

# Klassen en objecten

Programming Basics

## INHOUD

<b>1</b>	<b>INLEIDING</b>	<b>4</b>
1.1	Classificatie	4
1.2	Abstractie (black-box)	4
1.2.1	Encapsulatie	5
<b>2</b>	<b>VOORBEREIDING CURSUSVOORBEELD</b>	<b>6</b>
<b>3</b>	<b>CLASS LIBRARY</b>	<b>7</b>
<b>4</b>	<b>GEBRUIK VAN KLASSEN</b>	<b>10</b>
4.1	Een klasse aanmaken	10
4.2	Objecten aanmaken	11
4.3	Praktisch voorbeeld van klassen en objecten	13
4.3.1	Klassen en velden public maken	13
4.3.2	Referentie maken naar een klassenbibliotheek	14
4.3.3	Gebruik van de klasse Car	16
4.4	Constructors	18
4.4.1	Default constructor	18
4.4.1.1	Conclusie :	18
4.4.2	Constructors overladen	19
4.5	Gebruik van static	21
4.5.1	Static fields	21
4.5.2	Static methods	23
4.6	Partial klassen	26
4.7	Besluiten tot dusver	28
<b>5</b>	<b>GEBRUIK VAN DE KLASSENIBLIOTHEEK</b>	<b>29</b>
5.1	override string ToString()	31
5.2	Een object casten	32
<b>6</b>	<b>ANONIEME KLASSE</b>	<b>33</b>
<b>7</b>	<b>PROPERTIES (EIGENSCHAPPEN) : INLEIDING</b>	<b>34</b>
7.1	Wat zijn properties	34
7.2	Praktisch voorbeeld	34
<b>8</b>	<b>DATA AFSCHERMEN MET EEN METHODE (GETTERS &amp; SETTERS)</b>	<b>36</b>

8.1	Conclusie	38
9	DATA AFSCHERMEN MET EEN PROPERTY	39
9.1	Een eerste prop maken	39
9.2	Good practice	41
9.3	Properties : verkorte notatie	41
9.4	Nullable properties	42
9.5	Auto-property initializer	42
9.6	Read-only properties	43
10	AANPASSEN VAN ONZE VOORBEELD APPLICATIE	44
11	COMPOSITIE	47

# 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 (ander woord: instantie of instance). 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 `Car` 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.

