maal herhaald moet worden. In de afbeelding hieronder wordt dit visueel voorgesteld:



Lussen vormen een basisstructuur in om het even welke programmeertaal. Daardoor is het uitermate belangrijk er een goed begrip van te hebben.

Voorbeelden waarbij gebruik wordt gemaakt van lussen zijn:

- Een (variabele) boodschap moet een aantal keer op het scherm worden gezet
- Een getallenreeks wordt overlopen bij bijvoorbeeld het zoeken naar delers van een getal
- Een reeks waarden dient gesorteerd te worden
- De namen van alle bestanden in een map moeten getoond worden
- Een tekstbestand moet regel per regel worden ingelezen
- Records uit een databasetabel moeten overlopen worden
- ...

## 2 SOORTEN LUSSEN

### 2.1 WHILE

Een while-lus kan gebruikt worden om statements uit te voeren zolang een bepaalde voorwaarde waar (true) is.

```
while(booleaanse expressie)
{
    statement(s);
}
```

- De expressie moet een boolean als resultaat geven.
- De expressie moet tussen ronde haken genoteerd worden.
- Als de expressie al direct als false geëvalueerd wordt, worden de statements niet uitgevoerd.
- Als er meer dan één statement uitgevoerd moet worden, dan plaats je deze tussen accolades .

Voorbeeld: lussen van 0 tot en met 5, waarde van de teller wordt getoond:

```
int teller = 0;
string resultaat = "";
while(teller < 6)
{
    Console.WriteLine("Teller is nu gelijk aan " + teller.ToString());
    resultaat += "Teller: " + teller + "\n";
    teller++;
}
MessageBox.Show(resultaat);
```

## VOOBEELD

### De computer een getal laten raden

```
Random random = new Random();
const int MAX_GETAL = 10000;

public MainWindow()
{
    InitializeComponent();
}

private void BtnRaad_Click(object sender, RoutedEventArgs e)
{
    int teRaden;
    int gok = -1;
    int pogingen = 0;

    teRaden = int.Parse(txtTeRaden.Text);

    while (gok != teRaden)
    {
        gok = random.Next(1, MAX_GETAL + 1);
        lstGokjes.Items.Insert(0, gok);
        pogingen++;
    }
    lblFeedback.Content = $"Het getal {teRaden} is geraden in {pogingen} pogingen";
}
```



#### Code repository

De volledige code van dit voorbeeld is te vinden op

`git clone https://github.com/howest-gp-prb/cu-lussen-laat_de_computer_raden.git`

## 2.2 FOR

Via een for-lus kunnen we code telkens opnieuw laten uitvoeren aan de hand van een teller tot die teller een bepaalde waarde bereikt heeft. Elke keer dat de lus doorlopen wordt, wordt de teller verhoogd met een vastgestelde waarde.

We vragen de computer dus:

- te tellen van getal x tot y
- in stappen van een bepaalde grootte (meestal 1)
- bij elke tel bepaalde code uit te voeren

In het voorbeeld hieronder laten we de computer van nul tot vijf tellen in stappen van één

```
string resultaat = "";
```

```
for (int i = 0; i <= 5; i++)
{
    resultaat += "De teller i is nu gelijk aan: " + i.ToString() +
Environment.NewLine;
}

MessageBox.Show(resultaat);
```

Door het for-statement als volgt aan te passen

```
for (int i = 0; i <= 10; i += 2)
```

wordt er van nul tot tien geteld in stappen van twee.

Het start- en eindgetal kan ook afhankelijk zijn van een variabele of het resultaat van een functie:

```
string woord = "programmeren";
string resultaat = "Het woord " + woord + " bevat de volgende letters:\n";

for (int i = 0; i < woord.Length; i++)
{
    resultaat += i.ToString() + " - " + woord.Substring(i,1) + Environment.NewLine;
}

MessageBox.Show(resultaat);
```

## 2.3 FOREACH

Bij arrays en collections kunnen we alle elementen overlopen en bij elk element code laten uitvoeren.

In het laatste voorbeeld van de for-lussen zouden we de letters van een woord kunnen opvragen via een foreach-lus. Een string is namelijk een collection van characters.

```
string woord = "programmeren";
string resultaat = "Het woord " + woord + " bevat de volgende letters:\n";

foreach (char letter in woord)
{
    resultaat += letter + Environment.NewLine;
}

MessageBox.Show(resultaat);
```

In het volgende voorbeeld doorlopen we een array:

```
string resultaat = "";
string[] seizoenen = { "lente", "zomer", "herfst", "winter" };
```



```
foreach (string seizoen in seizoenen)
{
    resultaat += seizoen + Environment.NewLine;
}

MessageBox.Show(resultaat);
```

Ook een list kan via een foreach-lus doorlopen worden:

```
string resultaat = "Het dubbel van de getallen van nul tot en met tien:\n";
List<int> getallenTotTien = new List<int>();

for (int i = 0; i <= 10 ; i++)
{
    getallenTotTien.Add(i);
}

foreach (int getal in getallenTotTien)
{
    resultaat += "Het dubbel van " + getal + " = " + getal * 2 + Environment.NewLine;
}

MessageBox.Show(resultaat);
```

## 2.4 DO

Bij while- en for-lussen wordt de voorwaarde bij de start van de lus getest. Als aan de voorwaarde van bij het begin niet voldaan is, worden de statements binnen de lus nooit uitgevoerd.

Een do-lus bevat de voorwaarde **na** de statements:

```
string resultaat = "Tellen van één tot vijf:\n";
int i = 1;
do
{
    resultaat += "De teller i is nu gelijk aan: " + i.ToString() +
Environment.NewLine;
    i++;
} while (i <= 5);
MessageBox.Show(resultaat);
```

De lusstatements worden hier dus minstens één keer uitgevoerd.

In andere programmeertalen bestaat soms de do ... until-lus. In C# bestaat die niet, maar uiteindelijk komt zo'n lus overeen met een do .... while zolang een statement false is. In pseudocode zou je dus kunnen zeggen dat

```
teller = 0
do
```

```
    verhoog teller met 1  
until teller == 5
```

op hetzelfde neerkomt als

```
teller = 0  
do  
    verhoog teller met 1  
while teller != 5
```

## 3 BREAK & CONTINUE

### 3.1 BREAK

Normaal gezien wordt een lus doorlopen tot alles overlopen is (einde array of collection, bereiken eindwaarde in for-lus, niet meer voldoen aan een bepaalde conditie bij do en while).

Het is mogelijk om de lus vroegtijdig te onderbreken als aan een bepaalde voorwaarde voldaan wordt. Hiervoor hebben we een if-statement nodig. Een toepassing hiervan is het doorlopen van een reeks (array, list ...) tot er een bepaalde waarde gevonden is.

Zo wordt in de toepassing `LussenOnderbreken` gezocht in de array *namen* tot er een bepaalde naam is gevonden. Eens die gevonden is, heeft het geen zin om verder te zoeken in de lijst. We gebruiken hier een **break**.

```
int ZoekNaamIndex(string speler)
{
    int index = -1;
    foreach (string naam in namen)
    {
        index++;
        if (naam == speler)
        {
            break;
        }
    }
    return index;
}
```

### 3.2 CONTINUE

Met een **continue**-statement wordt de code in de lus onderbroken. Als aan de gestelde voorwaarde voldaan wordt (met een if-statement), wordt de code die volgt op de continue niet uitgevoerd. Als de eindvoorwaarde nog niet bereikt is, gaat de lus echter gewoon verder. Een continue-statement kun je dus gebruiken als je een lus nodig hebt, waarbij code uitgevoerd moet worden **behalve** in bepaalde gevallen.

Bvb: Je zou een bepaalde getallenreeks één voor één kunnen overlopen om de getallen die deelbaar zijn door drie te zoeken. Bij elk getal wordt de tekst '*x is deelbaar door drie*' getoond. Indien de modulo van een getal niet gelijk is aan nul, zou een continue-statement ervoor kunnen zorgen dat de tekst niet getoond wordt.

```
void ToonDrievouden()
{
    for (int i = 0; i < 100; i++)
    {
        if (i % 3 != 0)
        {
            continue;
        }
        Console.WriteLine($"{i} is deelbaar door 3");
    }
}
```

## 4 SCOPE VAN VARIABLEN BINNEN EN LUS

Een variabele die binnen een lus gedeclareerd wordt, verdwijnt uit het geheugen van zodra de lus doorlopen is. Buiten de lus kan de variabele niet gebruikt worden. In het volgende voorbeeld zou je bij de `Console.WriteLine`-statements dus telkens een foutmelding krijgen:

```
for (int i = 0; i<10; i++)  
{  
    string getal = i.ToString();  
}  
Console.WriteLine(i);  
Console.WriteLine(getal);
```

# **HOOFDSTUK 8**

# **ERRORS & EXCEPTIONS**



## INHOUDSOPGAVE

<b>1</b>	<b>RUNTIME ERRORS</b>	<b>145</b>
<b>1.1</b>	<b>Runtime Errors</b>	<b>145</b>
1.1.1	Voorbeeldscenario	145
1.1.2	Logische fouten	145
<b>2</b>	<b>EXCEPTIES AFHANDELEN</b>	<b>146</b>
<b>2.1</b>	<b>Try Catch</b>	<b>150</b>
<b>2.2</b>	<b>Try Catch</b>	Fout! Bladwijzer niet gedefinieerd.
<b>2.3</b>	<b>Soorten Exceptions</b>	<b>151</b>
2.3.1	Meerdere exceptions afhandelen	151
2.3.2	Alle exceptions afhandelen	152
<b>2.4</b>	<b>Throw</b>	<b>153</b>
<b>2.5</b>	<b>Finally</b>	<b>153</b>
<b>2.6</b>	<b>Het checked keyword en overflows</b>	<b>155</b>





## 1 RUNTIME ERRORS

Je hebt nu al geleerd hoe je met statements, methoden en variabelen omgaat in C#.

In dit hoofdstuk leer je hoe je kan omgaan met fouten die kunnen optreden tijdens de uitvoering van een toepassing. Dit soort fouten zijn zogenaamde **runtime errors**, of worden ook wel **Exceptions** genoemd.

We hebben het hier dus niet over **syntactische** fouten die je maakt tijdens het typen van je programmacode, of fouten die je maakt tegen de "spelregels" van C#, dergelijke fouten worden reeds tijdens het compileren van je code gedetecteerd en verhinderen de uitvoering van je programma (**compile-time errors**).

### 1.1 RUNTIME ERRORS

Dit hoofdstuk gaat in op fouten die tijdens het uitvoeren van een programma kunnen optreden (runtime-errors). Het is belangrijk dat je in je programma rekening houdt met mogelijke foute handelingen van een gebruiker, mogelijke hardware-matige problemen (netwerkverbindingen vallen uit, bestand is onleesbaar, ...), ...

#### 1.1.1 VOORBEELDSCENARIO

Je maakt bijvoorbeeld een toepassing die twee getallen die door de gebruiker worden ingegeven optelt en de som op het scherm toont.

Wat gebeurt er als de gebruiker letters invult of helemaal niks?

De bedoeling van dit hoofdstuk is dat je dergelijke mogelijke probleemsituaties kan inschatten en softwarematig een oplossing biedt, zonder dat je toepassing "crasht".

Hierbij wordt niet bedoeld dat je elk probleem moet kunnen oplossen: letters kunnen we nu eenmaal niet optellen. Je kan de gebruiker wel op de hoogte stellen van het probleem en informeren hoe dit probleem kan worden opgelost door bijvoorbeeld een bericht te tonen: "Gelieve numerieke waarden in te geven.".

Je kan dus in je programmacode een pad voorzien voor het goede verloop van de toepassing, maar je moet er zeker rekening mee houden dat dit verloop weleens verstoord kan worden.

#### 1.1.2 LOGISCHE FOUTEN

Het is belangrijk in te zien dat we dergelijke onvoorziene omstandigheden in een programma uitzonderingen of exceptions noemen, deze kunnen we gestructureerd in programmacode afhandelen. **Logische fouten** zijn daarentegen verkeerde denkwijzen of verkeerde programma algoritmes: als je de gebruiker belooft twee getallen op te tellen, maar je berekent het verschil, dan zal je programma wellicht niet vastlopen, maar de gebruiker zal ook niet echt gelukkig zijn.

Logische fouten zijn dus geen run-time fouten en om ze te detecteren zijn uitvoerige **tests** van je programma nodig.

## 2 EXCEPTIES AFHANDELEN



### Code repository

De volledige broncode van dit onderdeel is te vinden op

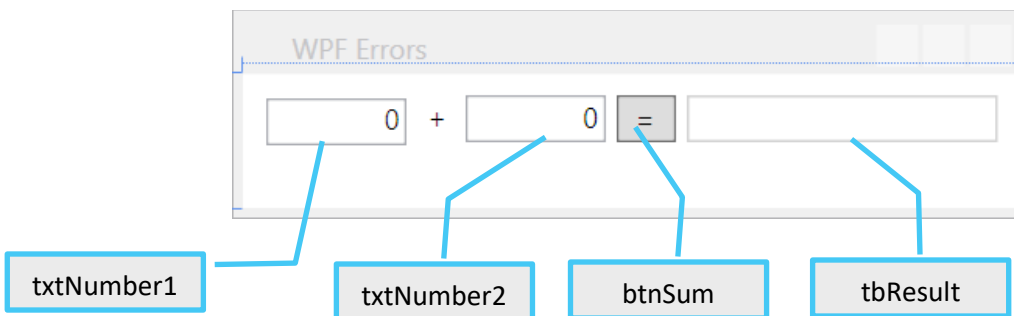
git clone <https://github.com/IVOBrugge/CsCourse.ErrorsAndExceptions.git>

### Startsituatie

[https://github.com/IVOBrugge/CsCourse.ErrorsAndExceptions/releases/tag/tutorial\\_start](https://github.com/IVOBrugge/CsCourse.ErrorsAndExceptions/releases/tag/tutorial_start)

Maak een eenvoudige applicatie om het principe van foutafhandeling te leren kennen. Bij het klikken op een knop zet de applicatie twee ingevoerde string waarden om naar gehele getallen en toont de som in een TextBlock control.

30. Maak een WPF Window met twee TextBoxes, een Label, een Button en een TextBlock control:

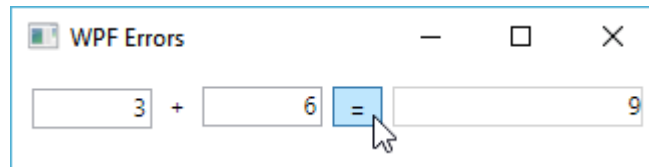


31. Genereer een Event Handler methode voor de knop via het Properties venster.  
32. Wijzig de code van de codebehind als volgt:

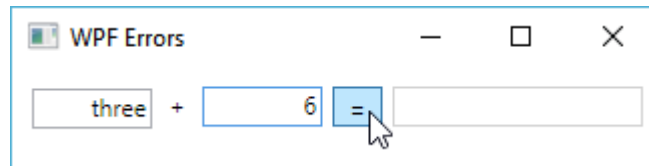
```
private void BtnSum_Click(object sender, RoutedEventArgs e)
{
    int numberLeft = int.Parse(txtNumber1.Text); //parse string to int
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}

/// <summary>
/// Calculates the sum of two numbers
/// </summary>
private int CalculateSum(int number1, int number2) {
    return number1 + number2;
}
```

33.  
34. Compileer en voer de applicatie uit.

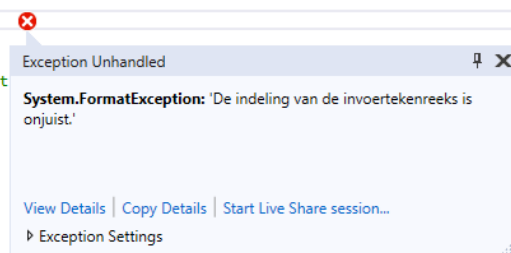


35. Voer een waarde in die niet kan worden omgezet naar een integer. Er doet zich nu een run-time error (van het type `FormatException`) voor en de applicatie **crasht**.



```
private void btnSum_Click(object sender, RoutedEventArgs e)
{
    int numberLeft = int.Parse(txtNumber1.Text); //parse string to int
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set
}

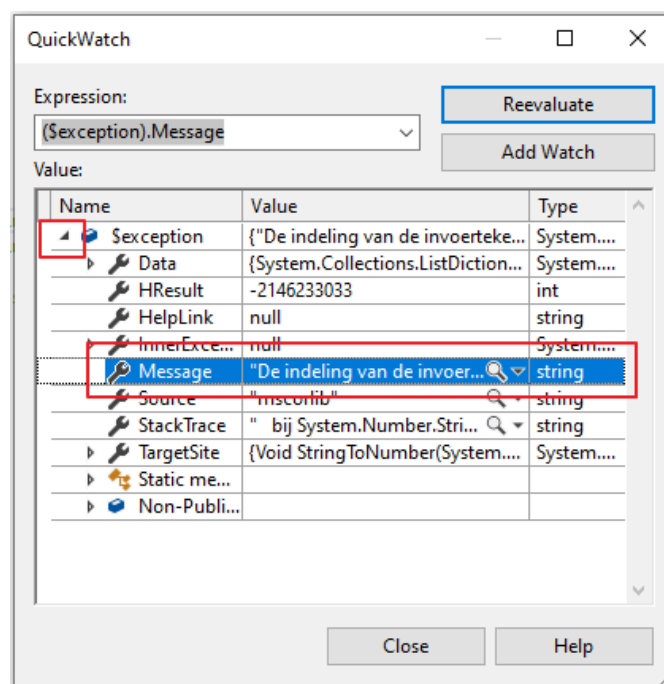
/// <summary>
/// Calculates the sum of two numbers
/// </summary>
1 reference
private int CalculateSum(int number1, int number2)
{
    return number1 + number2;
}
```



Als programmeur zien wij deze melding verschijnen in Visual Studio, wanneer we de toepassing starten. Visual Studio kijkt achter onze schouder mee bij het werken met de toepassing en grijpt in wanneer het verkeerd loopt.

36. Om meer details te zien van de Exception klik je in het venster op de link **View Details**:

37.



Om een som te kunnen oplossen moet er gewerkt worden met variabelen van een numeriek type (int, decimal, float, enz.). Daarom moeten de string waarden omgezet worden naar een numerieke waarde. In dit geval wordt een int verwacht.

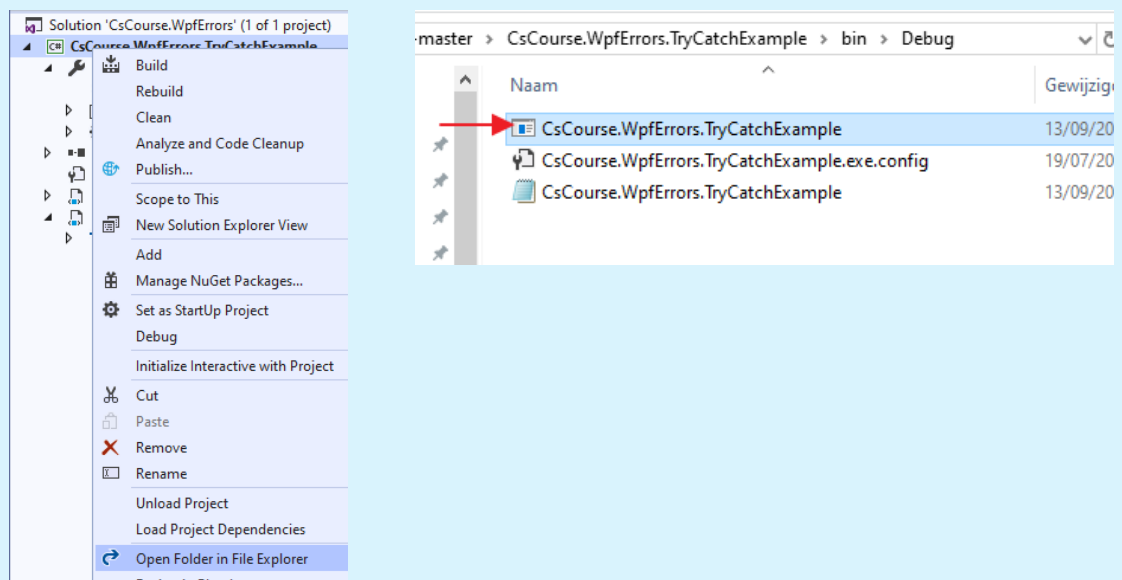
De fout doet zich voor omdat de string variabele die de gebruiker heeft ingetikt, "three", **niet** kan worden omgezet naar een variabele van het type int. Dit levert een FormatException met de melding "Input string was not in a correct format." of op een Nederlands Windowssysteem: "De indeling van de invoertekenreeks is onjuist."

Wanneer je dit programma verspreidt voor gebruik, dan voeren de gebruikers een .exe-bestand uit, los van Visual Studio. Het uitvoerbare bestand vind je in de /bin/Debug map van je huidige toepassing.

38. Ga naar de /bin/Debug map van je applicatie en voer de executable uit.



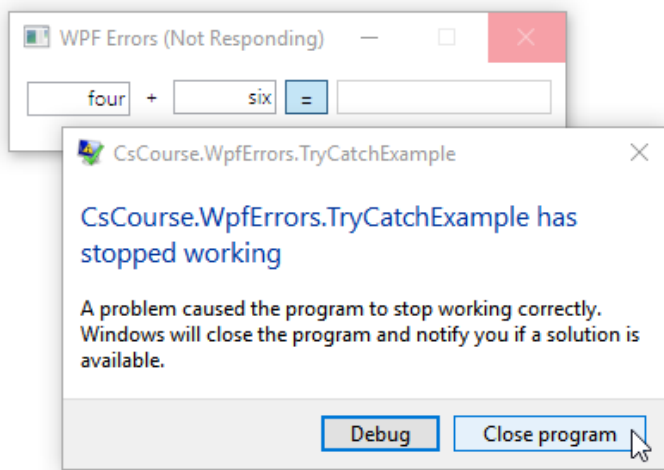
**Tip :**  
klik met de rechtermuisknop op je project en kies in het snelmenu voor "Open Folder in File explorer"



- 39.