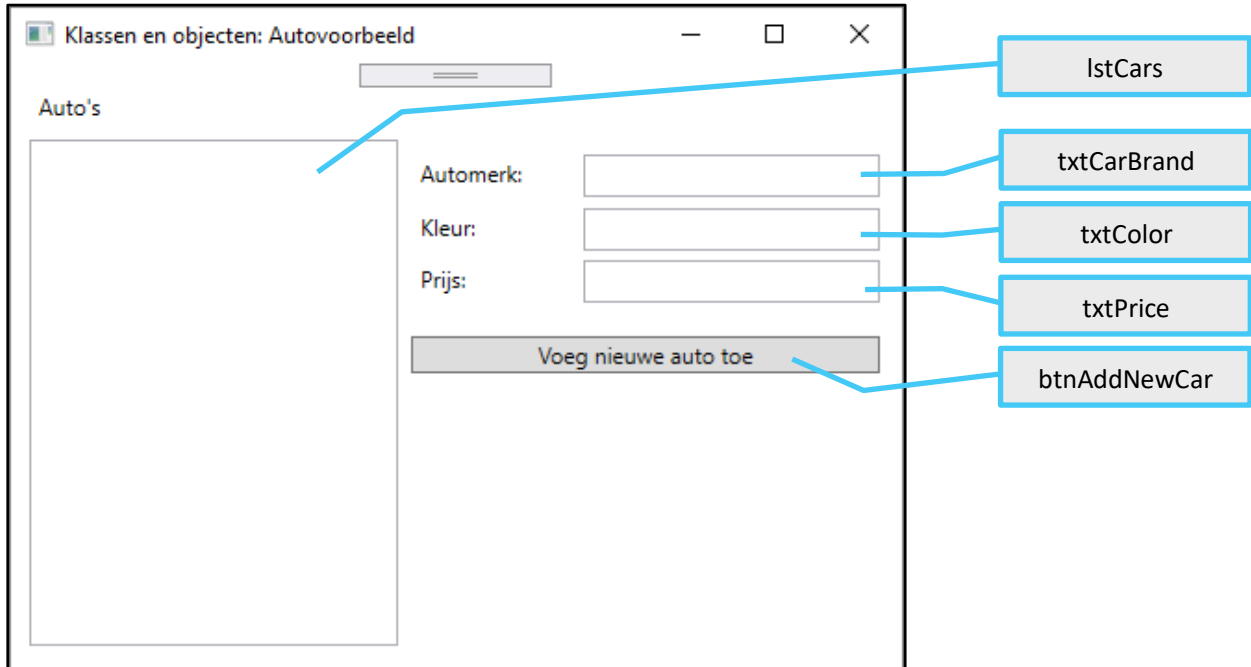
- Wanneer je een WPF `Window` aanmaakt, hoef je je geen zorgen te maken over hoe deze getoond wordt, hoe de knopjes minimaliseren, maximaliseren en sluiten worden aangemaakt.
- Wanneer je de methode `WriteLine` van de klasse `Console` uitvoert, hoef je je niet af te vragen hoe deze zin in de `Console` wordt getoond.
- Wanneer je de methode `Accelerate` van je eigen klasse `Car` 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.

### **1.2.1 ENCAPSULATIE**

Een ander doel van het gebruik van klassen, het ontwerpen van een object georiënteerd model, is de afscherming 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. Verderop in dit hoofdstuk gaan we aan de slag met deze WPF-applicatie.



### Code repository

De broncode van dit cursusvoorbeeld is te vinden op

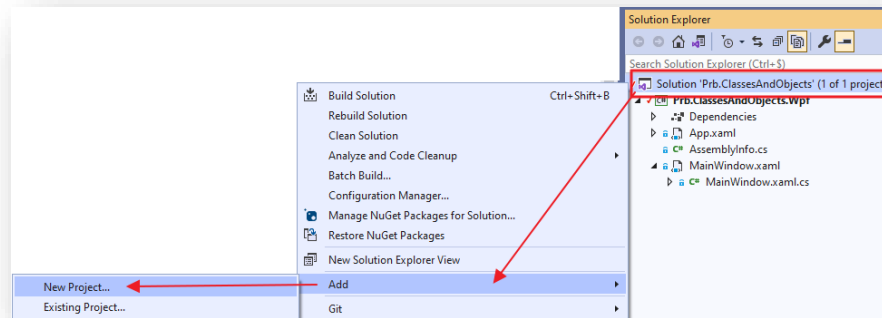
 `git clone https://github.com/howest-gp-prb/cu-h9-KlassenEnObjecten-start.git`

### 3 CLASS LIBRARY

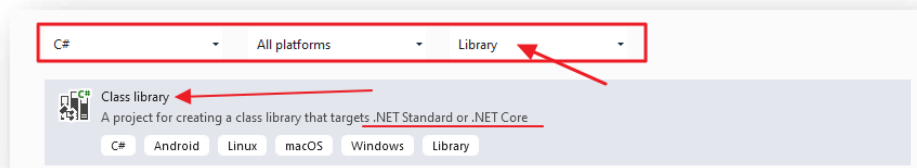
Wanneer je een klasse definieert, kan het de bedoeling zijn om deze klasse te gaan hergebruiken in een andere applicatie. Dit kan een andere WPF-applicatie, maar ook een WinForms, Console of ASP.net toepassing zijn.