40. Geef een ongeldige waarde in voor de numerieke waarde(n) en klik op de knop.

De applicatie crasht en wordt afgesloten na een summiere foutmelding (of zelfs helemaal geen melding en sluit gewoon af) :



41.

Het is voor de gebruiker onduidelijk wat de precieze oorzaak van de fout was. Om meer duidelijkheid te scheppen en te verhinderen dat de applicatie crasht moeten we deze mogelijke fout afhandelen.

## 2.1 TRY CATCH

C# maakt het eenvoudig om foutafhandelingscode te scheiden van code die de hoofdflow van je programma vormt. Om programma's te maken die zich bewust zijn van fouten doe je twee zaken:

- Schrijf je code binnen een try-blok. Wanneer de code wordt uitgevoerd, wordt gepoogd elk statement na elkaar uit te voeren. Wanneer een fout optreedt, wordt het try-blok verlaten en wordt de fout afgehandeld binnen een catch-blok.
- Schrijf na een try-blok één of meer catch-blokken om mogelijke fouten af te handelen.

Pas dit toe in het voorbeeldprogramma.

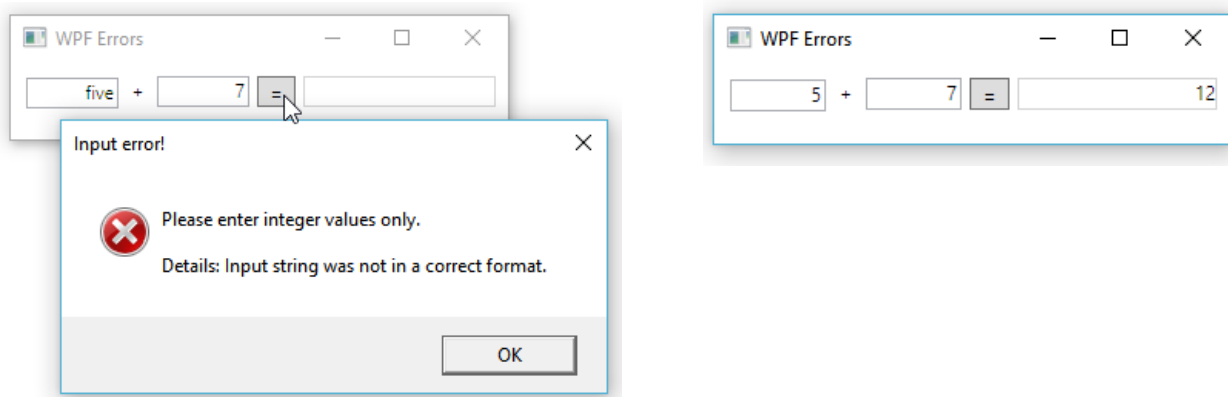
42. Zet de statements die de fout kunnen veroorzaken in een try -blok en de foutafhandelingscode in het bijbehorende catch-blok daaronder.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
        "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

Indien er zich nu een FormatException voordoet in de instructies die in het try-blok staan, dan wordt die fout opgevangen in het catch-blok. Er wordt dan een MessageBox getoond die de gebruiker inlicht over de precieze oorzaak van de fout.

Omdat je de fout **opvangt** in een catch-blok zal de applicatie **niet crashen**. De gebruiker krijgt de gelegenheid om een nieuwe invoer te proberen.

43. Voer de applicatie uit en test de foutafhandeling door foutieve invoer in te geven.



## 2.2 SOORTEN EXCEPTIONS

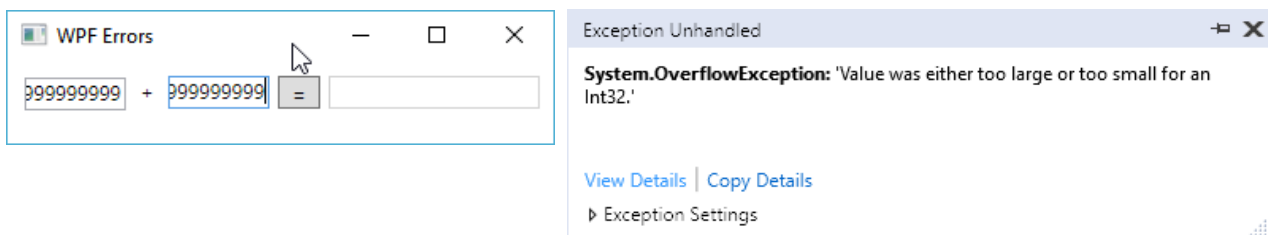
### 2.2.1 MEERDERE EXCEPTIONS AFHANDELEN

Er bestaan heel wat soorten exceptions. De exception die je zonet hebt afgehandeld was van het type `FormatException` dat meestal voorkomt bij het *parsen* van een string naar een variabele van een ander type.

Onze applicatie is nog niet helemaal vrij van potentiële crashes. Een variabele van het type `int` kan nooit groter zijn dan 2.147.483.647 en nooit kleiner dan -2.147.483.648. Indien deze waarden overschreden worden dan krijg je een `OverflowException`.

44. Voer de applicatie uit en geef getallen in waarvan de som groter is dan 2.147.483.647

- De applicatie crasht opnieuw, ditmaal met een `OverflowException`.



45. Voeg nog een catch-blok toe aan de je try/catch structuur, dat de `OverflowException` moet opvangen.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
        "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
catch (OverflowException oEx)
{
    MessageBox.Show("The numbers you entered are too large or too small.\n\nDetails: "
+ oEx.Message,
        "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

46. Voer de applicatie nogmaals uit en test opnieuw. De applicatie is nu een stuk robuuster geworden en zal in deze twee specifieke gevallen niet meer crashen.

### 2.2.2 ALLE EXCEPTIONS AFHANDELEN

Het is onbegonnen werk om een catch-blok te schrijven voor elk type fout dat zich kan voordoen. Je gebruikt de specifieke catch-blokken enkel om een verschillende foutafhandeling te leveren. Bijvoorbeeld als de weer te geven tekst verschilt of als je enkel een OverflowException naar een apart logbestand wenst te registreren.

Om je applicatie te beschermen tegen elke mogelijke fout kan je de joker Exception gebruiken in een catch-blok.

47. Vervang het catch-blok voor de OverflowException met een algemener catch-blok dat **elke** soort Exception kan afhandelen.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
        "Input error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
catch (Exception ex) //catches all unhandled exceptions (FormatException are already
handled above)
{
    //generic error message is shown
    MessageBox.Show("An error has occurred.\n\nDetails: " + ex.Message,
        "Input error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Wens je geen gebruik te maken van het Exception object, dat onder meer de foutmelding bevat, dan kan dit nog korter:

```
try
{
    //error prone code goes here
}
catch
{
    //catches all unhandled exceptions here
    MessageBox.Show("An unknown error has occurred!");
}
```

Als je meerdere catch-blokken gebruikt, dan moet het algemene catch-blok steeds de **laatste** zijn. Op die manier kan je de specifiekere exception soorten eerst afhandelen.



## 2.3 THROW

Je kan zelf ook Exceptions opgooien, om te voorkomen dat de applicatie een scenario uitvoert dat niet toegestaan is. Zo kan je jezelf of je collega-programmeurs verplichten om een foutafhandeling uit te werken voor dat scenario.

Eerst moet je een nieuwe Exception instantie aanmaken met het `new` keyword. Vervolgens gooi je de exception instantie op met het `throw` keyword. Bij het maken van een instantie kan je zelf kiezen welk type fout je wenst op te gooien, of je houdt het algemeen met de klasse `Exception`.

Stel dat je wenst te verhinderen dat de `CalculateSum()` methode negatieve getallen optelt, dan kan je hier een controle op doen en, indien een negatief getal wordt gedetecteerd, een fout van het type `ArgumentOutOfRangeException` opgooien.

48. Wijzig de code van de `CalculateSum()` methode als volgt:

```
private int CalculateSum(int number1, int number2) {
    //check if either number is negative
    if (number1 < 0 || number2 < 0)
    {
        throw new ArgumentOutOfRangeException("Terms should not be negative");
    }
    //this instruction will only be executed reached if no exception occurred
    return number1 + number2;
}
```

49. Voer de applicatie uit en test of je nu nog negatieve getallen kan invoeren. De Exception wordt afgehandeld door het algemene catch-blok in de Event Handler van de knop.

## 2.4 FINALLY

Wanneer een fout optreedt in een `try`-blok wordt de flow van een programma veranderd. De instructies na de plaats van de fout worden niet meer uitgevoerd. In plaats daarvan wordt het overeenkomstig `catch` blok uitgevoerd.

Soms is het nodig om code te schrijven die altijd moet uitgevoerd worden, ongeacht het feit of er een fout is opgetreden. Dit kan je bereiken door **na het laatste** `catch` blok (of na het `try` blok, indien je geen foutafhandeling wenst) een `finally` blok te voorzien.

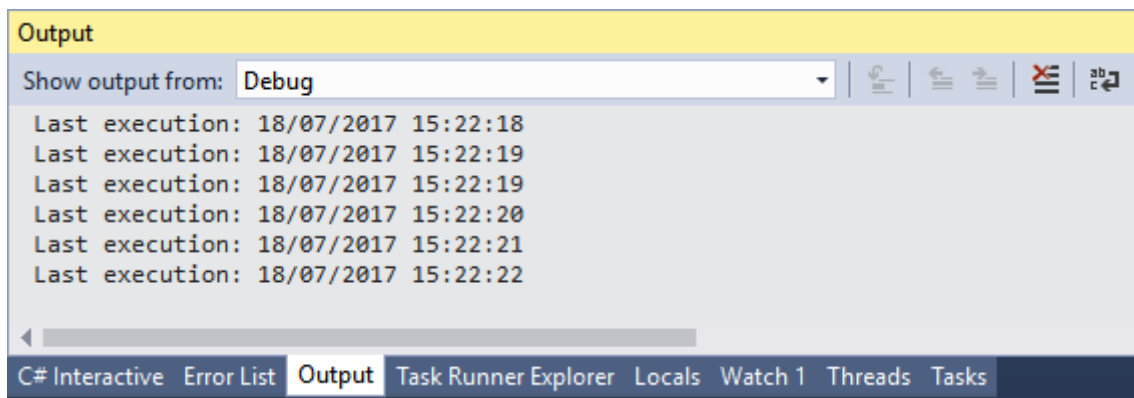
```
try
{
    //code met die mogelijke fouten kan veroorzaken komt hier te staan
}
catch(Exception ex)
{
    //vangt elke exception op
}
finally
{
    //code die steeds moet worden uitgevoerd, of er zich nu een fout voordoet of niet.
```

```
}
```

50. Voeg een finally blok toe onder het laatste catch blok dat steeds de tijd weergeeft in het Output venster van Visual Studio. Hiervoor heb je de namespace System.Diagnostics nodig als using statement.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch (FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
        "Input error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
catch (Exception ex) //catches all unhandled exceptions (FormatException are already
handled above)
{
    //generic error message is shown
    MessageBox.Show("An error has occurred.\n\nDetails: " + ex.Message,
        "Input error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
finally
{
    //always write to output window (in Visual Studio)
    Debug.WriteLine("Last execution: {0:dd/MM/yyyy HH:mm:ss}", DateTime.Now);
}
```

51. Voer de applicatie uit. Tel zowel geldige als ongeldige getallen op. In het Output venster van Visual Studio zie je nu telkens de tijd van uitvoering verschijnen.



Je kan het Output venster zichtbaar maken tijdens het debuggen van je applicatie via **View → Output**. Het venster staat standaard onderaan in het Visual Studio hoofdvenster.

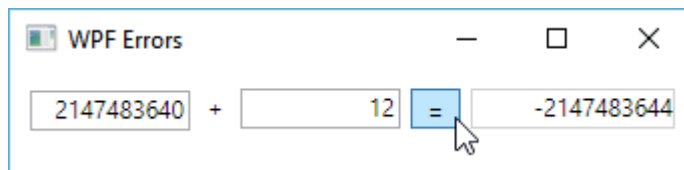
## 2.5 HET CHECKED KEYWORD EN OVERFLOWS

Zoals eerder al aangegeven, zijn de numerieke datatypes beperkt in grootte. De bovengrens van een `int` is 2147483647 ( $2^{31} - 1$ ) en de ondergrens is -2147483648 ( $2^{31}$ ). Binair is dat:

2147483647 = 01111111111111111111111111111111  
 - 2147483648 = 11111111111111111111111111111111

Als de bovengrens van een `int` wordt overschreden in een optelling, dan wordt er gewoon verder gerekend vanaf de ondergrens. Dat is zo voor numerieke waarden die negatief kunnen zijn (signed) of steeds positief zijn (unsigned).

52. Voer de applicatie uit. Tel de 2147483640 op met 12. Het resultaat hiervan ligt boven de grens van een `int`.



Dit probleem rijst omdat de .NET runtime toestaat dat de berekening leidt tot een overflow. Deze logische fout wordt toegestaan omdat een controle op overflow voor elke berekening zou leiden tot een gigantisch performantieverlies. Door het ontbreken van een controle noemen we dit een **unchecked** bewerking.

Eigenlijk gaat de .NET CLR er van uit dat je voor berekeningen steeds een realistische keuze in datatype maakt waarbij je aanneemt dat de grens (in de verste verte) niet bereikt zal worden.

Soms kan je dit echter niet voorzien, in dergelijke gevallen kan je een **check** doen op eventuele overflow met het `checked` keyword.

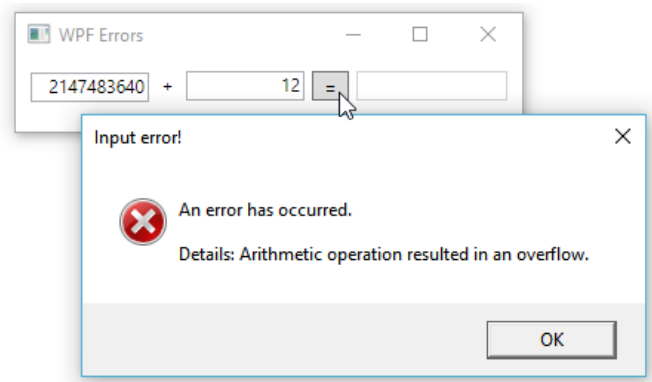
53. Wijzig de code van de methode `CalculateSum()`

```
private int CalculateSum(int number1, int number2)
{
    //check if either number is negative
    if (number1 < 0 || number2 < 0)
    {
        throw new ArgumentOutOfRangeException("Terms should not be negative");
    }
    //this instruction will only be executed reached if no exception occurred
    return checked(number1 + number2);
}
```

54. Voer de applicatie uit en test dezelfde berekening.

Door het gebruik van het `checked` keyword zal de runtime deze keer een `OverflowException` genereren, net zoals bij het Parsen.

De fout wordt opgevangen in het laatste catch-blok.



# HOOFDSTUK 9

# KLASSEN EN

# OBJECTEN



## INHOUDSOPGAVE

<b>1</b>	<b>INLEIDING</b>	<b>161</b>
<b>1.1</b>	<b>Classificatie</b>	<b>161</b>
<b>1.2</b>	<b>Abstractie (black-box)</b>	<b>161</b>
1.2.1	Encapsulatie	161
<b>2</b>	<b>VOORBEREIDING CURSUSVOORBEELD</b>	<b>162</b>
<b>3</b>	<b>CLASS LIBRARY</b>	<b>163</b>
<b>4</b>	<b>GEBRUIK VAN KLASSEN</b>	<b>165</b>
<b>4.1</b>	<b>Een klasse aanmaken</b>	<b>165</b>
<b>4.2</b>	<b>Objecten aanmaken</b>	<b>167</b>
<b>4.3</b>	<b>Praktisch voorbeeld van klassen en objecten</b>	<b>168</b>
4.3.1	Klassen en velden public maken	168
4.3.2	Referentie maken naar een klassenbibliotheek	169
4.3.3	Gebruik van de klasse Auto	172
<b>4.4</b>	<b>Constructors</b>	<b>173</b>
4.4.1	Default constructor	173
4.4.2	Constructors overloaden	174
<b>4.5</b>	<b>Gebruik van static</b>	<b>177</b>
4.5.1	Static fields	177
4.5.2	Static methods	179
<b>4.6</b>	<b>Partial klassen</b>	<b>183</b>
<b>4.7</b>	<b>Anonieme klasse</b>	<b>185</b>
<b>4.8</b>	<b>Besluiten</b>	<b>186</b>
<b>5</b>	<b>GEBRUIK VAN DE KLASSENBIBLIOTHEEK</b>	<b>187</b>





## 1 INLEIDING

In de lessenreeks heb je tot hiertoe gebruikt gemaakt van een aantal klassen: `Console`, `MessageBox`, `Exception`, ...

Het .Net Framework telt echter **duizenden** dergelijke klassen, die je kan uitbreiden door zelf nieuwe klassen aan te maken.

In dit deel leer je hoe je een klasse kan aanmaken. Je kunt dan zelf objecten van deze klasse creëren en bewerken.

### 1.1 CLASSIFICATIE

Wanneer je een klasse ontwerpt, ga je informatie systematisch classificeren in een object. Dit lijkt mogelijks wat wereldvreemd, maar dit classificeren is iets wat eigenlijk iedereen doet, niet enkel programmeurs.

Je hebt in het dagelijks leven bijvoorbeeld een concept met de naam auto. Niemand kan 100 % beschrijven wat het concept auto eigenlijk inhoud. Het concept auto is in feite een blauwdruk, een plan voor hetgeen waarvan we een concreet beeld hebben. Een object van dit concept (of de klasse) auto is dan bijvoorbeeld een grijze, handmatig versnelde Opel met een lederen interieur en ABS.

Een klasse in het object georiënteerd programmeren is de basis voor eender welk object.

Een klasse vormt een sjabloon dat aangeeft hoe een object van een klasse er zal gaan uitzien.

In het dagelijkse leven zou je ook kunnen zeggen dat de klasse `Auto` een sjabloon is voor alle reële auto's. Dit sjabloon beschrijft welke **kenmerken** (kleur, topsnelheid, motorisatie ...) een auto kan hebben en welke **acties** (accelereren, remmen, draaien ...) ermee uitgevoerd kunnen worden. Bij elk concreet geval (object) van een klasse kan de waarde van een kenmerk telkens verschillen. Mijn auto is beigegekleurig en kan maximum 170 km/u. rijden; de zwarte BMW van mijn buurman kan een heel stuk sneller.

### 1.2 ABSTRACTIE (BLACK-BOX)

Een van de bedoelingen van **object georiënteerd** programmeren (waar klassen en hun instanties genaamd objecten aan de basis liggen) is het opsplitsen van je applicatie in functionele eenheden.

- Wanneer je een WPF `Window` aanmaakt, hoef je je geen zorgen te maken hoe deze getoond wordt, hoe de knopjes minimaliseren, maximaliseren en sluiten worden aangemaakt.
- Wanneer je de methode `WriteLine` van de klasse `Console` uitvoert, hoef je je niet af te vragen hoe deze zin in de `Console` wordt getoond.
- Wanneer je de methode `Versnel` van je eigen klasse `Auto` uitvoert, hoef je je niet af te vragen wat de implementatie van die methode is. De implementatie zal in concreto de uitvoering van de statements binnen de methode inhouden.

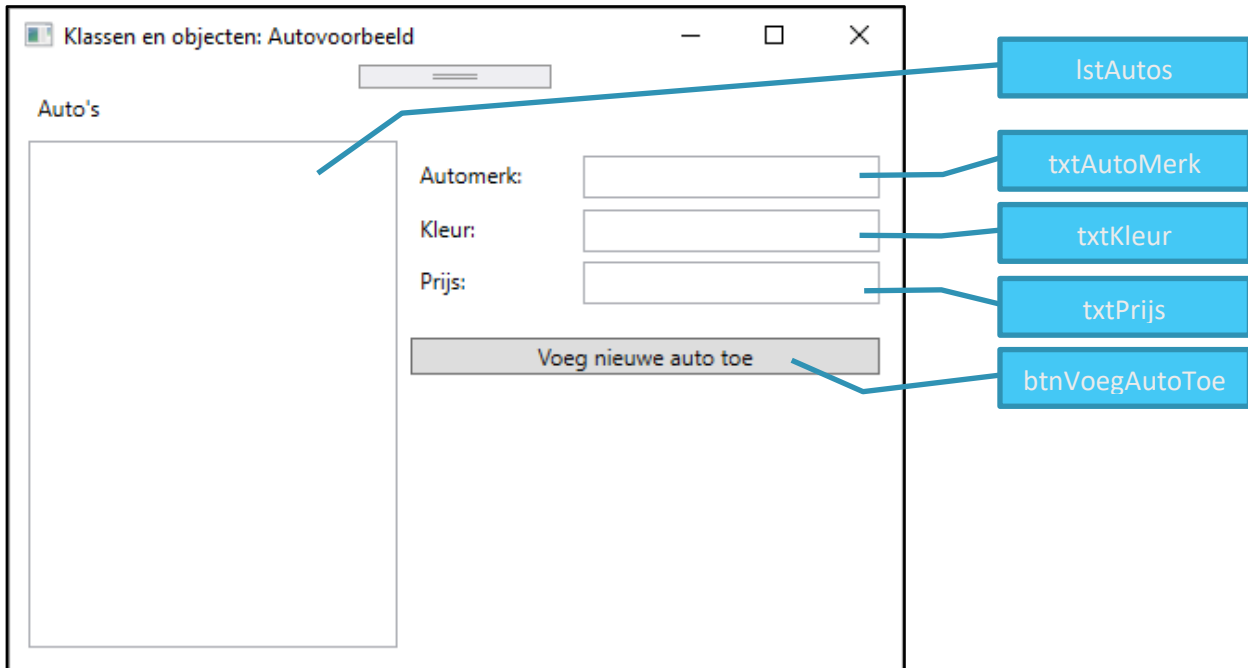
### 1.3 ENCAPSULATIE

Een ander doel van het gebruik van klassen, het ontwerpen van een object georiënteerd model, is het afschermen van informatie tegen oneigenlijk gebruik (het aantal leerlingen van een klas kan niet negatief zijn, een credit card nummer zul je kunnen ingeven, maar niet terug opvragen ...).

Een manier om dit te doen is het werken met Properties i.p.v. directe fields (variabelen). We diepen dit later verder uit.


## 2 VOORBEREIDING CURSUSVOORBEELD

Om het gebruik van klassen en objecten te tonen, zullen we gebruik maken van een WPF applicatie. Om geen tijd te verliezen met de lay-out is er voor jullie reeds een solution voorzien waarin de lay-out reeds voor jullie werd gemaakt. Op het einde van dit hoofdstuk gaan we aan de slag met deze WPF-applicatie.



### Code repository

De broncode van dit cursusvoorbeeld is te vinden op

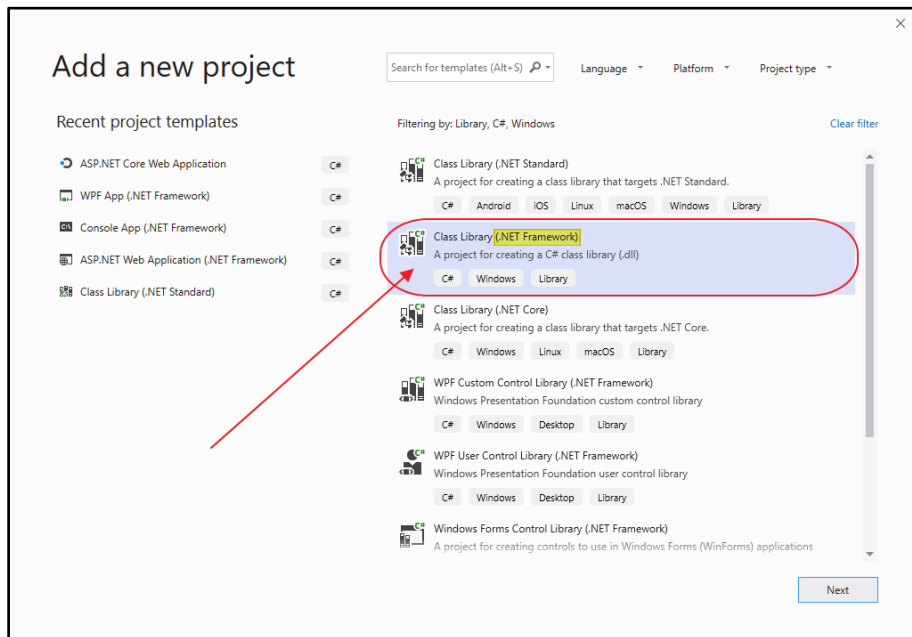
 `git clone` <https://github.com/howest-gp-prb/cu-KlassenEnObjecten-start>

### 3 CLASS LIBRARY

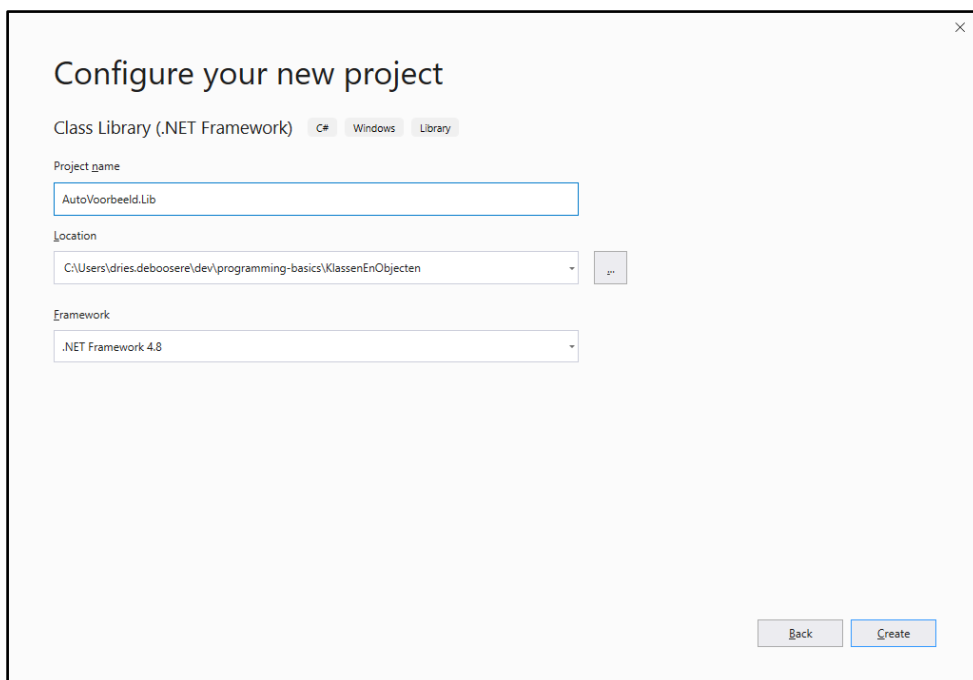
Wanneer je een klasse definieert, kan het de bedoeling zijn om deze klasse te gaan hergebruiken in een andere applicatie. Dit kan een andere WPF-applicatie, maar ook een WinForms, Console of ASP.net toepassing zijn.

Daarom maken we altijd gebruik van een Class Library; een klassenbibliotheek.

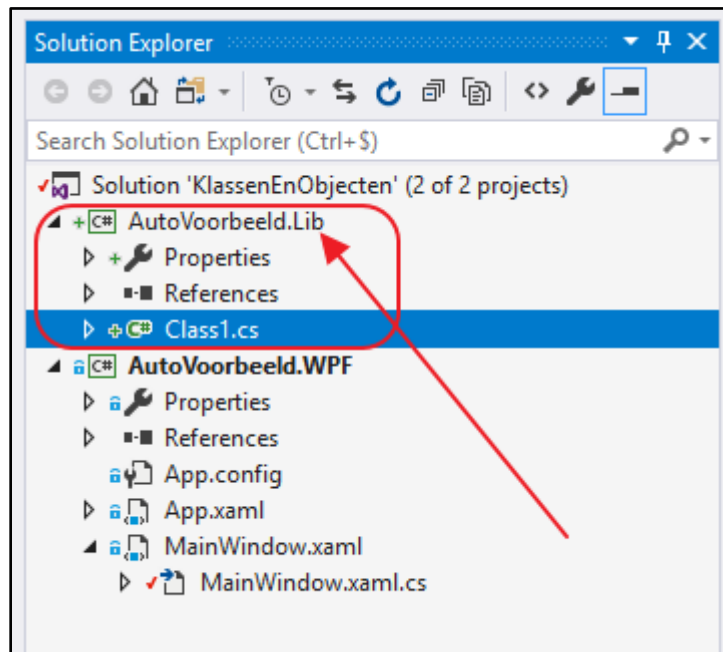
19. Rechtsklik op de **solution** en selecteer **Add > New Project** in het snelmenu en selecteer **Class Library (.NET Framework)** uit de templates.



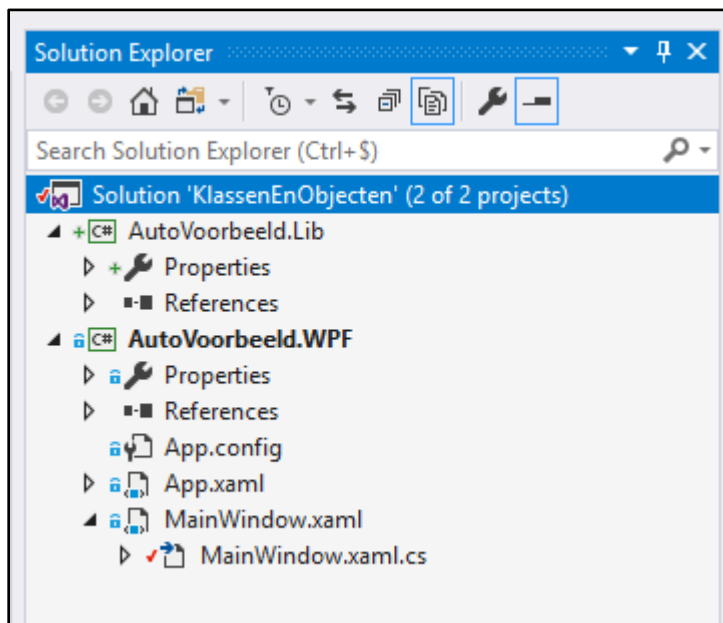
20. Geef de ClassLibrary de naam **AutoVoorbeeld.Lib** en klik op Create.



21. De Class Library met naam AutoVoorbeeld.Lib werd aangemaakt en is nu te vinden als een tweede project in je solution explorer.



22. Verwijder het codebestand `Class1.cs` dat automatisch wordt aangemaakt.



### Klassenbibliotheek of Class Library

Een klassenbibliotheek definieert typen en methoden die door een applicatie opgevraagd kunnen worden. Een class library die werd gemaakt met behulp van de .Net Standaard ondersteunt het .Net Standaard Framework, waardoor deze bibliotheek ingezet kan worden in elk .Net Platform dat de versie van de .Net Standaard ondersteunt.

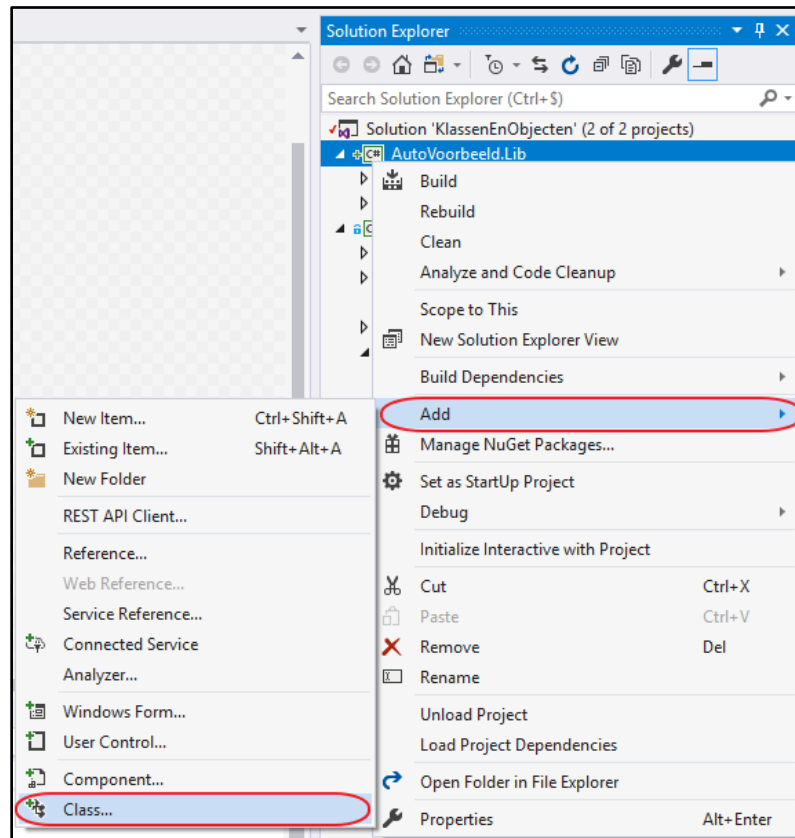
Na aanmaak van een klassenbibliotheek kun je bepalen of deze gedistribueerd wordt als een onderdeel van een applicatie of je ze wilt integreren als een gebundeld onderdeel met een of meerdere toepassingen.

## 4 GEBRUIK VAN KLASSEN

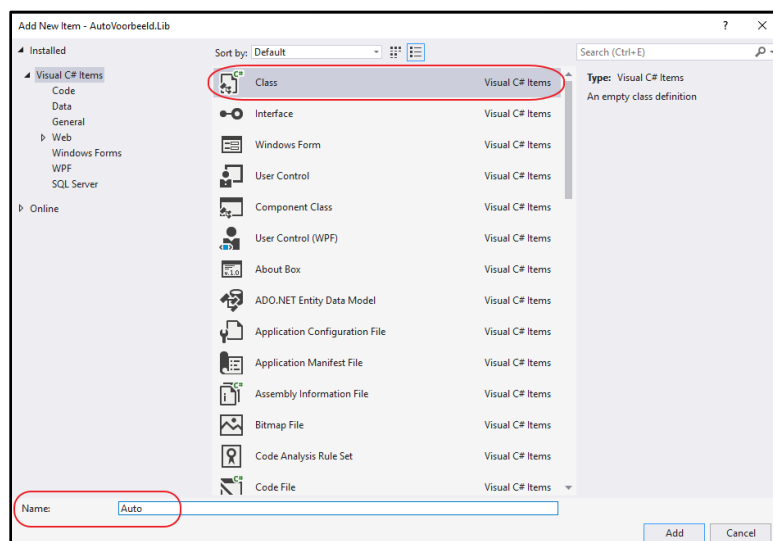
### 4.1 EEN KLASSE AANMAKEN

We gaan in onze aangemaakte klassenbibliotheek (Class Library) een nieuwe klasse aanmaken.

**23.** Klik rechts op je Class Library `AutoVoorbeeld.Lib` en kies **Add → Class...**



**24.** Geef de class de naam **Auto** en klik op **Add**.





### Naamgeving klassen

- De naam van de klasse geven we altijd in het **enkelvoud**. De klasse **Auto** is een blauwdruk van **een** auto.
- In de naam van een klasse proberen we cijfers zoveel mogelijk te vermijden.

25. We zien nu volgende code

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

namespace AutoVoorbeeld.Lib
{
    class Auto
    {
    }
}
```

De definitie van een klasse wordt gekenmerkt door het gereserveerde sleutelwoord **class** dat voorafgaat aan de naam van de klasse. In ons geval dus de naam Auto.

In het geval van onze applicatie worden van een auto volgende gegevens getoond:

- Merk
- Kleur
- Prijs

Dat zijn dus enkele eigenschappen van een auto die wij willen tonen in onze applicatie. We gaan van onze klasse Auto een blauwprint/sjabloon maken die kan dienen om de eigenschappen van één of meerdere concrete auto's bij te houden. Aangezien we al weten welke eigenschappen we willen tonen (merk, kleur en prijs), zullen we volgende code toevoegen in de body van onze klasse Auto:

```
namespace AutoVoorbeeld.Lib
{
    class Auto
    {
        string merk;
        string kleur;
        decimal prijs;
    }
}
```

Onze eerste klasse is nu aangemaakt!

## 4.2 OBJECTEN AANMAKEN

---

We weten nu wat een klasse is. Maar wat zijn dan objecten?

Een object is een instantie van een klasse. In andere woorden, een object is een mogelijke invulling van een klasse. We hebben nu net een klasse `Auto` aangemaakt met de verschillende eigenschappen (merk, kleur en prijs) maar dit wil nog niet zeggen dat we al een effectieve auto hebben toegevoegd in onze applicatie. We hebben nog nergens gezegd dat de auto bijvoorbeeld van het merk Volkswagen is, dat deze auto een rode kleur heeft en dat deze auto 27.000 euro kost.

Om dit te doen moeten we nog een **object** creëren van de klasse `Auto`.

In C# maken we gebruik van het sleutelwoord **new** om een object te maken van een klasse. Zie onderstaand voorbeeld:

```
Auto auto1 = new Auto();
```

- We starten dus met het schrijven van `Auto`, dit is de naam van de klasse en duidt ook het **type** van het object aan.
- Daarna geven we de naam van het object, in dit geval `auto1`. We kunnen zelf kiezen welke naam we aan objecten geven. We konden dit object evengoed `nieuweAuto` genoemd hebben in plaats van `auto1`.
- Na de object naam schrijven we het `=` teken, wat betekent dat we een waarde zullen toekennen aan ons `auto1` object.
- Na het `=` teken schrijven we het keyword **new**. Dit is een speciaal keyword om aan te geven aan de compiler dat er een nieuw object gecreëerd moet worden van de klasse rechts van het `new` keyword. Is ons geval dus `Auto`.
- `Auto()` betekent eigenlijk dat we de constructor aanroepen van de klasse `Auto`. Constructors zullen later in deze cursus nog besproken en uitgelegd worden.

Wanneer we een nieuw object van de klasse `Auto` willen maken dat `nieuweMooieAuto` moet heten, doen we dit als volgt:

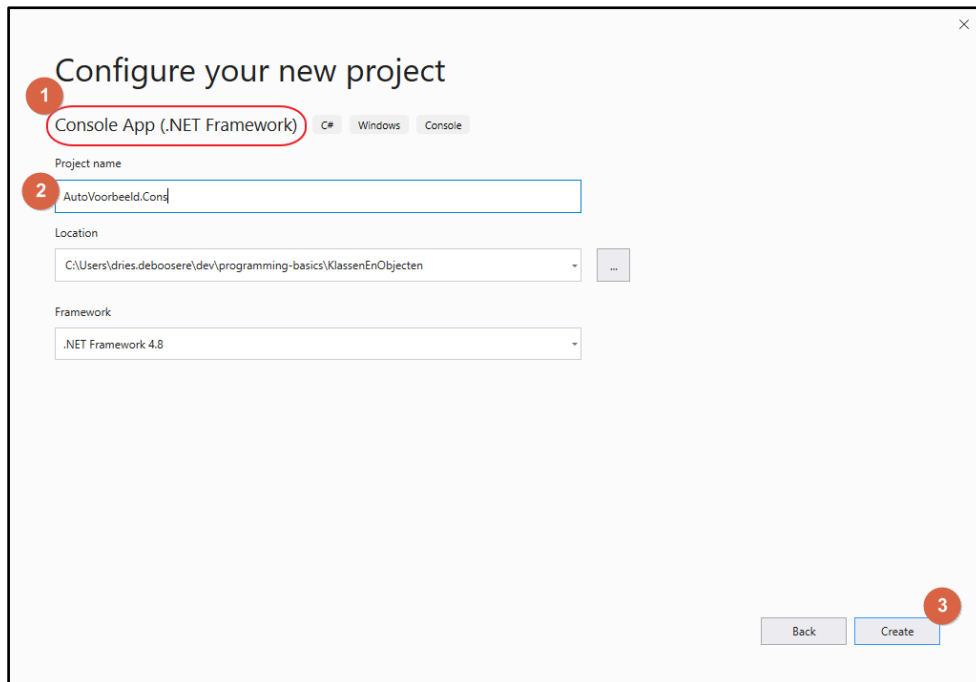
```
Auto nieuweMooieAuto = new Auto();
```

We hebben nu reeds twee objecten van het type `Auto`:

- `auto1`
- `nieuweMooieAuto`

## 4.3 PRAKTISCH VOORBEELD VAN KLASSEN EN OBJECTEN

Voeg een nieuwe Console App toe aan je bestaande solution “KlassenEnObjecten” en geef deze de naam **AutoVoorbeeld.Cons**



### 4.3.1 KLASSEN EN VELDEN PUBLIC MAKEN

55. We kijken even terug naar onze klasse Auto en zien volgende blauwdruk:

```
namespace AutoVoorbeeld.Lib
{
    class Auto
    {
        string merk;
        string kleur;
        decimal prijs;
    }
}
```

Onze klasse Auto is **private**, we kunnen deze klasse enkel gebruiken binnen zijn eigen namespace (AutoVoorbeeld.Lib). Dit geldt ook voor de velden (die behoren tot de members van de klasse), die kunnen enkel gebruikt worden door de klasse zelf.

Om dit op te lossen maken we de klasse Auto en zijn members (in dit geval fields/velden) **public**.



56. Maak de klasse Auto public en pas de code als volgt aan:

```
namespace AutoVoorbeeld.Lib
{
    public class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;
    }
}
```

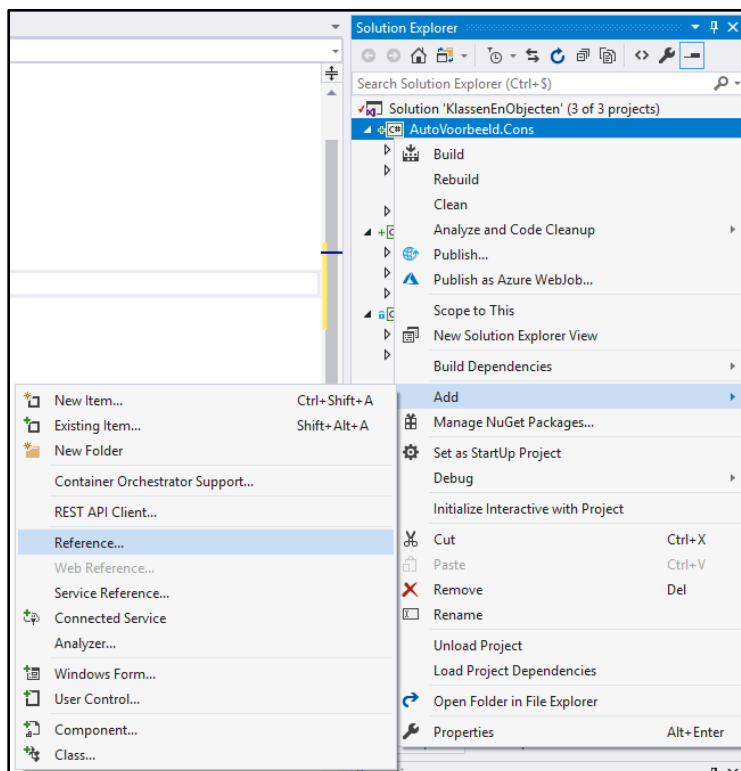
We maken in dit praktisch voorbeeld gebruik van twee projecten in onze solution, namelijk AutoVoorbeeld.Cons en AutoVoorbeeld.Lib waarin onze **public** klasse Auto aanwezig is.