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

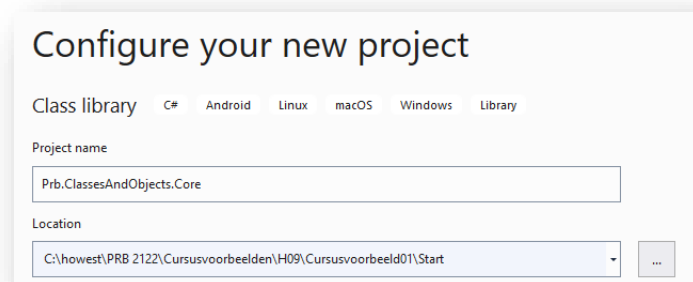
1. Rechtsklik op de **solution** en selecteer **Add > New Project** in het snelmenu :



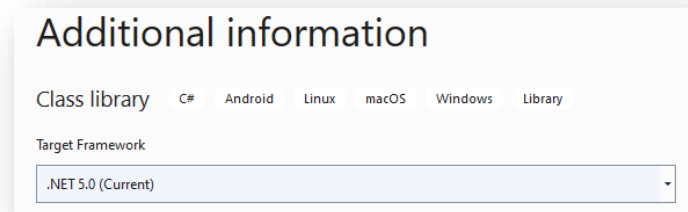
2. Selecteer **Class Library** uit de templates (pas eventueel de filters bovenaan aan) en daarna op Next .



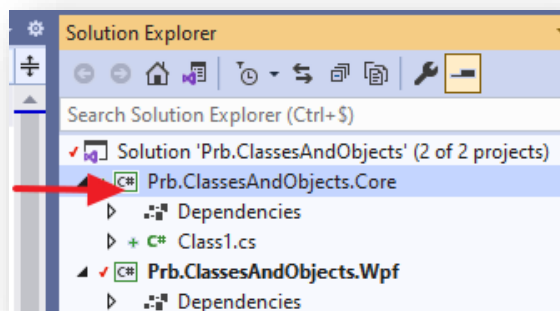
3. Geef de ClassLibrary de naam **Prb.ClassesAndObjects.Core** en klik daarna op Next.



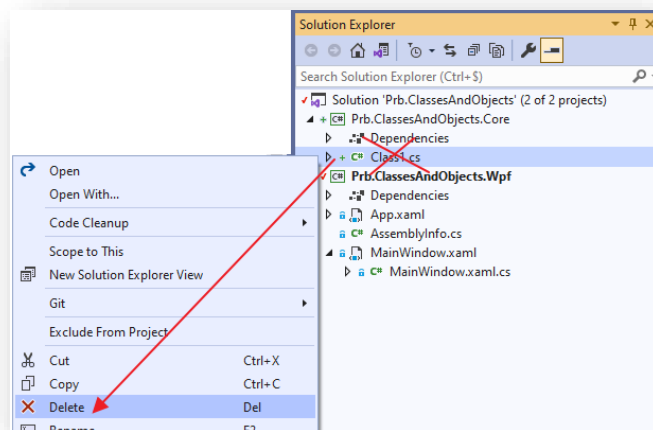
4. Kies in het laatste venster de .Net versie waarmee je aan de slag wenst te gaan. Deze MOET dezelfde zijn als de versie die in je WPF project gebruikt wordt (in de starterscode wordt **.Net 5.0** gebruikt, dus kiezen we hier ook deze versie). Klik tenslotte op Create.



5. De Class Library met naam Prb.ClassesAndObjects.Core werd aangemaakt en is nu te vinden als een tweede project in je solution explorer.



6. Verwijder het codebestand `Class1.cs` dat automatisch werd aangemaakt.







### **Klassenbibliotheek of Class Library**

Een klassenbibliotheek definieert typen en methoden die door een applicatie opgevraagd kunnen worden.