In de console app willen we een object aanmaken van de klasse Auto en in de class library zit onze blueprint van de klasse Auto. Maar we hebben nog geen koppeling of referentie tussen de twee projecten gelegd. We zullen dit zo meteen doen.

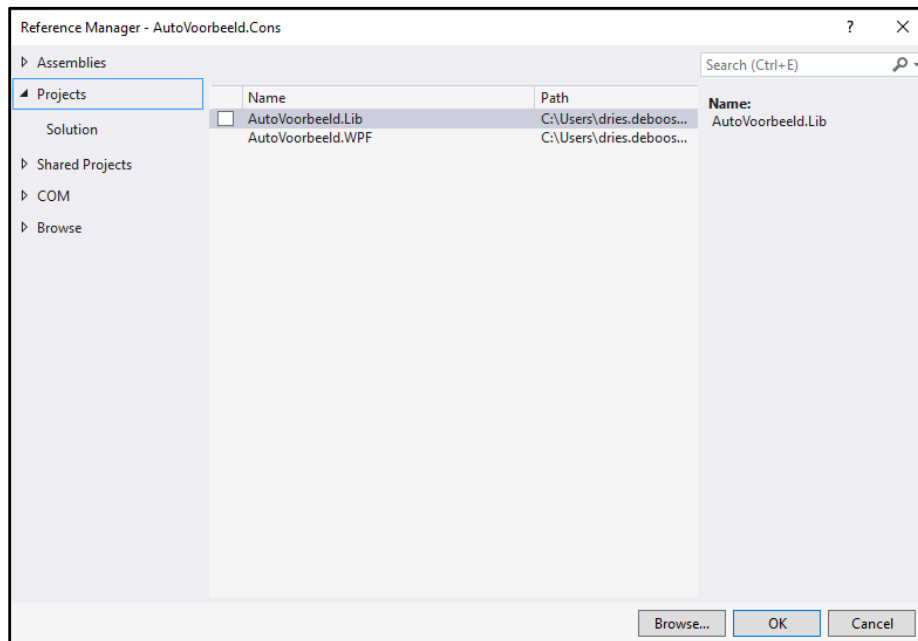
#### 4.3.2 REFERENTIE MAKEN NAAR EEN KLASSENBIBLIOTHEEK

We wensen dus vanuit onze console app gebruik te maken van de klasse Auto. We moeten dus een referentie maken vanuit onze console app naar onze class library die de klasse Auto bevat.

57. Rechtsklik op het project **AutoVoorbeeld.Cons** en klik op **Add → Reference...**

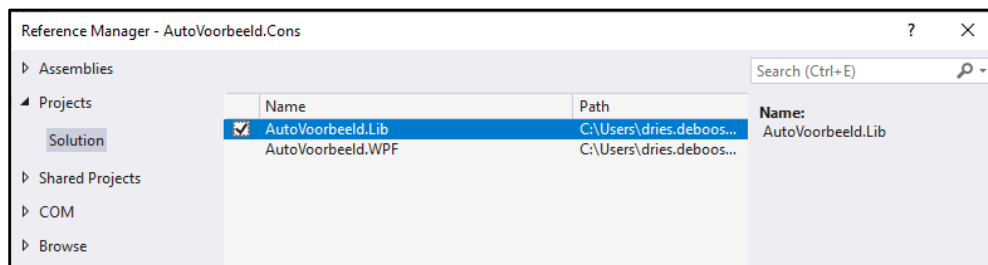


58. De Reference Manager opent en we zien onze twee andere projecten in deze solution in de lijst staan.

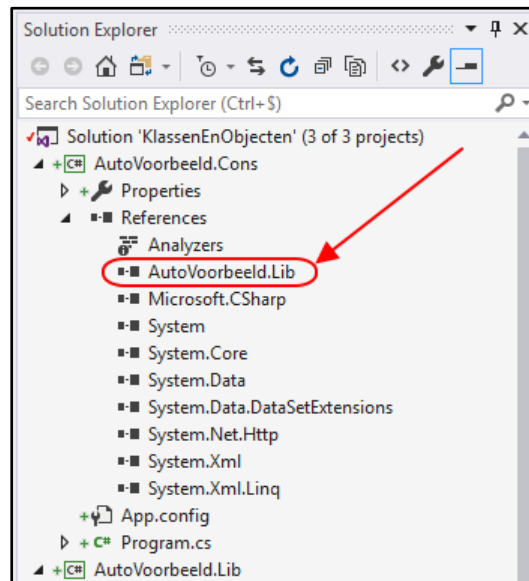


Mocht je de projecten niet zien staan, dan moet je in de linkerkolom kiezen voor Solution onder Projects. Hierdoor zal je alle projecten zien gekoppeld aan onze solution.

59. Vink de checkbox aan bij het project waar je een referentie naartoe wenst te leggen. In ons geval **AutoVoorbeeld.Lib**. Want we willen dat de klasse Auto gekend is in onze Console applicatie. Klik op OK.



60. Na het uitklappen van de References in AutoVoorbeeld.Cons zien we dat de namespace weldegelijk is toegevoegd:



61. Compileer nu je volledige solution (**CTRL + SHIFT + B**) of via **Build → Build Solution**.
62. Wensen we de klasse Auto te gebruiken in onze Console applicatie, dan kunnen we aan te geven dat we de namespace willen gebruiken. We gebruiken hiervoor een using statement:

```
using AutoVoorbeeld.Lib;
```

```
namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto nieuweAuto = new Auto();
        }
    }
}
```

Hiermee vermijden we dat we telkens de naam van de namespace moeten tikken als we iets uit AutoVoorbeeld.Lib gebruiken.

```
AutoVoorbeeld.Lib.Auto nieuweAuto = new AutoVoorbeeld.Lib.Auto();
```

Het is duidelijk dat deze manier van werken minder overzichtelijke code oplevert.

### 4.3.3 GEBRUIK VAN DE KLASSE AUTO

In de `Program.cs` klasse van ons Console App project gaan we een nieuw object van de klasse `Auto` gaan aanmaken. We doen dit in de body van de `Main` methode:

```
using AutoVoorbeeld.Lib;

namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto nieuweAuto;
            nieuweAuto = new Auto();
        }
    }
}
```

We ontleden even deze code

```
Auto nieuweAuto;
```

Een variabele met de naam **nieuweAuto** wordt **gedeclareerd** als object van het type **Auto**.

```
nieuweAuto = new Auto();
```

De variabele wordt **geïnitieerd** met het keyword **new**.

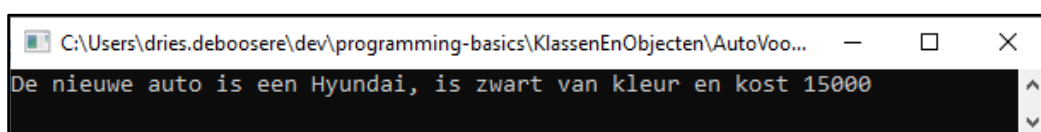
.NET zal nu een stukje geheugen reserveren voor alle members (fields en eventuele methoden) van het nieuwe object.

We kunnen nu alle public members van de klasse benaderen:

```
static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

    Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
    Console.ReadLine();
}
```



We kunnen nu zoveel objecten van het type Auto aanmaken als we willen:

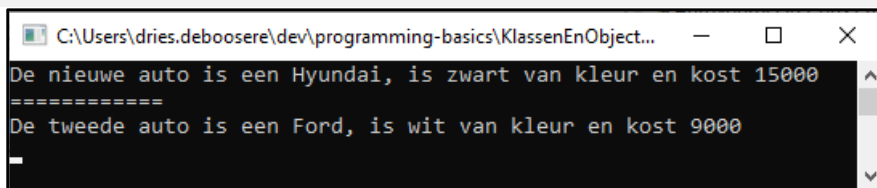
```
static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

    Auto tweedeAuto;
    tweedeAuto = new Auto();

    tweedeAuto.kleur = "wit";
    tweedeAuto.merk = "Ford";
    tweedeAuto.prijs = 9000M;

    Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur}
van kleur en kost {tweedeAuto.prijs}");
    Console.ReadLine();
}
```



## 4.4 CONSTRUCTORS

### 4.4.1 DEFAULT CONSTRUCTOR

Wanneer je een het sleutelwoord `new` gebruikt, geef je de opdracht om een nieuw object of nieuwe instantie te maken van een klasse.

Met een constructor in een klasse is het mogelijk velden van een nieuw object onmiddellijk te initialiseren. Wanneer je geen constructor voorziet in je klasse, dan genereert de compiler een standaardconstructor (**default constructor**) voor je klasse. Elk veld van de klasse moet nu eenmaal geïnitieerd worden.

- Volgend stukje code komt overeen met de standaardconstructor voor onze klasse `Auto`:

```
namespace AutoVoorbeeld.Lib
{
    public class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
        }
    }
}
```

Wanneer een nieuw `Auto`-object gemaakt wordt, worden de kleur en het merk op **null** (standaardwaarde bij declaratie van variabelen van datatype `string`) geplaatst en de prijs op **0** (standaardwaarde bij declaratie van getallen). Dit zal ook zo zijn wanneer we zelf geen (default) constructor voorzien.

### CONCLUSIE :

- Een constructor is een methode die wordt uitgevoerd wanneer een nieuwe instantie wordt aangemaakt (`new`)
- Een constructor heeft exact dezelfde naam als de klasse en geen returnwaarde
- In elke klasse is minstens 1 constructor aanwezig. Desnoods wordt die automatisch door de compiler aangemaakt.



#### Tip

Voor het aanmaken van een constructor kun je de code-snippet **ctor** gebruiken. Na ingeven van **ctor** en **2\* de tab-toets** indrukken zorgt voor een automatische aanmaak van een constructor.

## 4.4.2 CONSTRUCTORS OVERLOADEN

Gezien constructors eigenlijk speciale methoden zijn, kunnen we deze net zoals gewone methoden overladen. Een constructor heeft nooit een returnwaarde. Overloads van een constructor zullen dus enkel verschillen in de parameterlijst.

## 63. Voorzie volgende nieuwe constructors in de klasse Auto

```

namespace AutoVoorbeeld.Lib
{
    public class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
        }

        public Auto(string merk) // Constructor ontvangt 1 parameter
        {
            this.merk = merk;
        }

        public Auto(string merk, string kleur) // Constructor ontvangt 2 parameters
        {
            this.merk = merk;
            this.kleur = kleur;
        }

        public Auto(string merk, string kleur, decimal prijs) // Constructor ontvangt 3
parameters
        {
            this.merk = merk;
            this.kleur = kleur;
            this.prijs = prijs;
        }
    }
}

```

Opgepast, zoals in de tweede, derde en vierde constructor worden de argumenten ontvangen in een variabele met dezelfde naam als een bestaand field. Nu moet je het sleutelwoord `this` gebruiken om te verwijzen naar het field.

Wanneer we deze aanpassing hebben gedaan kunnen we in de codebehind van ons Console project bij het initialiseren gaan kiezen tussen de 4 constructors die we hebben aangemaakt:



64. We passen de code aan als volgt:

```
static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

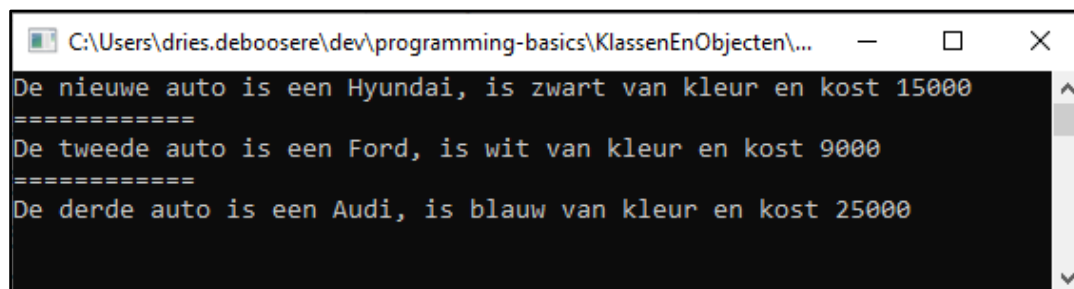
    Auto tweedeAuto;
    tweedeAuto = new Auto();

    tweedeAuto.kleur = "wit";
    tweedeAuto.merk = "Ford";
    tweedeAuto.prijs = 9000M;

    Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

    Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur}
van kleur en kost {tweedeAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van
kleur en kost {derdeAuto.prijs}");
    Console.ReadLine();
}
```

Met volgend resultaat:





## 4.5 GEBRUIK VAN STATIC

### 4.5.1 STATIC FIELDS

We kunnen binnen klassen velden `static` maken. Dit betekent dat dit static field data zal bevatten die **hetzelfde is voor elke instantie van de klasse**.

Een voorbeeld is het bijhouden van het aantal instanties dat voor een klasse gemaakt werd.

We doen dit als volgt:

65. Declareer een static field met de naam `aantalAutos`:

```
public class Auto
{
    public string merk;
    public string kleur;
    public decimal prijs;
    public static int aantalAutos;

    public Auto() // Default constructor
    {
        merk = null;
        kleur = null;
        prijs = 0;
        aantalAutos++;
    }

    public Auto(string merk):this() // Constructor ontvangt 1 parameter
    {
        this.merk = merk;
    }

    public Auto(string merk, string kleur):this() // Constructor ontvangt 2 parameters
    {
        this.merk = merk;
        this.kleur = kleur;
    }

    public Auto(string merk, string kleur, decimal prijs):this() // Constructor ontvangt
3 parameters
    {
        this.merk = merk;
        this.kleur = kleur;
        this.prijs = prijs;
    }
}
```

Een static field van het type `int` met de naam `aantalAutos` werd public gedeclareerd. In de parameterloze constructor zorgen we ervoor dat `aantalAutos` met 1 wordt opgehoogd als een `Auto` object wordt geïntanceerd.

We willen natuurlijk dat dit ook gebeurt wanneer we een andere constructor gebruiken. Hier zou je de opdracht `aantalAutos++` kunnen hernemen. Beter is echter de constructors aan elkaar te ketenen. (constructor-chaining).

Dit doen we door vanuit een constructor een andere constructor aan te roepen met behulp van `:this()` syntax. Waar dit vermeld staat zeggen we eigenlijk dat eerst de defaultconstructor (parameterloos) moet worden uitgevoerd vooraleer de constructor met deze verwijzing wordt uitgevoerd.

De werking ervan kunnen we demonstreren in onze console applicatie:

66. Pas de code in de console applicatie als volgt aan :

```
static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

    Auto tweedeAuto;
    tweedeAuto = new Auto();

    tweedeAuto.kleur = "wit";
    tweedeAuto.merk = "Ford";
    tweedeAuto.prijs = 9000M;

    Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

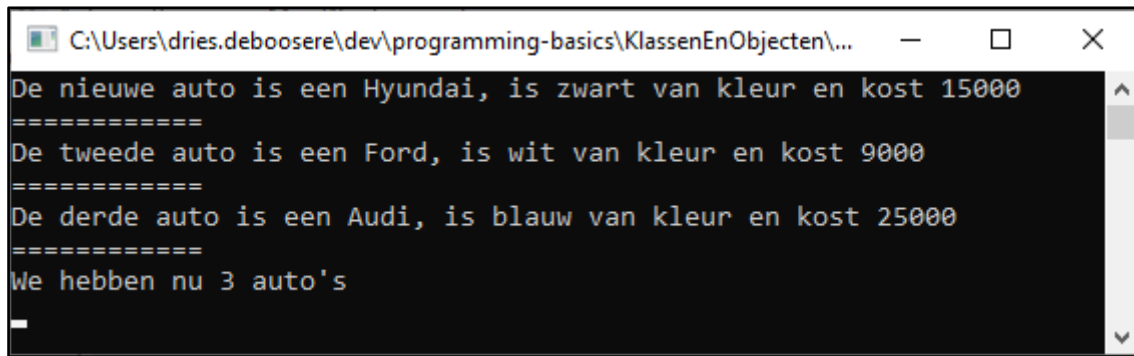
    Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur}
van kleur en kost {tweedeAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van
kleur en kost {derdeAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"We hebben nu {Auto.aantalAutos} auto's");
    Console.ReadLine();
}
```



#### Opgelet

- We verwijzen hier niet naar een Auto variabele (nieuweAuto, tweedeAuto of derdeAuto) maar naar de klasse Auto. Dit omdat het static field (aantalAutos) door alle Auto-objecten wordt gedeeld!

Met als resultaat:



```

C:\Users\dries.deboosere\dev\programming-basics\KlassenEnObjecten\...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's

```

#### 4.5.2 STATIC METHODS

Niet alleen fields kunnen static zijn, ook methoden binnen de klasse kunnen static zijn.

Willen we bijvoorbeeld het absolute prijsverschil weten tussen 2 Auto-objecten, dan zouden we dit zonder static methods op volgende manier kunnen oplossen:

67. Voeg volgende code toe in de klasse Auto:

```

namespace AutoVoorbeeld.Lib
{
    public class Auto
    {
        ...

        public decimal PrijsVerschil(Auto wagen)
        {
            return Math.Abs(this.prijs - wagen.prijs);
        }
    }
}

```

In de methode wordt als parameter een object met variabelenaam wagen meegegeven. We zoeken het verschil met de wagen die deze methode aanroept en retourneren een decimal-waarde.

68. We passen de code aan in de console applicatie:

```

static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

    Auto tweedeAuto;
    tweedeAuto = new Auto();
}

```

```

tweedeAuto.kleur = "wit";
tweedeAuto.merk = "Ford";
tweedeAuto.prijs = 9000M;

Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

decimal kostprijsVerschil = nieuweAuto.PrijsVerschil(derdeAuto);

Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
Console.WriteLine("=====");
Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur}
van kleur en kost {tweedeAuto.prijs}");
Console.WriteLine("=====");
Console.WriteLine($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van
kleur en kost {derdeAuto.prijs}");
Console.WriteLine("=====");
Console.WriteLine($"We hebben nu {Auto.aantalAutos} auto's");

Console.WriteLine("=====");
Console.WriteLine($"Prijsverschil tussen {nieuweAuto.prijs} en {derdeAuto.prijs} is
{kostprijsVerschil}");
Console.ReadLine();
}

```

Met volgend resultaat:

```

C:\Users\dries.deboosere\dev\programming-basics\KlassenEnObjecten...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's
=====
Prijsverschil tussen 15000 en 25000 is 10000

```

Een andere manier is gebruik maken van **static Methods**.

Die zorgen ervoor dat we niet steeds een instantie nodig hebben om vertrekkende vanuit deze instantie een methode aan te roepen.

69. Pas de code in de klasse Auto aan als volgt:

```

public class Auto
{
    public string merk;
    public string kleur;
    public decimal prijs;
    public static int aantalAutos;
}

```

```

...

public Auto(string merk, string kleur, decimal prijs):this() // Constructor ontvangt
3 parameters
{
    this.merk = merk;
    this.kleur = kleur;
    this.prijs = prijs;
}

public decimal PrijsVerschil(Auto wagen)
{
    return Math.Abs(this.prijs - wagen.prijs);
}

public static decimal PrijsVerschil(Auto wagen1, Auto wagen2)
{
    return Math.Abs(wagen1.prijs - wagen2.prijs);
}
}

```

70. Pas de code in de console applicatie aan als volgt:

```

static void Main(string[] args)
{
    Auto nieuweAuto;
    nieuweAuto = new Auto();

    nieuweAuto.kleur = "zwart";
    nieuweAuto.merk = "Hyundai";
    nieuweAuto.prijs = 15000M;

    Auto tweedeAuto;
    tweedeAuto = new Auto();

    tweedeAuto.kleur = "wit";
    tweedeAuto.merk = "Ford";
    tweedeAuto.prijs = 9000M;

    Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

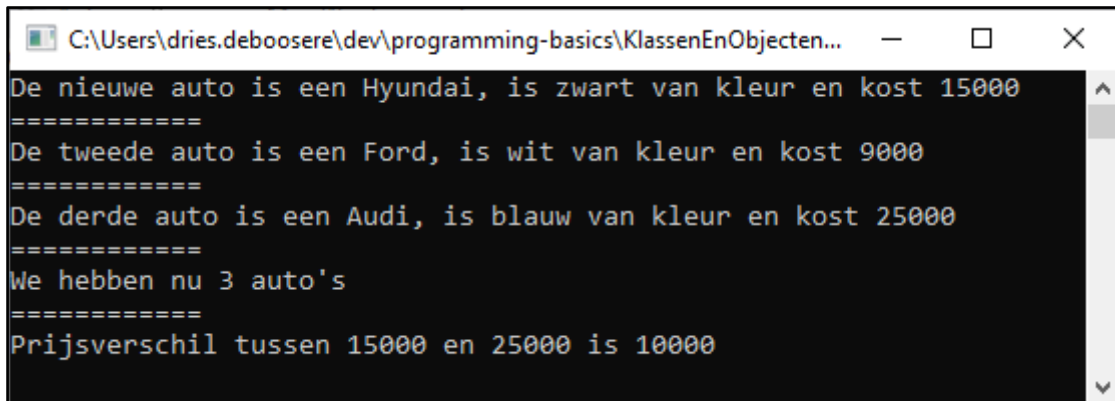
    decimal kostprijsVerschil = Auto.PrijsVerschil(nieuweAuto, derdeAuto);

    Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur}
van kleur en kost {nieuweAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur}
van kleur en kost {tweedeAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van
kleur en kost {derdeAuto.prijs}");
    Console.WriteLine("=====");
    Console.WriteLine($"We hebben nu {Auto.aantalAutos} auto's");
}

```

```
Console.WriteLine("=====");  
Console.WriteLine($"Prijsverschil tussen {nieuweAuto.prijs} en {derdeAuto.prijs} is  
{kostprijsVerschil}");  
Console.ReadLine();  
}
```

Met volgend resultaat:



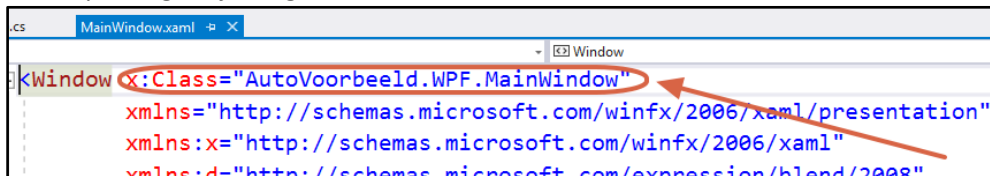
```
C:\Users\dries.deboosere\dev\programming-basics\KlassenEnObjecten...  
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000  
=====  
De tweede auto is een Ford, is wit van kleur en kost 9000  
=====  
De derde auto is een Audi, is blauw van kleur en kost 25000  
=====  
We hebben nu 3 auto's  
=====  
Prijsverschil tussen 15000 en 25000 is 10000
```

## 4.6 PARTIAL KLASSEN

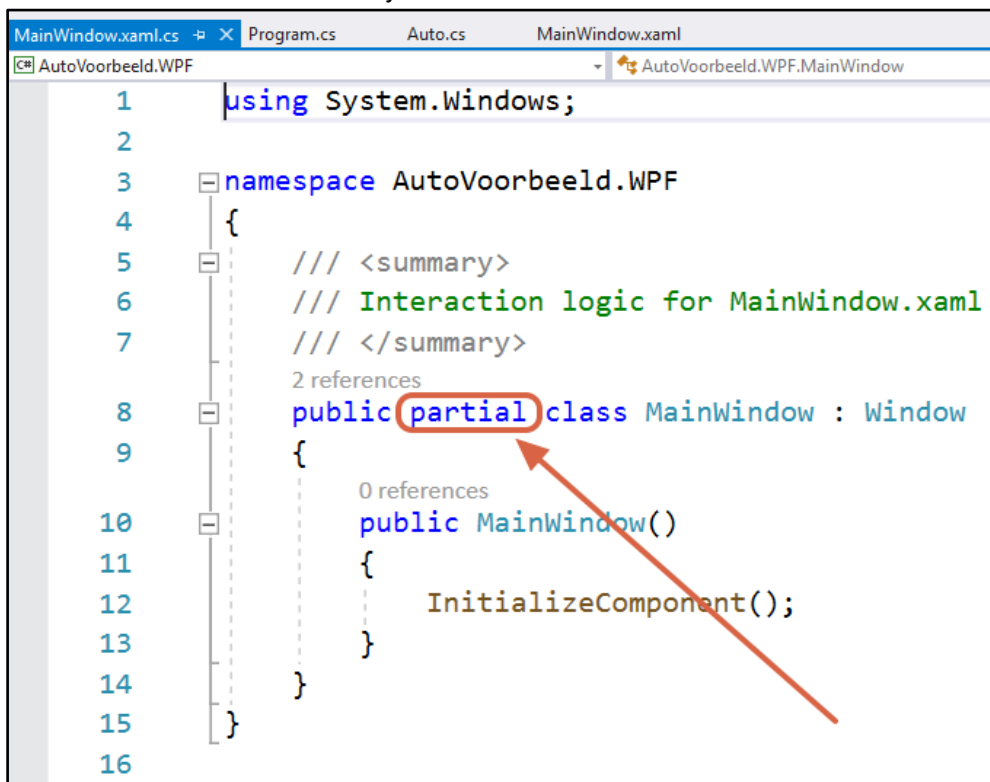
Een klasse kan veel methoden, velden, constructors en andere items bevatten. Een nuttige klasse kan op die manier behoorlijk omvangrijk worden. Met C# kan je de broncode voor een klasse uitsplitsen in meerdere beheersbare stukken code en zelfs over meerdere fysieke bestanden.

Dit principe wordt eigenlijk gebruikt in elke WPF toepassing of ASP.Net website die je maakt en heet **partial classes**.

- In je WPF toepassing zie je volgende code in de XAML-file:



- In de Code-behind van dit Window zie je:



Eigenlijk vormt de code die je voorziet in je codebehind samen met de code in de XAML-file de klasse **MainWindow**. De 2 files vormen dus bij compilatie **één klasse**.

Vandaar dat wijzigingen die je in de ene file uitvoert onmiddellijk hun weerslag hebben in de andere : een control (button, textbox, ...) die je op het window voorziet is direct bruikbaar in de codebehind-file.

We kunnen dit gaan toepassen op de eerder gemaakte klasse Auto:

71. Voeg een nieuwe klasse Auto2 toe aan de klassenbibliotheek (AutoVoorbeeld.Lib)

72. Wijzig de code als volgt:

a. Auto2 klasse:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
    }
}
```

b. Auto klasse:

```
using System;

namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;
        public static int aantalAutos;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
            aantalAutos++;
        }

        ...
    }
}
```

Je kunt nu alle methodes uit de **Auto**-klasse knippen en plakken in de **Auto2**- klasse.

Na hercompileren merk je geen verschil: beide klassen werken als 1 klasse!



## 4.7 ANONIEME KLASSE

Een anonieme klasse is een klasse die (zoals je wellicht vermoedt) geen naam heeft. In sommige toepassingen binnen .Net, zoals Linq wordt dit intensief gebruikt.

73. Voeg volgende code toe aan de console applicatie:

```
using AutoVoorbeeld.Lib;
using System;

namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

            decimal kostprijsVerschil = Auto.PrijsVerschil(nieuweAuto, derdeAuto);

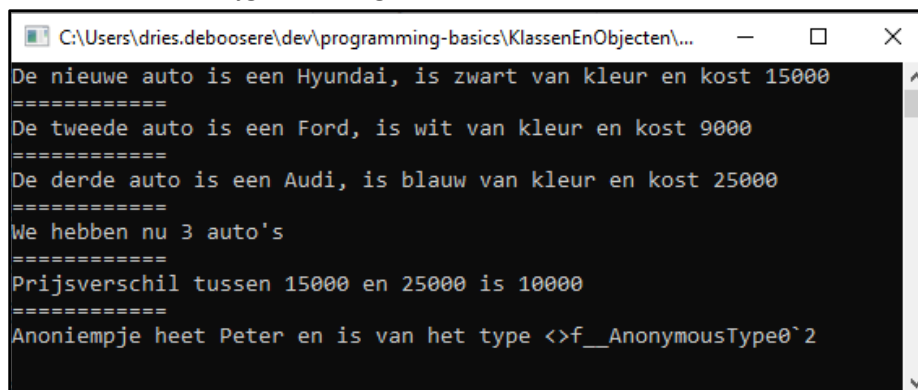
            var anoniempje = new { Naam = "Peter", Leeftijd = 40 };

            Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is
{nieuweAuto.kleur} van kleur en kost {nieuweAuto.prijs}");

            ...

            Console.WriteLine("=====");
            Console.WriteLine($"Anoniempje heet {anoniempje.Naam} en is van het type
{anoniempje.GetType().Name}");
            Console.ReadLine();
        }
    }
}
```

74. Na uitvoeren van de code krijgen we volgend resultaat:



```
C:\Users\dries.deboosere\dev\programming-basics\KlassenEnObjecten\...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's
=====
Prijsverschil tussen 15000 en 25000 is 10000
=====
Anoniempje heet Peter en is van het type <>f__AnonymousType0`2
```

Hier wordt een anonieme klasse gemaakt met 2 eigenschappen Naam en Leeftijd. In de Console zie je de door de compiler toegekende klassennaam.

## 4.8 BESLUITEN

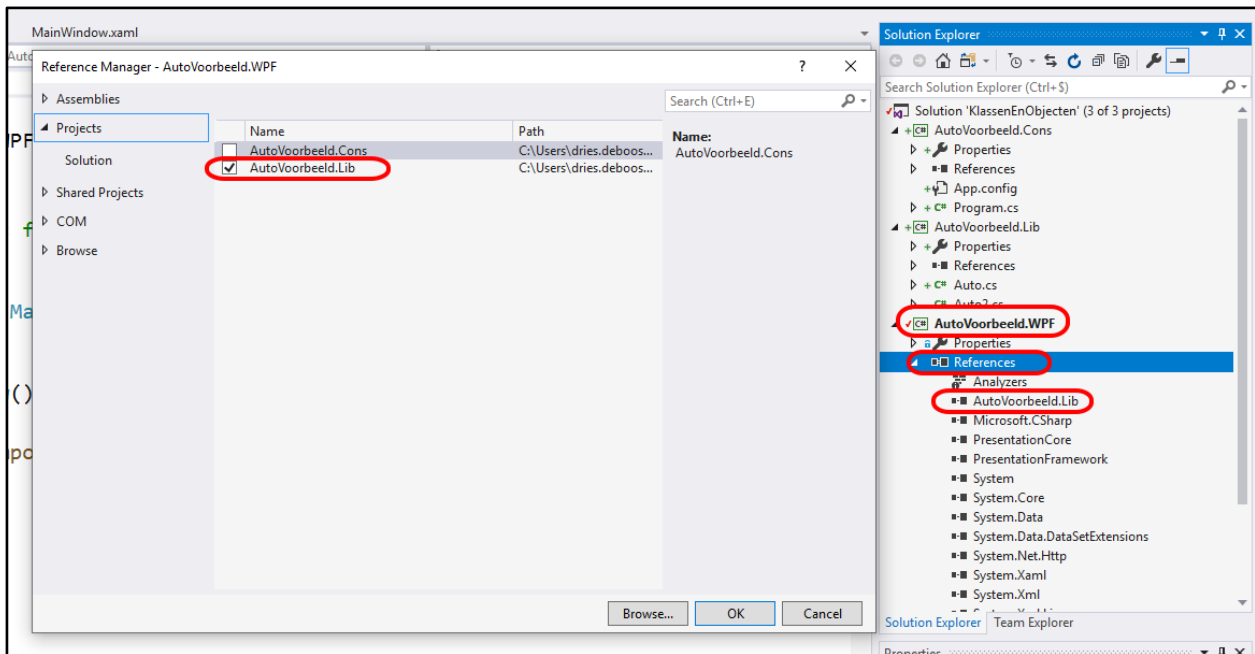
---

- Een klasse is een **blauwdruk** van een object. Een object is **data in het geheugen**, gestructureerd aan de hand van een klasse.
- Klassen omvatten **velden, eigenschappen** (zie volgend hoofdstuk) en **methodes**. Deze worden **members** genoemd.
- Een **constructor** wordt uitgevoerd tijdens de initialisatie van een object. Een **destructor** wordt uitgevoerd wanneer het object na gebruik uit het geheugen wordt verwijderd.
- **Overloading** is een techniek om dezelfde methode of constructor een andere signatuur te geven (nieuwe argumenten,...)
- **Static** members worden door elk object van een klasse gedeeld.

## 5 GEBRUIK VAN DE KLASSENBIBLIOTHEEK

We hebben reeds een klassenbibliotheek aangemaakt (AutoVoorbeeld.Lib) en hebben deze bibliotheek gebruikt in onze Console applicatie. We gaan deze klassenbibliotheek ook gebruiken voor onze WPF applicatie.

75. We maken een referentie vanuit de WPF applicatie naar de klassenbibliotheek zodat we de klasse Auto ook kunnen gebruiken in de WPF applicatie:



76. Voorzie een variabele die alle objecten van het type Auto zal bijhouden in de code-behind:

```
using AutoVoorbeeld.Lib;
using System.Collections.Generic;
using System.Windows;

namespace AutoVoorbeeld.WPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        List<Auto> lijstAutos;

        public MainWindow()
        {
            InitializeComponent();
            lijstAutos = new List<Auto>();
        }
    }
}
```

77. Wanneer er op de knop Voeg auto toe gedrukt wordt, dienen de textboxen uitgelezen te worden en wordt een nieuw object van het type Auto gemaakt en worden de uitgelezen waarden toegekend aan de fields van het object (of via de constructor met 3 parameters). De nieuwe auto wordt dan in de lijst van auto's toegevoegd en de listbox met auto's wordt opgevuld met alle auto's in de lijst:

```

using AutoVoorbeeld.Lib;
using System;
using System.Collections.Generic;
using System.Windows;

namespace AutoVoorbeeld.WPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        List<Auto> lijstAutos;

        public MainWindow()
        {
            InitializeComponent();
            lijstAutos = new List<Auto>();
        }

        private void BtnVoegAutoToe_Click(object sender, RoutedEventArgs e)
        {
            string kleurAuto = txtKleur.Text;
            string merkAuto = txtAutoMerk.Text;
            decimal prijsAuto = Convert.ToDecimal(txtPrijs.Text);

            Auto nieuweAuto = new Auto();
            nieuweAuto.kleur = kleurAuto;
            nieuweAuto.merk = merkAuto;
            nieuweAuto.prijs = prijsAuto;

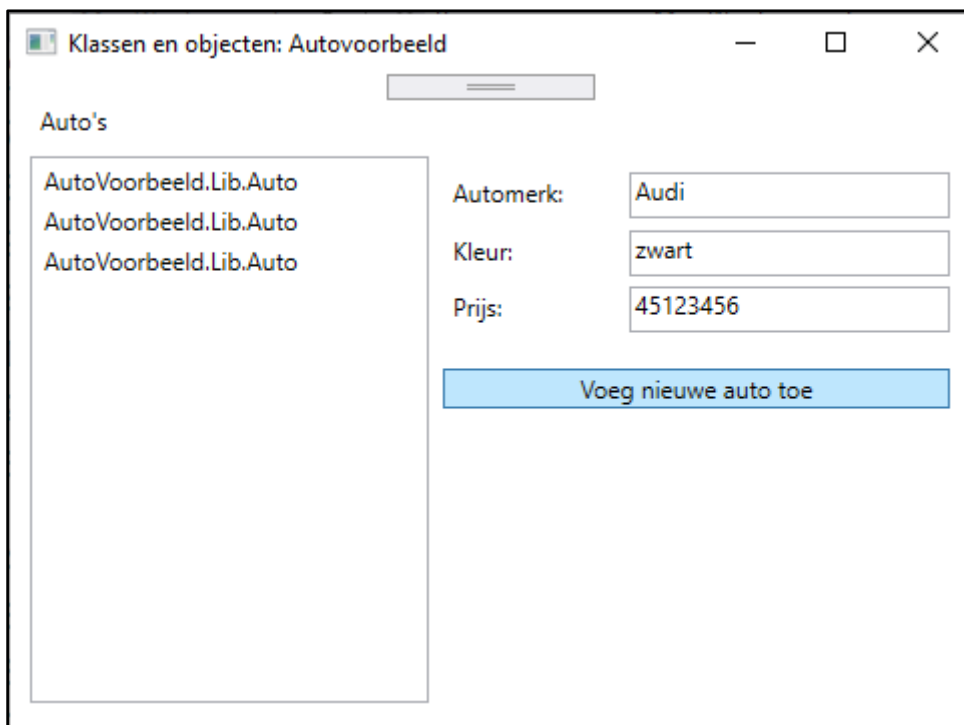
            // OF via constructor:
            //Auto nieuweAuto = new Auto(merkAuto, kleurAuto, prijsAuto);

            lijstAutos.Add(nieuweAuto);
            UpdateAutoLijst();
        }

        private void UpdateAutoLijst()
        {
            lstAutos.Items.Clear();
            foreach (Auto auto in lijstAutos)
            {
                lstAutos.Items.Add(auto);
            }
        }
    }
}

```

Wanneer we één of meerdere auto's toevoegen zien we volgend resultaat:



Op elk ListBoxItem wordt in de achtergrond de ToString()-methode toegepast.

Bij class-objecten is de standaard-return van deze methode de naam van namespace, gevolgd door een punt en de naam van de class. Dit is voor een gebruiker uiteraard weinig verhelderend.

Er bestaat in C# echter een mogelijkheid om de standaard-return tegen te houden en te vervangen door een andere return. Dit doen we via een override.

We maken een methode aan: `public override string ToString();`

- `public`: ze moet toegankelijk zijn van buiten de class
- `override`: ze **vervangt** de standaard return van een methode met dezelfde naam. Het is dus geen overload.
- `string`: de return is van het type string

In de methode stellen we dan de tekst samen die we willen tonen op basis van één of meerdere eigenschappen (fields). Deze tekst wordt dan geretourneerd.

78. Pas de code in de klasse Auto als volgt aan:

```
using System;

namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public decimal PrijsVerschil(Auto wagen)
        {
            return Math.Abs(this.prijs - wagen.prijs);
        }

        public static decimal PrijsVerschil(Auto wagen1, Auto wagen2)
```

```

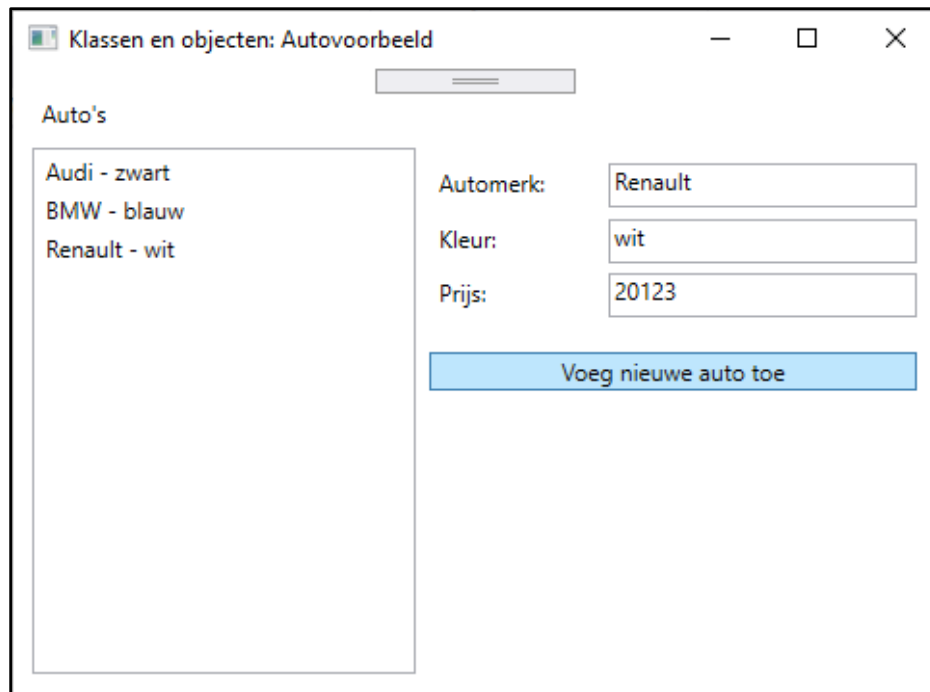
    {
        return Math.Abs(wagen1.prijs - wagen2.prijs);
    }

    public override string ToString()
    {
        string autoInfo = $"{merk} - {kleur}";
        return autoInfo;
    }
}

```

Via de override ToString() methode geven we aan dat we een object van het type Auto willen voorstellen als Merk – Kleur.

79. Voeg enkele auto's toe in de WPF-applicatie en je zult zien dat de listbox met auto's nu als volgt getoond wordt:



80. We zorgen ervoor dat de gegevens van de geselecteerde auto getoond worden in de textboxes. Creëer een event-handler op de listbox (SelectionChanged) die ervoor zorgt dat alle info in de textboxes getoond worden. Pas je code in de code-behind als volgt aan:

```
namespace AutoVoorbeeld.WPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        ...