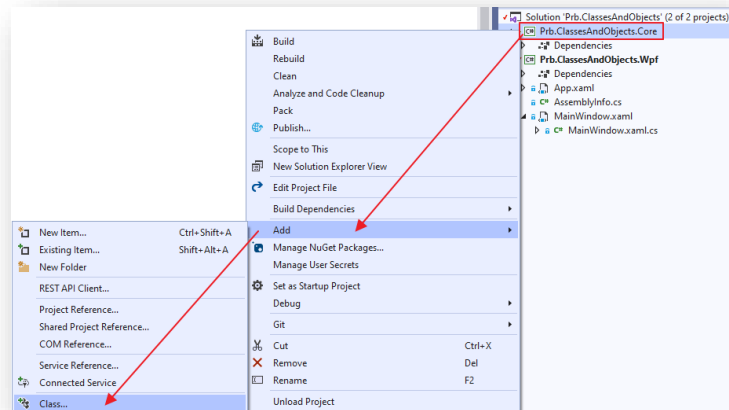
Na aanmaak van een klassenbibliotheek kun je bepalen of deze gedistribueerd wordt als een onderdeel van een applicatie of je ze wilt integreren als een gebundeld onderdeel met een of meerdere toepassingen.

## 4 GEBRUIK VAN KLASSEN

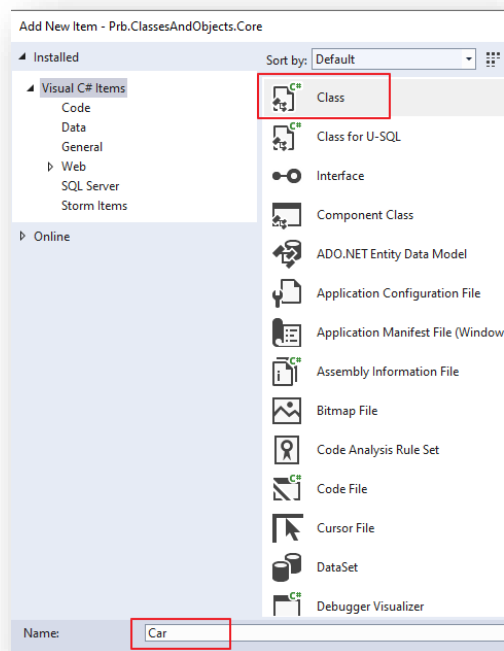
### 4.1 EEN KLASSE AANMAKEN

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

1. Klik rechts op je Class Library `Prb.ClassesAndObjects.Core` en kies **Add** → **Class...**



Geef de class de naam **Car** en klik op **Add**.



#### Naamgeving klassen

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

We zien nu volgende code verschijnen :

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

namespace Prb.ClassesAndObjects.Core
{
    class Car
    {
    }
}
```

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 Car.

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

- Brand (merk)
- Color (kleur)
- Price (prijs)

Dat zijn dus enkele kenmerken van een auto die wij willen tonen in onze applicatie. We gaan van onze klasse Car 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 Car:

```
namespace Prb.ClassesAndObjects.Core
{
    class Car
    {
        string brand;
        string color;
        decimal price;
    }
}
```

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. Met andere woorden, een object is een mogelijke invulling van een klasse. We hebben nu net een klasse Car aangemaakt met de verschillende kenmerken (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 `Car`.

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

```
Car car1 = new Car();
```

Of

```
Car car1;  
car1 = new Car();
```

- We starten dus met het schrijven van `Car`, 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 `car1`. We kunnen zelf kiezen welke naam we aan objecten geven. We konden dit object evengoed `newCar` genoemd hebben in plaats van `car1`.

*Car is een klasse*

*car1 is een object*

*car1 is een instance van de klasse Car*

- Na de object naam schrijven we het `=` teken, wat betekent dat we een waarde zullen toekennen aan ons `car1` 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 `Car`.
- `Car()` betekent eigenlijk dat we de constructor aanroepen van de klasse `Car`. Constructors zullen later in dit hoofdstuk nog besproken en uitgelegd worden.

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