        private void UpdateAutoLijst()
        {
            lstAutos.Items.Clear();
            foreach (Auto auto in lijstAutos)
            {
                lstAutos.Items.Add(auto.ToString());
            }
        }

        private void LstAutos_SelectionChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
        {
            Auto geselecteerdeAuto = (Auto)lstAutos.SelectedItem;

            txtAutoMerk.Text = geselecteerdeAuto.merk;
            txtKleur.Text = geselecteerdeAuto.kleur;
            txtPrijs.Text = geselecteerdeAuto.prijs.ToString();
        }
    }
}
```

Wanneer de selectie verandert in de listbox halen we het geselecteerde item op uit de listbox en **casten** we deze naar een object van het type Auto (geselecteerdeAuto). We hoeven hier geen gebruik te maken van het new keyword omdat geselecteerdeAuto reeds bestaat als object in het geheugen (dit object zit al in de lijst lijstAutos). We maken dus een referentie vanuit geselecteerdeAuto naar het effectieve object in het geheugen. We komen hier nog op terug in het hoofdstuk Value & references.

Via het object geselecteerdeAuto vullen we alle textboxes in aan de hand van de fields van het Auto-object.

Run je applicatie, voeg enkele auto's toe en selecteer dan de verschillende auto's in de listbox en je zal zien dat de textboxes ingevuld worden met de specificaties (fields) van de geselecteerde auto.



#### Code repository

De volledige broncode van dit hoofdstuk is te vinden op

`git clone` <https://github.com/howest-gp-prb/cu-KlassenEnObjecten-volledig>





# **HOOFDSTUK 10**

## **PROPERTIES**



## INHOUDSOPGAVE

<b>1</b>	<b>INLEIDING</b>	<b>197</b>
<b>1.1</b>	<b>Wat zijn properties</b>	<b>197</b>
<b>1.2</b>	<b>Praktisch voorbeeld</b>	<b>197</b>
<b>2</b>	<b>DATA AFSCHERMEN MET EEN METHODE</b>	<b>199</b>
<b>2.1</b>	<b>Conclusie</b>	<b>202</b>
<b>3</b>	<b>DATA AFSCHERMEN MET EEN PROPERTY</b>	<b>203</b>
<b>3.1</b>	<b>Conclusie</b>	<b>204</b>
<b>4</b>	<b>AUTOMATISCHE PROPERTIES</b>	<b>205</b>
<b>4.1</b>	<b>Nullable properties</b>	<b>205</b>
<b>4.2</b>	<b>Auto-property initializer</b>	<b>206</b>
<b>4.3</b>	<b>Read-only properties</b>	<b>206</b>
<b>5</b>	<b>AANPASSEN VAN ONZE VOORBEELD APPLICATIE</b>	<b>208</b>



## 1 INLEIDING

In het vorige hoofdstuk heb je kennisgemaakt met instantievariabelen. Om deze instantievariabelen binnen de klasse te bereiken, heb je gebruik gemaakt van de **access modifier** `public`, waarna deze toegankelijk werd voor de Code-behind klasse.

Dit werd trouwens ook gebruikt voor de klasse (binnen een andere namespace). Daardoor worden niet alleen de instantievariabelen maar ook de methoden, die `public` werden ingesteld, toegankelijk vanuit je code behind of een andere klasse.

### 1.1 WAT ZIJN PROPERTIES

Properties of eigenschappen stellen de ontwikkelaar in staat om data af te schermen van oneigenlijk gebruik of verkeerd gebruik: **data encapsulation**.

### 1.2 PRAKTISCH VOORBEELD



#### Code repository

Gebruik de solution waarmee we het hoofdstuk “Klassen en objecten” hebben beëindigd of clone onderstaand git project.

`git clone https://github.com/howest-gp-prb/cu-properties-start.git`

81. Open de klasse `Auto` in het bestand `Auto.cs` in de `AutoVoorbeeld.Lib` class library.

82. Voeg volgende code toe:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;
        public static int aantalAutos;
        public int topSnelheid;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
            aantalAutos++;
        }
        ...
    }
}
```

83. Open het Program.cs van de console applicatie project AutoVoorbeeld.Cons

84. Voeg volgende code toe:

```
namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto nieuweAuto;
            nieuweAuto = new Auto();

            nieuweAuto.kleur = "zwart";
            nieuweAuto.merk = "Hyundai";
            nieuweAuto.prijs = 15000M;
            nieuweAuto.topSnelheid = -190;

            Auto tweedeAuto;

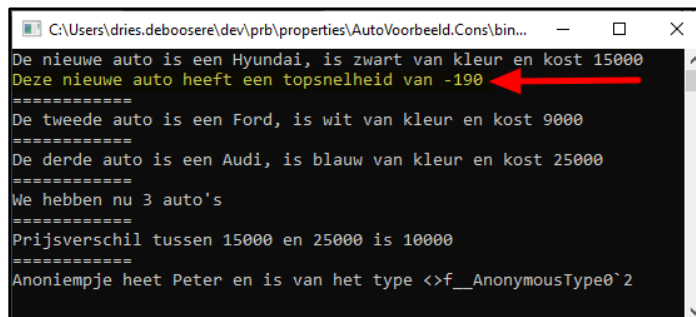
            ...

            var anoniempje = new { Naam = "Peter", Leeftijd = 40 };

            Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is
{nieuweAuto.kleur} van kleur en kost {nieuweAuto.prijs}");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van
{nieuweAuto.topSnelheid}");
            Console.WriteLine("=====");

            ...
        }
    }
}
```

Met volgend resultaat:



```
C:\Users\dries.deboosere\dev\prb\properties\AutoVoorbeeld.Cons\bin...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
Deze nieuwe auto heeft een topsnelheid van -190
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's
=====
Prijsverschil tussen 15000 en 25000 is 10000
=====
Anoniempje heet Peter en is van het type <>f__AnonymousType0`2
```

Uiteraard willen we niet werken met een negatieve topsnelheid. In deze context kunnen we negatieve waarden dan ook niet toestaan. Door het gebruik van data encapsulation kunnen we dit voorkomen.

Er bestaan verschillende manieren om onze data af te schermen of ervoor te zorgen dat onze data niet verkeerd gebruikt wordt. We bespreken enkele mogelijkheden, de één al wat omslachtiger dan de andere.

## 2 DATA AFSCHERMEN MET EEN METHODE

Je kan in de klasse `Auto` het field `topSnelheid` afschermen door het `private` te maken en een public methode `SetTopSnelheid` te voorzien die voor de controle zorgt.

85. Voorzie volgende wijzigingen in de klasse `Auto`:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;
        public static int aantalAutos;
        private int topSnelheid;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
            aantalAutos++;
        }

        ...

        public void SetTopSnelheid(int snelheid)
        {
            if (snelheid > 0) topSnelheid = snelheid;
            else topSnelheid = 0;
        }
    }
}
```

Wanneer de opgegeven topsnelheid verkregen via de methode `SetTopSnelheid` kleiner is dan 0, stellen we die in op 0. Een andere mogelijkheid zou kunnen zijn dat we een `Exception` gooien met de melding dat de snelheid niet negatief mag zijn.

Wanneer we na deze wijzigingen onze solution opnieuw builden krijgen we compileerfouten:

	Code	Description	Project	File
✖	CS0122	'Auto.topSnelheid' is inaccessible due to its protection level	AutoVoorbeeld.C...	Program.cs
✖	CS0122	'Auto.topSnelheid' is inaccessible due to its protection level	AutoVoorbeeld.C...	Program.cs

Aangezien we het field `topSnelheid` in de klasse `Auto` hebben gewijzigd van `public` naar `private` is dit field niet meer toegankelijk vanuit onze console applicatie:



```

D:\references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Auto nieuweAuto;
        nieuweAuto = new Auto();

        nieuweAuto.kleur = "zwart";
        nieuweAuto.merk = "Hyundai";
        nieuweAuto.prijs = 15000M;
        nieuweAuto.topSnelheid = -190;

        Auto tweedeAuto;
        tweedeAuto = new Auto();

        tweedeAuto.kleur = "wit";
        tweedeAuto.merk = "Ford";
        tweedeAuto.prijs = 9000M;

        Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

        decimal kostprijsVerschil = Auto.PrijsVerschil(nieuweAuto, derdeAuto);

        var anoniempje = new { Naam = "Peter", Leeftijd = 40 };

        Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur} van kleur en kost {nieuweAuto.prijs}");
        Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van {nieuweAuto.topSnelheid}");
        Console.WriteLine("=====");
        Console.WriteLine($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur} van kleur en kost {tweedeAuto.prijs}");
        Console.WriteLine("=====");
        Console.WriteLine($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van kleur en kost {derdeAuto.prijs}");
        Console.WriteLine("=====");
        Console.WriteLine($"We hebben nu {Auto.aantalAutos} auto's");

        Console.WriteLine("=====");
        Console.WriteLine($"Prijsverschil tussen {nieuweAuto.prijs} en {derdeAuto.prijs} is {kostprijsVerschil}");

        Console.WriteLine("=====");
        Console.WriteLine($"Anoniempje heet {anoniempje.Naam} en is van het type {anoniempje.GetType().Name}");
        Console.ReadLine();
    }
}

```

86. Om in onze console applicatie de `topSnelheid` in te stellen van een `auto` object passen we volgende code aan:

```

namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

            nieuweAuto.topSnelheid = -190; // Verander deze code door onderstaande code
            nieuweAuto.SetTopSnelheid(-190);

            ...
        }
    }
}

```

We stellen nu de `topSnelheid` in via de methode `SetTopSnelheid`.

We hebben nog altijd één compileerfout; omdat het field `topSnelheid` `private` is kunnen we dit ook niet meer opvragen.



87. Voorzie volgende methode in de klasse Auto:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        ...

        public void SetTopSnelheid(int snelheid)
        {
            if (snelheid > 0) topSnelheid = snelheid;
            else topSnelheid = 0;
        }

        public int GetTopSnelheid()
        {
            return topSnelheid;
        }
    }
}
```

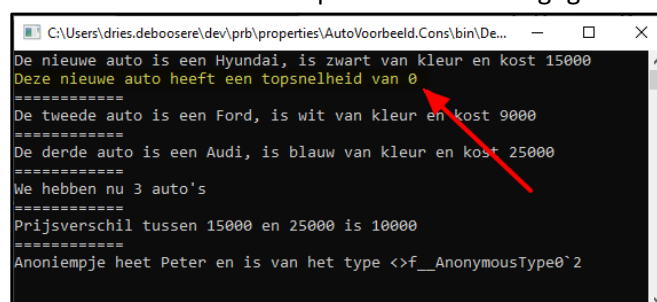
88. Wijzig in Program.cs volgende code:

```
namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

            Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is
{nieuweAuto.kleur} van kleur en kost {nieuweAuto.prijs}");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van
{nieuweAuto.GetTopSnelheid()}");
            Console.WriteLine("=====");

            ...
        }
    }
}
```

89. Run de console applicatie en we zien dat de topsnelheid nu weergegeven wordt als 0:



```
C:\Users\dries.deboosere\dev\prb\properties\AutoVoorbeeld.Cons\bin\De...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
Deze nieuwe auto heeft een topsnelheid van 0
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's
=====
Prijsverschil tussen 15000 en 25000 is 10000
=====
Anoniempje heet Peter en is van het type <>f__AnonymousType0`2
```

## 2.1 CONCLUSIE

---

- Werken met public fields kan leiden tot een verkeerd gebruik van data
- Werken met private velden en publieke getter- en settermethoden kan dit probleem oplossen
- De oplossing met deze methoden is **omslachtig**:
  - Je moet voor elk field twee methoden toevoegen
  - Om een waarde op te vragen moet je steeds een methodeaanroep uitvoeren:
    - `nieuweAuto.topSnelheid` was eigenlijk toch gemakkelijker dan `nieuweAuto.GetTopSnelheid()`
  - Om een waarde aan te passen, moet je ook een methodeaanroep doen:
    - `nieuweAuto.topSnelheid = -190;`  
werd  
`nieuweAuto.SetTopSnelheid(-190);`

### 3 DATA AFSCHERMEN MET EEN PROPERTY

Een ideale oplossing voor dit probleem vormen properties of eigenschappen: je hebt de afscherming zoals met methoden, maar naar de buitenwereld toe kan je de data manipuleren zoals een field.

90. Vervang de methoden `GetTopSnelheid` en `SetTopSnelheid` in de klasse `Auto` door onderstaande code:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public string merk;
        public string kleur;
        public decimal prijs;
        public static int aantalAutos;
        private int topSnelheid;

        public Auto() // Default constructor
        {
            merk = null;
            kleur = null;
            prijs = 0;
            aantalAutos++;
        }

        ...

        public int TopSnelheid
        {
            get { return topSnelheid; }
            set
            {
                if (value > 0) topSnelheid = value;
                else topSnelheid = 0;
            }
        }
    }
}
```

Hier hebben we de eigenschap/property `TopSnelheid` gedeclareerd.

Het onderdeel **get** geeft aan wat er gebeurt wanneer de property wordt gelezen. Laat je dit onderdeel weg, dan kan de eigenschap niet uitgelezen worden buiten de eigen klasse; de property is **write-only**.

Het onderdeel **set** geeft aan hoe de eigenschap wordt ingesteld en veranderd. In dit onderdeel kan je gebruik maken van het sleutelwoord **value** om te verwijzen naar de waarde waarop de property ingesteld wordt. Laat je dit onderdeel weg, dan kan de eigenschap niet gebruikt worden om een waarde te veranderen; de property is **read-only**.

De naam van een field begint met een **kleine letter**, die van properties met een **hoofdletter**.

Opgepast, het field `topSnelheid` is hier nog steeds nodig, de property `TopSnelheid` gaat dit field enkel afschermen tegen oneigenlijk gebruik.

91. Pas de code als volgt aan in Program.cs om de builderrors aan te pakken:

```
namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

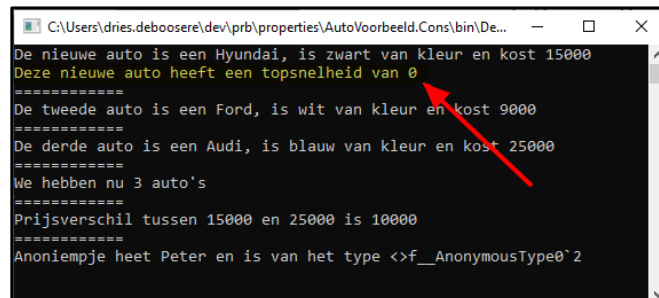
            nieuweAuto.TopSnelheid = -190;

            ...

            Console.WriteLine($"De nieuwe auto is een {nieuweAuto.merk}, is
{nieuweAuto.kleur} van kleur en kost {nieuweAuto.prijs}");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van
{nieuweAuto.TopSnelheid}");
            Console.WriteLine("=====");

            ...
        }
    }
}
```

92. Run de console applicatie en we zien dat de topsnelheid ook nu weergegeven wordt als 0:



```
C:\Users\dries.deboosere\dev\prb\properties\AutoVoorbeeld.Cons\bin\De...
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
Deze nieuwe auto heeft een topsnelheid van 0
=====
De tweede auto is een Ford, is wit van kleur en kost 9000
=====
De derde auto is een Audi, is blauw van kleur en kost 25000
=====
We hebben nu 3 auto's
=====
Prijsverschil tussen 15000 en 25000 is 10000
=====
Anoniempje heet Peter en is van het type <>f__AnonymousType0`2
```



#### Tip

Voor het aanmaken van een dergelijke property kun je de **code-snippet propfull** gebruiken. Na ingeven van **propfull** en **2 \* de tab-toets** indrukken, zorgt die voor een automatische aanmaak van een property met get en set accessor en private field.

## 3.1 CONCLUSIE

We hebben nu dezelfde functionaliteit, maar dan zonder dat we extra methodes moeten aanmaken!

## 4 AUTOMATISCHE PROPERTIES

Vanaf nu het gebruik je best properties die private fields afschermen. Het is dan ook een **goede attitude** om, zelfs wanneer je voorlopig geen extra controle of andere bijkomstige code bij een field nodig hebt, elk field toch private te maken en een public property te voorzien:

```
private int waarde;

public int Waarde
{
    get { return waarde; }
    set { waarde = value; }
}
```

Bij het gebruik van een private field en public property zou men in feite bovenstaande code moeten schrijven voor elke property van een object. Een beetje omslachtig als er geen controles of manipulaties moeten gebeuren met de property. Sinds C# 3.0 kunnen we bovenstaande code inkorten door volgende code:

```
public int Waarde { get; set; }
```

Dit codefragment maakt de property *Waarde* aan. Intern wordt deze verkorte schrijfwijze vertaald tot het voorbeeld erboven: het field en de get en set-implementatie van de property worden dus automatisch voorzien. Het field *waarde* is dus niet zichtbaar.

De property kan even eenvoudig worden aangemaakt als een field, maar je beschikt over een property die je later kan uitbreiden.

Starten met een field, en later overschakelen naar een property kan omslachtiger zijn.


### 4.1 NULLABLE PROPERTIES

Soms wensen we NULL toe te kennen aan een property. Bij value-type (int, decimal, DateTime, ...) properties is dit standaard niet mogelijk. Neem onderstaand voorbeeld:

```
public class Kalender
{
    public DateTime Datum { get; set; }

    public Kalender() // Default constructor
    {
        Datum = null;
    }
}
```

We krijgen een error:

	Code	Description
	CS0037	Cannot convert null to 'DateTime' because it is a non-nullable value type

Wensen we de property Datum toch nullable te maken, dan kunnen we dit doen door na het datatype (DateTime) een **vraagteken (?)** te plaatsen:

```
public DateTime? Datum { get; set; }
```

Een ander voorbeeld:

```
public int? Aantal { get; set; }
```

## 4.2 AUTO-PROPERTY INITIALIZER

Soms wensen we een property reeds een standaard waarde te geven. Bekijk onderstaand voorbeeld:

```
public class PepperoniPizza
{
    public decimal ExtraPrijs { get; set; }

    public PepperoniPizza()
    {
        ExtraPrijs = 0.25m;
    }
}
```

In de constructor van deze klasse initialiseren we de property *ExtraPrijs* met een waarde van 0,25. Dus elk geïntanceerd object van deze klasse zal een property *ExtraPrijs* hebben met een waarde van 0,25.

Sinds C# 6.0 kunnen we dit in deze verkorte versie schrijven:

```
public class PepperoniPizza
{
    public decimal ExtraPrijs { get; set; } = 0.25m;
}
```

We kennen automatisch een waarde toe aan de property. We hebben in dit geval de constructor hiervoor dus niet meer nodig.

## 4.3 READ-ONLY PROPERTIES

Soms willen we dat een gebruiker enkel een property kan uitlezen. De waarde van de property kan ingesteld worden via de constructor, auto-property initializer, ...

Als we de property willen afschermen tegen wijzigingen van buitenaf, dus door het gebruik maken van de set van de property, kunnen we de set gewoon weghalen uit de automatische property.

Zie onderstaand voorbeeld:

```
public class PepperoniPizza
{
    public decimal ExtraPrijs { get; } = 0.25m;
}
```

De property `ExtraPrijs` kan dus enkel uitgelezen worden met de `get`. Aangezien de `set` hier niet aanwezig is kan de waarde van deze property niet van buiten deze klasse aangepast worden.

## 5 AANPASSEN VAN ONZE VOORBEELD APPLICATIE

Aangezien we beter kunnen werken met automatische properties zullen we de andere public fields (merk, kleur, prijs en aantalAutos) van de klasse Auto ook aanpassen.

93. Wijzig en vervang volgende code in de klasse Auto in het bestand Auto.cs:

```
namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public string merk;
        public string Merk { get; set; }
        public string kleur;
        public string Kleur { get; set; }
        public decimal prijs;
        public decimal Prijs { get; set; }
        public static int aantalAutos;
        public static int AantalAutos { get; set; }

        private int topSnelheid;

        public Auto() // Default constructor
        {
            merk = null;
            Merk = null;
            kleur = null;
            Kleur = null;
            prijs = 0;
            Prijs = 0;
            aantalAutos++;
            AantalAutos++;
        }

        public Auto(string merk) : this() // Constructor ontvangt 1 parameter
        {
            this.merk = merk;
            Merk = merk;
        }

        public Auto(string merk, string kleur) : this() // Constructor ontvangt 2 parameters
        {
            this.merk = merk;
            Merk = merk;
            this.kleur = kleur;
            Kleur = kleur;
        }

        public Auto(string merk, string kleur, decimal prijs) : this() // Constructor ontvangt
3 parameters
        {
            this.merk = merk;
            Merk = merk;
            this.kleur = kleur;
            Kleur = kleur;
            this.prijs = prijs;
            Prijs = prijs;
        }
    }
}
```



```

    }

    public int TopSnelheid
    {
        get { return topSnelheid; }
        set
        {
            if (value > 0) topSnelheid = value;
            else topSnelheid = 0;
        }
    }
}

```

Aangezien we nog steeds een negatieve topsnelheid willen opvangen, kunnen we de property `TopSnelheid` niet in zijn verkorte versie schrijven (`public int TopSnelheid { get; set; }`). Deze property heeft nog steeds een zichtbaar private field nodig.

94. Nog steeds in de klasse `Auto` maar dan in het bestand `Auto2.cs` (zie vorige hoofdstuk Klassen en objecten bij het onderdeel partial klassen) maken we nog gebruik van de fields in plaats van de properties. Pas de code aan, zodat er vanaf nu gebruik gemaakt wordt van de properties i.p.v. de fields:

```

namespace AutoVoorbeeld.Lib
{
    public partial class Auto
    {
        public decimal PrijsVerschil(Auto wagen)
        {
            return Math.Abs(this.prijs - wagen.prijs);
            return Math.Abs(Prijs - wagen.Prijs);
        }

        public static decimal PrijsVerschil(Auto wagen1, Auto wagen2)
        {
            return Math.Abs(wagen1.prijs - wagen2.prijs);
            return Math.Abs(wagen1.Prijs - wagen2.Prijs);
        }

        public override string ToString()
        {
            string autoInfo = $"{merk} - {kleur}";
            string autoInfo = $"{Merk} - {Kleur}";
            return autoInfo;
        }
    }
}

```

95. Uiteraard dienen we ook onze console applicatie aan te passen. Wijzig onderstaande code in Program.cs:

```
namespace AutoVoorbeeld.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto nieuweAuto;
            nieuweAuto = new Auto();

            nieuweAuto.kleur = "zwart";
            nieuweAuto.Kleur = "zwart";
            nieuweAuto.merk = "Hyundai";
            nieuweAuto.Merk = "Hyundai";
            nieuweAuto.prijs = 15000M;
            nieuweAuto.Prijs = 15000M;
            nieuweAuto.TopSnelheid = -190;

            Auto tweedeAuto;
            tweedeAuto = new Auto();

            tweedeAuto.kleur = "wit";
            tweedeAuto.Kleur = "wit";
            tweedeAuto.merk = "Ford";
            tweedeAuto.Merk = "Ford";
            tweedeAuto.prijs = 9000M;
            tweedeAuto.Prijs = 9000M;

            Auto derdeAuto = new Auto("Audi", "blauw", 25000M);

            decimal kostprijsVerschil = Auto.PrijsVerschil(nieuweAuto, derdeAuto);

            var anoniempje = new { Naam = "Peter", Leeftijd = 40 };

            Console.WriteLine(($"De nieuwe auto is een {nieuweAuto.merk}, is {nieuweAuto.kleur} van kleur en kost
{nieuweAuto.prijs}");
            Console.WriteLine($"De nieuwe auto is een {nieuweAuto.Merk}, is {nieuweAuto.Kleur} van kleur en kost
{nieuweAuto.Prijs}");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van {nieuweAuto.TopSnelheid}");
            Console.WriteLine("=====");
            Console.WriteLine(($"De tweede auto is een {tweedeAuto.merk}, is {tweedeAuto.kleur} van kleur en kost
{tweedeAuto.prijs}");
            Console.WriteLine($"De tweede auto is een {tweedeAuto.Merk}, is {tweedeAuto.Kleur} van kleur en kost
{tweedeAuto.Prijs}");
            Console.WriteLine("=====");
            Console.WriteLine(($"De derde auto is een {derdeAuto.merk}, is {derdeAuto.kleur} van kleur en kost
{derdeAuto.prijs}");
            Console.WriteLine($"De derde auto is een {derdeAuto.Merk}, is {derdeAuto.Kleur} van kleur en kost
{derdeAuto.Prijs}");
            Console.WriteLine("=====");
            Console.WriteLine(($"We hebben nu {Auto.aantalAutos} auto's");
            Console.WriteLine($"We hebben nu {Auto.AantalAutos} auto's");

            Console.WriteLine("=====");
            Console.WriteLine(($"Prijsverschil tussen {nieuweAuto.prijs} en {derdeAuto.prijs} is {kostprijsVerschil}");
            Console.WriteLine($"Prijsverschil tussen {nieuweAuto.Prijs} en {derdeAuto.Prijs} is {kostprijsVerschil}");

            Console.WriteLine("=====");
            Console.WriteLine($"Anoniempje heet {anoniempje.Naam} en is van het type {anoniempje.GetType().Name}");
            Console.ReadLine();
        }
    }
}
```

96. Ook in onze WPF applicatie dienen we aanpassingen te maken:

```
namespace AutoVoorbeeld.WPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        List<Auto> lijstAutos;

        public MainWindow()
        {
            InitializeComponent();
        }
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            lijstAutos = new List<Auto>();
        }

        private void BtnVoegAutoToe_Click(object sender, RoutedEventArgs e)
        {
            string kleurAuto = txtKleur.Text;
            string merkAuto = txtAutoMerk.Text;
            decimal prijsAuto = Convert.ToDecimal(txtPrijs.Text);

            Auto nieuweAuto = new Auto();
            nieuweAuto.Kleur = kleurAuto;
            nieuweAuto.Merk = merkAuto;
            nieuweAuto.Prijs = prijsAuto;

            // OF via constructor:
            //Auto nieuweAuto = new Auto(merkAuto, kleurAuto, prijsAuto);

            lijstAutos.Add(nieuweAuto);
            UpdateAutoLijst();
        }

        private void UpdateAutoLijst()
        {
            lstAutos.Items.Clear();
            foreach (Auto auto in lijstAutos)
            {
                lstAutos.Items.Add(auto);
            }
        }

        private void LstAutos_SelectionChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
        {
            Auto geselecteerdeAuto = (Auto)lstAutos.SelectedItem;

            txtAutoMerk.Text = geselecteerdeAuto.Merk;
            txtKleur.Text = geselecteerdeAuto.Kleur;
            txtPrijs.Text = geselecteerdeAuto.Prijs.ToString();
        }
    }
}
```

**Code repository**

De volledige broncode van dit hoofdstuk is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-properties-volledig.git>

# **HOOFDSTUK 11**

## **VALUE EN REFERENCE**



## INHOUDSOPGAVE

<b>1</b>	<b>INLEIDING</b>	<b>217</b>
1.1	De voorbeeldapplicatie	217
1.2	Value versus Reference	218
1.3	Het computergeheugen	219
1.1.1	Stack geheugen	219
1.1.2	Heap geheugen	219
<b>2</b>	<b>Value Types</b>	<b>221</b>
2.1	Primitieve types	221
2.2	Enums	221
2.3	Structs	222
<b>3</b>	<b>Reference Types</b>	<b>223</b>
3.1	Klassen	223
3.2	String	224
<b>4</b>	<b>De waarde “null” en Nullable Types</b>	<b>226</b>
4.1	null	226
4.2	Nullable types	226
<b>5</b>	<b>Ref en Out parameters</b>	<b>228</b>
5.1	Parameters doorgeven als referentie	228
5.2	Uitgaande parameters	229
<b>6</b>	<b>Boxing en Unboxing</b>	<b>230</b>
6.1	De klasse System.Object	230
6.2	Boxing	231
6.3	Unboxing	232
6.4	Veilig casten met as en is	232
6.5	Het nut van Boxing en Unboxing	234





## 1 INLEIDING

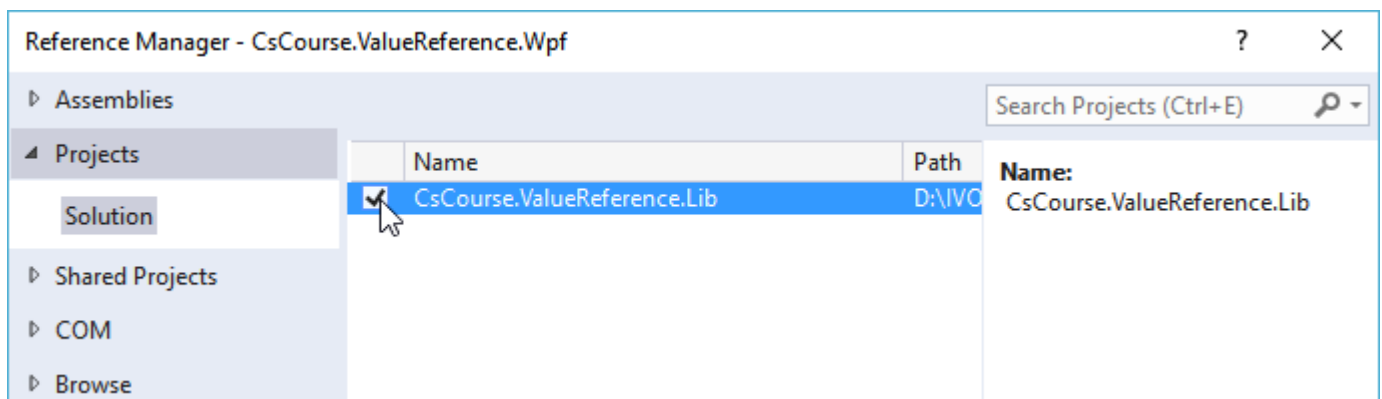
In C# bestaan er heel wat types. Zo bestaan er primitieve datatypes, zoals `int`, `byte`, `bool`, enz. die standaard ondersteund worden door de compiler. Er bestaan ook complexere types zoals een `struct` of een `class`. Het merendeel van de types die teruggevonden worden in het .NET Framework zijn complexe types. Denk daarbij aan klassen zoals `TextBox` en `Window`, of aan structures zoals `DateTime`.

De CLR behandelt de datatypes op twee manieren: als een **value** type of als een **reference** type. Dit hoofdstuk schept duidelijkheid over wat hiermee bedoeld wordt.

### 1.1 DE VOORBEELDAPPLICATIE

Om te starten, begin je met het maken van een nieuw project.

97. Maak een WPF applicatie genaamd `ValueReference.Wpf` in een solution genaamd `ValueReference`.
98. Voeg een **Class Library (.NET Framework)** project toe aan de solution genaamd **`ValueReference.Lib`**.
99. Verwijder het codebestand **`Class1.cs`** uit het Class Library project.
100. Leg een referentie vanuit het WPF project naar het Class Library project via een rechtsklik op het WPF project en **Add Reference**.



101. Voeg een nieuwe enum toe aan de Class Library genaamd `CarType`.

```
public enum CarType
{
    Unknown = 0,
    HatchBack = 1,
    Sedan = 2,
    MUV = 3, //multiple use vehicle
    SUV = 4, //sports use vehicle
    Crossover = 5,
}
```

102. Voeg een nieuwe struct toe aan de Class Library genaamd Vector3d. Dit type houdt de coördinaten van iets bij in een 3D wereld.

```
public struct Vector3d
{
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }
}
```

103. Voeg een nieuwe class toe aan de Class Library genaamd Car.

```
public class Car
{
    public Car(string model, CarType type, decimal currentSpeed)
    {
        this.Model = model;
        this.Type = type;
        this.Speed = currentSpeed;
    }

    public string Model { get; set; }
    public decimal Speed { get; set; }
    public Vector3d Position { get; set; }
    public CarType Type { get; set; }

    public override string ToString()
    {
        return $"{Model} driving at {Speed}";
    }
}
```



#### Code repository

De volledige broncode van dit onderdeel is te vinden op



git clone <https://github.com/howest-gp-prb/cu-value-reference-all.git>

#### Startsituatie:



git clone <https://github.com/howest-gp-prb/cu-value-reference-start.git>

## 1.2 VALUE VERSUS REFERENCE

Wanneer je een variabele, veld of property declareert, dan wordt er een geheugenplaats gereserveerd voor de waarde ervan. Die geheugenplaats kan twee zaken bevatten:

- de eigenlijke waarde van die variabele, of
- een verwijzing naar die waarde.

In dat laatste geval staat de eigenlijke waarde op een andere plaats (het heap-geheugen). De verwijzing is het geheugenadres van de plaats waar die waarde zich bevindt, een **referentie** dus. Er zijn dus twee

stukken geheugen nodig: een klein blokje voor het bewaren van het *geheugenadres* en een ander, groter blok voor het bewaren van de *eigenlijke waarde*.

Datatypes die op deze manier geheugen krijgen, noemen we **reference** types.

De **object** en **string** types zijn de meest voorkomende reference types. Met keywords **class** en **interface** kan je zelf reference type bouwen.

Een **value** type variabele heeft slechts één geheugenplaats nodig. Dat stukje geheugen bevat de eigenlijke waarde ervan. De grootte van zo'n value type variabele is afhankelijk van het gekozen datatype. Een `int` is bijvoorbeeld 32 bits groot, een `long` is 64 bits. Een `decimal` is 128 bits, en een `bool` heeft slechts 1 bit nodig.

Onderstaande tabel toont alle value types en hun grootte in C#.

<code>bool</code>	booleaanse waarde, 1 bit	<code>long</code>	geheel getal, signed, 64 bits
<code>byte</code>	geheel getal, unsigned, 8 bits	<code>sbyte</code>	geheel getal, signed, 8 bits
<code>char</code>	geheel getal, unsigned, 16 bits	<code>short</code>	geheel getal, signed, 16 bits
<code>decimal</code>	decimaal getal, 128 bits	<code>struct</code>	structure
<code>double</code>	decimaal getal, 64 bits	<code>uint</code>	geheel getal, unsigned, 32 bits
<code>enum</code>	enumeratie	<code>ulong</code>	geheel getal, unsigned, 64 bits
<code>float</code>	decimaal getal, 32 bits	<code>ushort</code>	geheel getal, unsigned, 16 bits
<code>int</code>	geheel getal, signed, 32 bits		

## 1.3 HET COMPUTERGEHEUGEN

Computers gebruiken het RAM geheugen om programma's uit te voeren en de gebruikte data bij te houden. Om het verschil tussen value types en reference types in te zien is het interessant om te begrijpen hoe het computergeheugen voor je applicatie georganiseerd is.

Besturingssystemen en runtimes zoals de CLR - Common Language Runtime - voor .NET gaan het beschikbare geheugen vaak opdelen in twee stukken die op een aparte manier benaderd worden: de **stack** en de **heap**.

### 1.1.1 STACK GEHEUGEN

**Stack-geheugen** kan je zien als een op elkaar gestapelde (stack) verzameling dozen. Wanneer een methode wordt aangeroepen, wordt elke parameter in een doos gestopt die bovenop de stapel wordt gelegd. Wanneer een methode uitgevoerd is, worden de dozen weer van de stapel genomen.

Het stack geheugen bevat:

- ✓ Een verwijzing naar de **laatst opgeroepen methode** en de **parameters** daarvan.
- ✓ De **lokale variabelen** die in de **huidige** methode zijn gedeclareerd.
- ✓ De **return** waarde van de huidige methode, indien deze beëindigd wordt.
- ✓ Verwijzingen naar waarden op de heap.

### 1.1.2 HEAP GEHEUGEN

- **Heap-geheugen** kan je zien als een hoop (heap) dozen die niet op een stapel liggen, maar in een grote ruimte door elkaar liggen. Elke doos heeft een label dat aangeeft of ze nog in gebruik is.

Wanneer een reference type object wordt aangemaakt met het **new** keyword, dan zoekt de runtime een vrije doos en gebruikt die voor de waarden van dat object. De referentie naar die doos wordt bewaard in een lokale variabele die op de **stack** wordt bewaard.

- Als de laatste referentie naar het object verdwijnt, dan markeert de runtime de doos als zijnde ongebruikt, en kan die vernietigd worden. In de .NET CLR wordt dat gedaan door de zogenaamde "Garbage Collector".

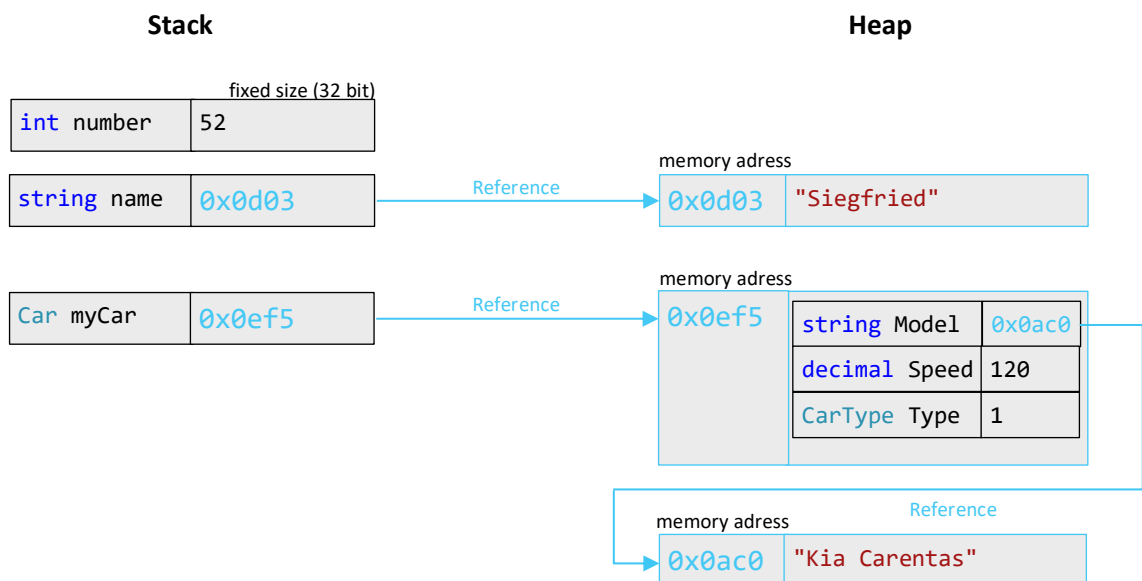
Het heap geheugen bevat dus:

- ✓ De effectieve waarden (instanties) van een **reference**-type object, zoals een class of een string.
- ✓ Lokale variabelen van de oproepende methode(s) die **niet meer toegankelijk zijn** in de huidige methode.

Stel dat je de volgende methode hebt:

```
public void DoStuff()
{
    int number = 52;
    string name = "Siegfried";
    Car myCar = new Car(model:"Kia Carentas", currentSpeed:120, type: CarType.HatchBack);
}
```

Vlak voor dat deze methode eindigt kunnen we het heap en het stack geheugen visueel voorstellen als volgt:



De variabelen *number*, *name* en *myCar* zijn lokale variabelen en bevinden zich op de **stack**. Alle andere variabelen zijn properties van referentie types en bevinden zich daarom op de **heap**.

Alle lokale variabelen en properties met datatype *int*, *decimal* en *CarType* (enum) zijn **value** types.

Alle lokale variabelen en properties met datatype *string* en *Car* zijn **reference types** en bevatten slechts een verwijzing naar het **geheugenadres** (geschreven als hexadecimale waarde) dat de eigenlijke waarde bevat.

## 2 VALUE TYPES

Primitieve types zoals *int*, *float*, *double* en *char*, maar ook *structs* en *enums* zijn **value types**. Wanneer je een dergelijke variabele declareert, zal de compiler code genereren die een stukje geheugen reserveert. Dat geheugenblokje is groot genoeg om de **effectieve waarde** van de variabele van een dergelijke datatype te bevatten. De declaratie van een *int* variabele zorgt ervoor dat 4 bytes of 32 bits aan geheugen worden gereserveerd.

```
int number; //reserveer 32 bits aan geheugen voor een int
```

Een statement dat ervoor zorgt dat bijvoorbeeld de waarde 42 aan de variabele wordt toegekend, zorgt ervoor dat de **waarde** van de variabele (42) effectief op die geheugenplaats wordt weggeschreven.

```
number = 42; //vul de geheugenplaats met de waarde 42 ( 0000 ... 0000 0010 1010)
```

### 2.1 PRIMITIEVE TYPES

De verschillen tussen value en reference types zijn van groot belang bij het programmeren.

104. Voorzie in de MainWindow een Button `btnCopyValueType` die bij het klikken volgende code uitvoert:

```
int i = 42;
int j = i;
i++;
Console.WriteLine($"i = {i}\nj = {j}");
```

De laatste instructie stuurt zijn uitvoer naar het Output venster van Visual Studio dat zichtbaar gemaakt kan worden via **View → Output**.

De output ziet er als volgt uit:

```
i = 43
j = 42
```

Merk op dat `i` werd opgehoogd tot 43, de waarde van `j` bleef 42.

Dit betekent dat bij de toekenning `j = i` enkel de waarde van `i` in de geheugenplaats van `j` werd **gekopieerd**. Verder is er geen blijvende koppeling tussen `i` en `j`.

### 2.2 ENUMS

Ook enumeraties zijn value types. Dat komt omdat een enumeratie steeds gebaseerd is op één van de primitieven *byte*, *int*, *Long*, *uint* of *uLong*.

105. Voorzie in de MainWindow een Button `btnCopyValueTypeEnum` die bij het klikken volgende code uitvoert:

```
CarType type1 = CarType.Crossover;
CarType type2 = type1;
type1 = CarType.HatchBack;
Debug.Print($"Type1 = {type1}, Type2 = {type2}");
```

De output ziet er als volgt uit:

```
Type1 = HatchBack, Type2 = Crossover
```

Tijdens de toekenning `type2` en `type1` werd ook hier enkel de waarde van `type1` naar `type2` **gekopieerd**. Wanneer variabele `type1` vervolgens gewijzigd wordt naar `HatchBack`, blijft de waarde van `type2` ongewijzigd, nl. `Crossover`.

## 2.3 STRUCTS

Een struct kan je best omschrijven als een sterk vereenvoudigde `class`. Structs encapsuleren eveneens properties en velden, maar zijn in tegenstelling tot klassen **value types**.

Ze worden heel vaak gebruikt om kleine hoeveelheden gegevens te bewaren. Omdat ze zich op de stack bevinden, kunnen ze snel bevraagd worden. Dat maakt ze populair bij bijvoorbeeld 3D games, waar prestaties zeer belangrijk zijn. Te veel of te grote structs zijn dan weer nefast voor het geheugengebruik.

106. Voorzie in de MainWindow een Button `btnCopyValueTypeStruct` die bij het klikken volgende code uitvoert:

```
Vector3d vector1 = new Vector3d(2, 3, 4);
Vector3d vector2 = vector1;
vector1.X++;
vector1.Y++;
vector1.Z++;
Debug.Print("Vector1: X = {0}, Y = {1}, Z = {2}", vector1.X, vector1.Y, vector1.Z);
Debug.Print("Vector2: X = {0}, Y = {1}, Z = {2}", vector2.X, vector2.Y, vector2.Z);
```

De output ziet er als volgt uit:

```
Vector1: X = 3, Y = 4, Z = 5
Vector2: X = 2, Y = 3, Z = 4
```

De eerste variabele `vector1` wordt gedeclareerd en dat levert een geheugenplaats op de stack die groot genoeg is om dat volledige object te kunnen bevatten (namelijk 3x een integer; voor X, Y en Z). Vervolgens wordt `vector1` geïntanceerd met het `new` keyword. Hierdoor worden de waarden van X, Y en Z ingesteld.

Een nieuw variabele, `vector2` wordt gedeclareerd en het value type object `vector1` wordt hierin **gekopieerd**.

Als de properties van `vector1` nu gewijzigd worden, blijven de properties van `vector2` - die de originele waarden bevatten - ongewijzigd.

## 3 REFERENCE TYPES

### 3.1 KLASSEN

Een class type zoals Car is een **reference type** en wordt anders behandeld in het geheugen dan een value type. Wanneer je een variabele van het type Car declareert, wordt er slechts een klein stukje geheugen gereserveerd dat groot genoeg is om een geheugenadres te bevatten. Dit geheugenadres verwijst naar de plaats waar een volledig Car object geplaatst kan worden. Het geheugenadres is de **referentie** naar de plaats waar het Car object zal komen te staan.

```
Car myCar;    //reserveer een geheugenblok voor een referentie naar een Car object
```

Het geheugen voor een volwaardig Car-object wordt pas gereserveerd wanneer het sleutelwoord new wordt gebruikt.