```
Car niceNewCar = new Car();
```

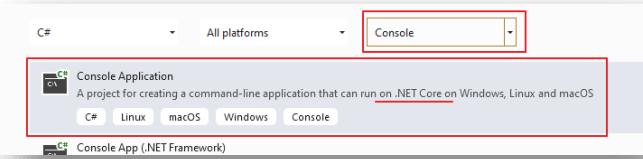
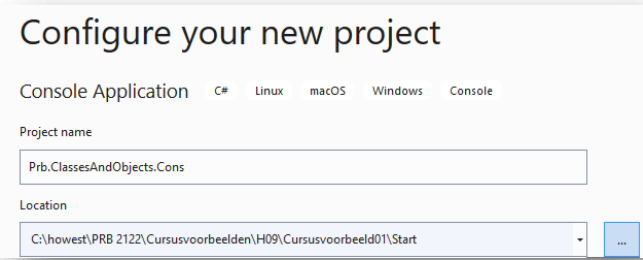
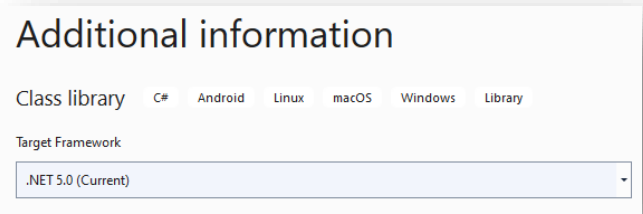
We hebben nu reeds twee objecten van het type `Car`:

- `car1`
- `niceNewCar`

## 4.3 PRAKTISCH VOORBEELD VAN KLASSEN EN OBJECTEN

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

Klik met de rechtermuisknop op je SOLUTION en kies in het snelmenu voor Add → New project.

Kies in de lijst met templates voor Console Application (pas eventueel je filters aan; let er op dat je kiest voor een .Net Core Console applicatie :	
In het volgende venster voer je de naam in van dit nieuwe project : Prb.ClassesAndObjects.Cons :	
Kies de juiste versie voor dit nieuwe project (.Net 5.0).  We gaan zo meteen aan de slag met dit nieuwe project.	

### 4.3.1 KLASSEN EN VELDEN PUBLIC MAKEN

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

```
namespace Prb.ClassesAndObjects.Core
{
    class Car
    {
        string brand;
        string color;
        decimal price;
    }
}
```

- Onze klasse Car is **private** : als er niets voor het sleutelwoord **class** staat, dan is deze impliciet private. We kunnen deze klasse enkel gebruiken binnen zijn eigen namespace (hier dus Prb.ClassesAndObjects.Core).

Dit geldt ook voor de aanwezige velden (de variabelen brand, color en price) : deze kunnen enkel gebruikt worden door de klasse zelf. Opnieuw, staat er niets voor het type (string, decimal) dan zijn deze impliciet private.

- Om dit op te lossen maken we de klasse Car en zijn members (in dit geval fields/velden) **public**. Maak de klasse Car public en pas de code als volgt aan:

```
namespace Prb.ClassesAndObjects.Core
{
    public class Car
    {
        public string brand;
        public string color;
        public decimal price;
    }
}
```

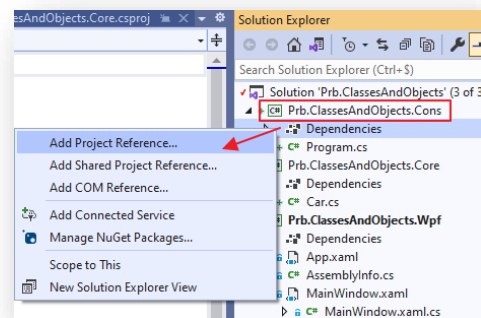
We maken in dit praktisch voorbeeld gebruik van twee (van de drie) projecten in onze solution, namelijk **Prb.ClassesAndObjects.Cons** en **Prb.ClassesAndObjects.Core** waarin onze **public** klasse Car aanwezig is.

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

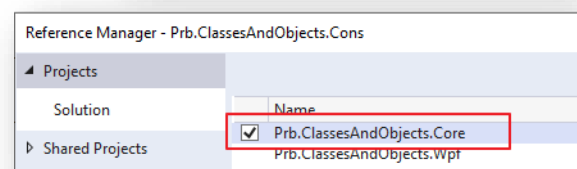
#### 4.3.2 REFERENTIE MAKEN NAAR EEN KLASSENBIBLIOTHEEK

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

- Rechtsklik in het project **Prb.ClassesAndObjects.Cons** op “Dependencies” en klik op **Add Project Reference...**



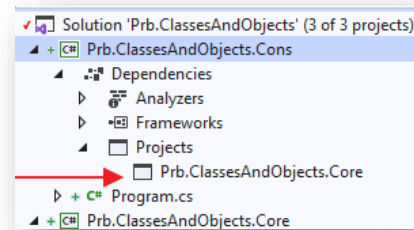
- De Reference Manager opent en we zien onze twee andere projecten in deze solution in de lijst staan.



*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.*

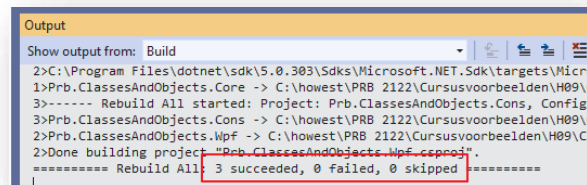
Vink de checkbox aan bij het project waar je een referentie naartoe wenst te leggen. In ons geval **Prb.ClassesAndObjects.Core** . Want we willen dat de klasse Car gekend is (gebruiken) in onze Console applicatie. Klik op OK.

- Na het uitklappen van de Dependencies -> Projects in **Prb.ClassesAndObjects.Cons** zien we dat de namespace wel degelijk is toegevoegd:



- Compileer nu je volledige solution (**CTRL + SHIFT + B**) of via **Build → Build Solution (of Rebuild Solution)**.

*Ter info, in je output window stel je inderdaad vast dat er nu 3 projecten worden gecompileerd :*

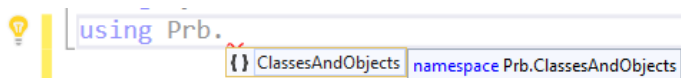


- Wensen we de klasse Car te gebruiken in onze console applicatie, dan kunnen we dit door aan te geven dat we de namespace willen gebruiken. We gebruiken hiervoor een using directive. Open de Program.cs file in je console applicatie :

```
using System;
using Prb.ClassesAndObjects.Core;

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

*Tip : van zodra je bij using "Prb." hebt ingetypt zal de intellisence van Visual Studio je snel op weg helpen om de juiste namespace te kiezen.*



Hiermee vermijden we dat we telkens de naam van de namespace moeten tikken als we iets uit **Prb.ClassesAndObjects.Core** gebruiken.

Ter info : mochten we deze using directive bovenaan NIET plaatsen, dan zouden we telkens volgende moeten schrijven :

```
Prb.ClassesAndObjects.CORE.Car newCar = new Prb.ClassesAndObjects.CORE.Car();
```

Het is duidelijk dat deze manier van werken minder overzichtelijke code oplevert.

### 4.3.3 GEBRUIK VAN DE KLASSE CAR

In de Program.cs klasse van ons Console App project gaan we een nieuw object van de klasse (type) Car gaan aanmaken. We doen dit in de body van de Main methode:

```
using Prb.ClassesAndObjects.Core;

namespace Prb.ClassesAndObjects.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Car newCar;
            newCar = new Car();
        }
    }
}
```

We ontleden even deze code

```
Car newCar;
```

Een variabele met de naam **newCar** wordt **gedeclareerd** als object van het type **Car**.

```
newCar = new Car();
```

De variabele wordt **geïnitialiseerd** met het keyword **new**.