```
myCar = new Car(); //creër een Car object in het het geheugen en plaats het adres in myCar
```

Je bestudeert het gedrag in het geheugen best aan de hand van een voorbeeld.

107. Voorzie in de MainWindow een Button btnCopyReferenceType die bij het klikken volgende code uitvoert:

```
Car car1 = new Car("Honda", CarType.HatchBack, 100M);
Car car2 = car1;
car1.Speed += 10;
Debug.Print("car 1 speed = {0}, car 2 speed = {1}", car1.Speed, car2.Speed);
```

De output ziet er als volgt uit:

```
car 1 speed = 110, car 2 speed = 110
```

Zoals je merkt is de snelheid van car1 en car2 gelijk, hoewel je enkel de snelheid van car1 hebt gewijzigd. Dit komt omdat de variabelen car1 en car2 slechts referenties zijn naar **hetzelfde object** op de **Heap**. Wijzigingen aan dat object zijn dus voor beide variabelen van toepassing.

Stap voor stap:

```
Car car1
```

Er wordt een geheugenplaatsje genaamd car1 gedeclareerd voor het bijhouden van een geheugenadres op de **heap**. Er is echter nog geen Car object aangemaakt op de heap, dus dat geheugenplaatsje refereert naar nergens (null).

```
Car car1 = new Car("Honda", CarType.HatchBack, 100M);
```

Er wordt een nieuw object aangemaakt van het type `Car`. Het geheugenblok dat hiervoor gebruikt wordt bevindt zich op de heap en is relatief groot; het moet immers alle properties van de instantie kunnen bijhouden.

Het adres (de referentie) van het heap-geheugenblok waar die instantie werd geplaatst wordt toegekend aan variabele `car1`. Met die variabele kan je het object benaderen vanuit je code.

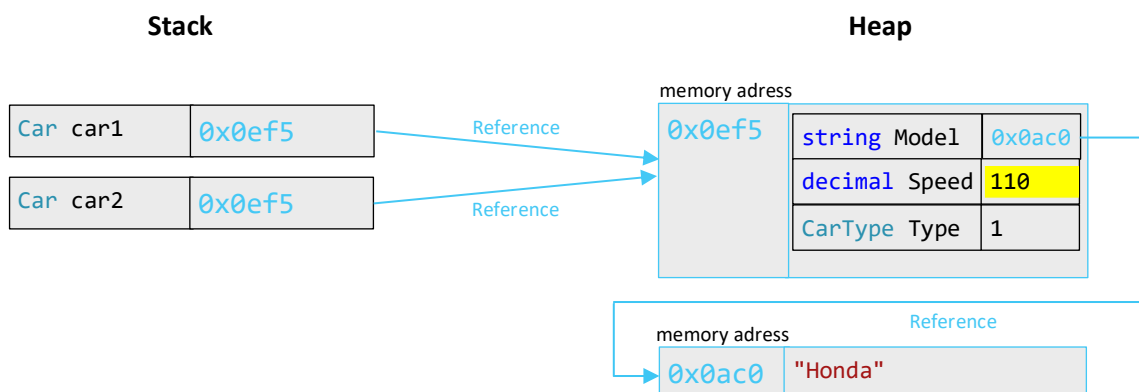
```
Car car2 = car1;
```

Er wordt een geheugenplaatsje genaamd `car2` gedeclareerd voor het bijhouden van een geheugenadres. De waarde van `car1`, een geheugenadres van een reeds bestaand object, wordt er aan toegekend. Zowel `car1` als `car2` refereren nu naar hetzelfde object op de heap.

```
car1.Speed += 10;
```

De snelheid van het object dat gerefereerd wordt door variabele `car1` wordt met 10 verhoogd. Omdat `car2` naar hetzelfde geheugenblok/instantie verwijst, is deze wijziging ook daaruit te lezen.

In het geheugen zou deze situatie er als volgt uitzien:



Let op de dubbele verwijzing naar het ene heap-geheugenblok die de `Car` instantie bijhoudt.

### 3.2 STRING

Het `string` type is een uitzonderlijk geval. Het is, tot grote verwarring, wel degelijk een **reference type**, want de eigenlijke waarde ervan bevindt zich steeds op de heap.

Een `string` gedraagt zich echter meer als een **value type**. Dat komt omdat de waarde van een `string` geheugenblok nooit gewijzigd mag worden. Dit wordt ook wel een *immutable* type genoemd.

Wanneer je de referentie van een eerste `string` variabele toewijst aan een andere `string` variabele, dan levert dat steeds een kopie op van de eerste `string`. Net zoals dat bij `value types` het geval is.

108. Voorzie in de `MainWindow` een `Button` `btnCopyReferenceTypeString` met de volgende Click handler:

```
string string1 = "hello";
string string2 = string1;
string1 = string1.ToUpper(); //string1 krijgt een nieuwe waarde
```



```
Debug.Print("string1 = {0}, string2 = {1}", string1, string2);
```

De output ziet er als volgt uit:

```
string1 = HELLO, string2 = hello
```

In tegenstelling tot class variabelen, en in overeenstemming met value types, is `string2` ongewijzigd gebleven. De nieuwe waarde van `string1` krijgt een **volledige** nieuwe plek op de heap, omdat het oorspronkelijk object waar `string1` naar verwijst niet gewijzigd mag worden (door de zogenaamde *immutability* van het `string` type). `string2` verwijst nog steeds naar het oorspronkelijke geheugenblok dat de waarde "hello" bevat.

## 4 DE WAARDE “NULL” EN NULLABLE TYPES

### 4.1 NULL

Wanneer een reference type variabele zoals een declaratie van de klasse Car nog niet verwijst naar een geheugenadres, dan bevat deze de waarde null. Deze waarde kan bijgevolg enkel voorkomen bij reference types.

```
Car car1;
```

De variabele car1 werd nog niet geïnitieerd en is nu nog leeg: car1 bevat de waarde null.

Eens de variabele is ingesteld op een object, kan je de objectverwijzing teniet doen door de reference waarde null toe te kennen:

```
Car car1 = new Car("Honda", CarType.HatchBack, 100M);
Car car2 = car1;
car1 = null; //car1 verwijst niet meer naar het object, maar car2 doet dat wel nog.
```

Merk op dat je daarbij het object in het geheugen niet vrijgeeft, maar je enkel de verwijzing (het geheugen-adres) naar dat blok geheugen vernietigt. Je mag nooit de properties of methoden van een variabele met waarde null opvragen. Dat levert een NullReferenceException en crasht de applicatie.

```
Car car1 = new Car("Honda", CarType.HatchBack, 100M);
Car car2 = car1; //kopieer de referentie
car1 = null; //car1 verwijst niet meer naar het object, maar car2 doet dat wel nog.
Debug.Print("Honda's speed: {0}", car2.Speed); //levert "Honda's speed: 100"
//Debug.Print("Honda's speed: {0}", car1.Speed); //Crash! NullReferenceException
car2 = null; //de Car instantie in het geheugen kan nu opgeruimd worden
```

Zolang er referenties naar toe blijven bestaan, zal ook dat heap-geheugenblok blijven bestaan. Eens alle referenties verloren zijn gegaan, bijvoorbeeld om dat de lokale variabelen niet meer beschikbaar zijn (einde van een methode), zal de Garbage Collector het toegekende heap-geheugen opschonen.

### 4.2 NULLABLE TYPES

De null value is handig om te controleren of een objectvariabele geïnitieerd is.

```
Car myCar;
if(myCar == null)
{
    MessageBox.Show("The variable named myCar has no reference to any object instance");
}
```

Omdat null een referentiewaarde is, kan je null **niet toewijzen** aan **value types**.

```
int number = null; //generates a compile-time error!
```

Het is echter wel mogelijk om voor een value type variabele aan te geven dat het **nullable** is, door middel van de **?** operator. De variabele wordt dan een reference type.

```
int? nullableNumber = null;
```

Je kan voor een nullable type checken op een waarde null of gebruik maken van de property `HasValue`:

```
int? nullableNumber = null;
if (nullableNumber.HasValue)
    Debug.Print($"nullableNumber has value: {nullableNumber.Value}");
else
    Debug.Print("nullabeNumber is null");
```

De property `Value` is read-only, om een nieuwe waarde aan een nullable type toe te kennen gebruik je een gewoon toekenningstatement:

```
int? nullableNumber = 42;
```

## 5 REF EN OUT PARAMETERS

### 5.1 PARAMETERS DOORGEVEN ALS REFERENTIE

Voor value types geldt dat, wanneer je de waarde van een variabele toekent aan een andere variabele, de **waarde** ervan gekopieerd wordt. Dit heb je uitvoerig gezien in de voorbeelden.

Dit principe is ook van toepassing wanneer je een value type variabele als argument meegeeft aan een methode.

109. Maak in de MainWindow een methode die een value type parameter verwacht:

```
private int IncrementNumber(int number)
{
    number++;
    return number;
}
```

110. Voorzie in de MainWindow een Button btnPassValueType met de volgende Click handler:

```
int getal1 = 5;
int getal2 = IncrementNumber(getal1);
Console.WriteLine($"getal1:
{getal1}\ngetal2: {getal2}");
```

De output ziet er als volgt uit:

```
getal1: 5
getal2: 6
```

Wellicht ben je enigszins verrast door dit resultaat?

De methode IncrementNumber ontvangt de int (value type) waarde 5 en kopieert de **waarde** naar zijn parameter number. Daarna wordt deze waarde verhoogd. De oorspronkelijke kopie, opgeslagen in variabele numberToInput, blijft ongewijzigd. Bij reference types is dat het geval niet.

Indien je de value type toch wenst door te geven als referentie, dan kan je gebruik maken van het keyword ref. Hiermee wordt dus het geheugenadres van de variabele bedoeld.

111. Maak in de MainWindow een methode IncrementNumberRef. Ditmaal verwacht het een ref parameter:

```
private void IncrementNumberRef(ref int number)
{
    number++;
}
```

112. Voorzie in de MainWindow een Button btnPassValueTypeByRef met de volgende Click handler:

```
int getal1 = 5;
int getal2 = IncrementNumber(ref getal1);
Console.WriteLine($"getal1: {getal1}\ngetal2: {getal2}");
```

De output ziet er als volgt uit:

```
getal1: 6
getal2: 6
```

Dit keer is niet de waarde van de value type variabele `numberToInput` meegegeven aan de methode, maar een **referentie** naar de waarde ervan. De methode `IncrementNumberRef` heeft zo rechtstreeks toegang tot de waarde en kan deze intern wijzigen.

## 5.2 UITGAANDE PARAMETERS

Indien een variabele nog niet geïnitieerd is alvorens je ze als een **by reference argument** meegeeft aan een methode, dan kan je het **out** keyword gebruiken.

113. Maak in de `MainWindow` een methode `InitializeMe` die een out parameter verwacht:

```
private bool InitializeMe(out int numberToInitialize)
{
    //the value of numberToInitialize is always discarded when passing as out
    numberToInitialize = 42;
    return true;
}
```

114. Voorzie in de `MainWindow` een Button `btnOutputParameter` met de volgende Click handler:

```
int numberComingOut = 66;
bool ok = InitializeMe(out numberComingOut);
Debug.Print("numberComingOut has value: {0}", numberComingOut);
```

De output ziet er als volgt uit:

```
numberComingOut has value: 42
```

De waarde 66, meegegeven als out argument, wordt steeds genegeerd. Het out keyword werkt slechts in **één richting**. Het ref keyword werkt in **beide richtingen**.

Een goed voorbeeld van out argumenten in de praktijk zijn de `int.TryParse()`, `double.TryParse()`, `DateTime.TryParse()`,... methoden.

Ze laten toe om op een zeer beknopte manier een string te parsen naar een ander datatype omdat je zelf geen try/catch structuur moet schrijven. Daarbij leveren ze twee waarden: een bool als return waarde die aangeeft of het parsen geslaagd is, en (als dat zo is) de omgezette waarde als out argument, bijvoorbeeld een `int`.

## 6 BOXING EN UNBOXING

### 6.1 DE KLASSE SYSTEM.OBJECT

Eén van de belangrijkste referentietypes uit het volledige .NET Framework is de klasse `Object` uit de namespace `System`.

Voorlopig is het voldoende in te zien dat `System.Object` de **basisklasse** is van alle types in .NET, als het ware de stamouder van alle types die ingebouwd zijn of die we zelf maken. Alle types **erven** ervan over. Dat is trouwens ook de reden waarom alle klassen automatisch de `ToString()` methode krijgen. Het sleutelwoord `object` is overigens een alias voor `System.Object`, ze betekenen allebei net hetzelfde.

Omdat alle types in het .NET Framework overerven van `object` zijn ze er allemaal compatibel mee. Dat is het principe van *Polymorfisme*, waar later dieper op ingegaan wordt. Voorlopig is het voldoende om te stellen dat elk type, of het nu een `int` of een `Car` is, ook als een variabele van het type `object` kan worden beschouwd.

115. Maak in de `MainWindow` een methode `PrintObjectAsString` aan:

```
private void PrintObjectAsString(object whateverObject)
{
    //print the actual TYPE of the object passed in followed by the value of the
    ToString() method
    Debug.Print("Object is a {0}, ToString: {1}",
        whateverObject.GetType().ToString(),
        whateverObject.ToString());
}
```

Omdat de parameter van het type `object` is, en omdat alles type beschouwd kunnen worden als `object`, maakt het niets uit wat het eigenlijke type van het meegegeven argument is. Deze methode aanvaardt dus gewoon variabelen van het type `object`, maar ook van alle types die van `object` overerven, zoals `int`, `string`, `Car`, `List<long>`, enz...

Omdat de klasse `object` de methodes `GetType()` en `ToString()` definieert, kunnen we deze zonder probleem benutten. Andere methoden, properties of velden van het meegegeven argument zijn echter niet meer toegankelijk in de `object`-vorm.

## 6.2 BOXING

Boxing is het proces waarbij een **value type** variabele naar een object (of een door dat type geïmplementeerde interface) wordt omgezet. Interfaces en het object type zijn **reference types**, waardoor de waarde van de value-type variabele op de heap bewaard zal worden.

116. Voorzie in de MainWindow een Button btnBoxing met de volgende Click Handler:

```
int number = 42;
object boxedNumber = number; //boxing happens implicitly

number++; //this variable is still on the stack

Car myCar = new Car("Ford", CarType.Sedan, 90);
object abstractCar = myCar; //no boxing, myCar is already on the heap

PrintObjectAsString(boxedNumber);
PrintObjectAsString(number);
PrintObjectAsString(abstractCar);
```

De output ziet er als volgt uit:

```
Object is a System.Int32, ToString: 42
Object is a System.Int32, ToString: 43
Object is a CsCourse.ValueReference.Lib.Car, ToString: Ford driving at
```

90

Betreffende het toekennen van een value type waarde aan een object type (dit is Boxen):

- `number` is een **lokale variabele** en een **value type**. De beginwaarde 42 wordt op de **stack** bewaard. Wanneer de waarde wordt toegekend aan een object type, dan spreken we van Boxen. Er wordt een nieuwe geheugenplaats op de heap aangemaakt, met daarin de waarde 42.
- `boxedNumber` is een **lokale variabele** en een **reference type**. De eigenlijke waarde 42 wordt op de **heap** bewaard. Op de stack wordt enkel de referentie naar dat heap-geheugen bewaard.
- Wanneer de waarde van `number` gewijzigd wordt, dan heeft dat uiteraard geen effect op de `boxedNumber` want dat is een volledig ander stukje geheugen.

Betreffende het toekennen van het reference type waarde aan een object type (dit is **geen** Boxen):

- `myCar` is een **lokale variabele** en een **reference type**. De eigenlijke waarde wordt op de heap bewaard en de verwijziging genaamd `myCar` staat op de stack.

Wanneer `myCar` wordt toegekend aan een object, dan maak je **geen** nieuw object aan op de heap zoals dat het geval was voor `number` naar `boxedNumber`. Dit is geen boxing want er verandert niets in de heap. Er wordt enkel een nieuwe stackvariabele genaamd `boxedCar` aangemaakt.

## 6.3 UNBOXING

Bij unboxen wordt de **omgekeerde** conversie gedaan: een geboxet value type wordt terug naar de stack gekopieerd naar zijn **originale vorm**.

117. Voorzie in de MainWindow een Button btnUnboxing met de volgende Click Handler:

```
int number = 42;
object boxedNumber = number; //boxing
```

Wanneer je de waarde van boxedNumber (reference type) nu in een nieuwe int variabele genaamd unboxed (value type) wil kopiëren, dan verwacht je misschien dat dit zal lukken:

```
int unboxed = boxedNumber; //compile error! you need to cast.
```

Dit leidt echter tot de volgende compileerfout: *Cannot implicitly convert type 'object' to 'int'. An explicit conversion exists (are you missing a cast?)*

Om de waarde van boxedNumber, dat zich op de heap bevindt, te kopiëren in de int value type unboxed moet je boxedNumber onderwerpen aan een *typecast*. Hiervoor geef je aan naar welke type je wenst terug te keren, omgeven door ronde haken. Unboxing is dus een expliciete operatie, omdat je de compiler een handje moet helpen.

118. Wijzig de code van de btnUnboxing\_Click methode als volgt:

```
int number = 42;
object boxedNumber = number; //boxing
int unboxed = (int)boxedNumber; //unboxing to correct type
```

Bij het unboxen **kopieer** je dus de geboxete waarde uit zijn omkaderende objectdoos en plaats je die op de stack. Er staan nu drie verschillende variabelen op de stack: number, boxedNumber en unboxed. Allen zijn onafhankelijk van elkaar.

## 6.4 VEILIG CASTEN MET AS EN IS

Wanneer je aan typecasting doet, moet je je hoeden voor een InvalidCastException. Dat geldt voor zowel value als reference types.

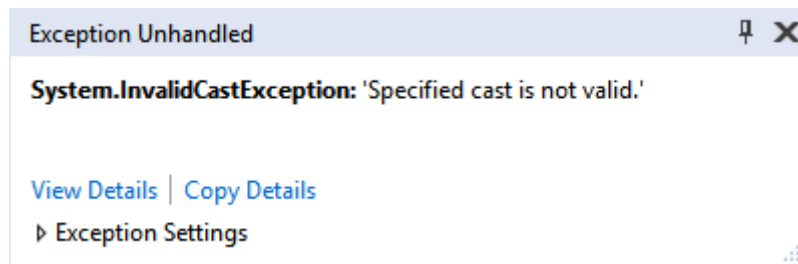
Stel dat je de volgende code schrijft:

```
Car myCar = new Car("Toyota", CarType.SUV, 65);
object boxedCar = myCar; //Remember, this is not boxing (reference types)
int myCarAgain = (int)boxedCar; //Crash! this throws an InvalidCastException!
```

Hier instantieer je een Car genaamd myCar. Je maakt een nieuwe variabele aan genaamd boxedCar die verwijst naar dezelfde instantie, maar ditmaal in object-vorm.



Daarna doe je een *typecast* van `boxedCar` naar het type `int`. Dat gaat helemaal fout, want op die geheugenplaats zit immers geen waarde van dat type. Er zit een `Car` instantie. De compiler laat deze code toe, maar je applicatie zal genadeloos crashen.



De enige juiste manier is de volgende:

```
Car myCarAgain = (Car)boxedCar; //Casting back to correct type
```

Wie graag gebruikt maakt van het `var` keyword moet dus extra goed opletten voor dit soort run-time fouten.

Er bestaan echter ook keywords die je helpen om deze fout te voorkomen.

## Is

Met het `is` keyword kan je controleren of een bepaalde variabele wel degelijk van het verwachte type is. Als dat niet het geval is dan cast je niet, om de exception te vermijden.

```
Car myCar = new Car("Toyota", CarType.SUV, 65);
object boxedCar = myCar;
if(boxedCar is Car) //only cast if we're sure about the type
{
    Car myCarAgain = (Car)boxedCar;
}
```

## As

Met het keyword `as` kan je een *safe typecast* doen. Indien de waarde van het object niet gelijk is aan het verwachte type, dan levert dat `null` op in plaats van een exception.

*Safe typecasting* met `as` werkt enkel met reference types, en kan je dus niet gebruiken voor unboxen (value types).

119. Voorzie in de `MainWindow` een Button `btnSafeCasting` met de volgende Click Handler:

```
int? myNumber = 42;
object boxedNumber = myNumber;
int? myNumberAgain = boxedNumber as int?; //safe typecast

if(myNumberAgain != null)
{
    Debug.Print($"myNumberAgain: {myNumberAgain.ToString()}");
}

Car myCar = new Car("Toyota", CarType.SUV, 65);
object boxedCar = myCar;
```

```
Window myCarAgain = boxedCar as Window; //safe cast fails due to type mismatch,
myCarAgain is null!

if (myCarAgain != null)
{
    Debug.Print($"myCarAgain: { myCarAgain.ToString()}");
}
```

De output ziet er als volgt uit:

```
myNumberAgain: 42
myCarAgain was casted to wrong type
```

De variabele `myCarAgain` wordt ingesteld op `null` omdat er gecast wordt naar een instantie van het type `Window`, maar de variabele `boxedCar` verwijst in werkelijkheid naar een instantie van `Car`. Het casten faalt, maar ditmaal zonder crash.

## 6.5 HET NUT VAN BOXING EN UNBOXING

Dankzij boxing en unboxing levert C# een universeel type systeem dat toelaat om objecten onafhankelijk van hun onderliggende data te benaderen. Er bestaan veel methodes in het .NET framework die argumenten van het type object aanvaarden. Wanneer je daar een value type in plaatst, dan ben je aan het Boxen. Zo'n methode is bijgevolg **zeer breed toepasbaar** voor **alle huidige én toekomstige** types.

Boxing en Unboxing betekent echter een extra werklast voor de CPU omdat de value type gekopieerd moet worden van en naar de heap. Het wordt dus bij voorkeur vermeden tenzij je echt geen andere keuze hebt. Overweeg altijd eerst het gebruik van **generics** alvorens je value types gaat boxen. Indien jouw applicatie geen nood heeft aan zo'n universeel systeem voor uitwisselbaarheid van types, dan hoeft je Boxing dan ook helemaal niet te gebruiken.

Hoe het ook zij, een goed begrip van boxing is onontbeerlijk voor elke programmeur die inzicht wenst te verwerven in het geheugenbeheer van .NET. Ook begrippen zoals type casting en safe typecasting worden daardoor duidelijker.