.NET zal nu een stukje geheugen reserveren voor alle members van het nieuwe object.



Onder **members** verstaan we alle eventuele fields, methoden maar – wat we verder in dit hoofdstuk gaan zien – ook properties en constructors van de betrokken klasse.



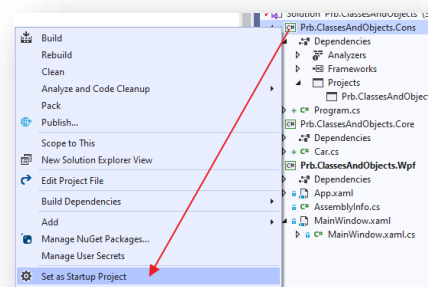
We kunnen nu alle public members van de klasse benaderen:

```
static void Main(string[] args)
{
    Car newCar;
    newCar = new Car();

    newCar.color = "zwart";
    newCar.brand = "Hyundai";
    newCar.price = 15000M;

    Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is
{newCar.color} van kleur en kost {newCar.price}");
    Console.ReadLine();
}
```

Vooraleer je het programma uitvoert, selecteer je eerst je console applicatie als startup project :



Run nu je applicatie :

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Cons\bin\Debug\net5.0\Prb.ClassesAndObjects.Cons.exe
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000
```

We kunnen nu zoveel objecten van het type Car aanmaken als we willen:

```
static void Main(string[] args)
{
    Car newCar;
    newCar = new Car();

    newCar.color = "zwart";
    newCar.brand = "Hyundai";
    newCar.price = 15000M;

    Car secondCar;
    secondCar = new Car();

    secondCar.color = "wit";
    secondCar.brand = "Ford";
    secondCar.price = 9000M;

    Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van
kleur en kost {newCar.price}");
    Console.WriteLine("=====");
    Console.WriteLine($"De tweede auto is een {secondCar.brand}, is {secondCar.color}
van kleur en kost {secondCar.price}");

    Console.ReadLine();
}
```

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Cons\bin
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
```

## 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 (of properties : zie verder) 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.

*Gezien we in onze klasse Car nog geen constructor hebben geschreven, werd dus telkens wij een object van het type Car maakten deze (onzichtbare) standaardconstructor uitgevoerd.*

- Volgend stukje code komt overeen met de standaardconstructor voor onze klasse Car:

```
namespace Prb.ClassesAndObjects.Core
{
    public class Car
    {
        public string brand;
        public string color;
        public decimal price;

        public Car() // dit is het equivalent van de default constructor
        {
            brand = null;
            color = null;
            price = 0;
        }
    }
}
```

- Wanneer een nieuw Car-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.

#### 4.4.1.1 CONCLUSIE :

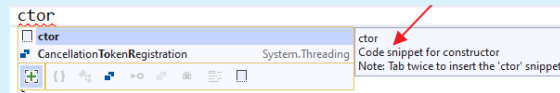
- Een **constructor** is een **methode** die wordt uitgevoerd wanneer een nieuwe instantie van de klasse wordt aangemaakt (**new**)
- Een constructor heeft **exact dezelfde naam** als de klasse en **geen** returnwaarde
- In elke klasse is minstens 1 constructor aanwezig. Desnoods wordt die automatisch door de compiler aangemaakt.



### Tip

Voor het aanmaken van een constructor kun je de code-snippet **ctor** gebruiken.

Na ingeven van **ctor** en **2 x 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 “overloaden” (overladen). Een constructor heeft (en geeft) nooit een returnwaarde. Overloads van een constructor zullen dus enkel verschillen in de parameterlijst.

- Voorzie volgende nieuwe constructors in de klasse `Car`

```
public class Car
{
    public string brand;
    public string color;
    public decimal price;

    public Car() // dit is het equivalent van de default constructor
    {
        brand = null;
        color = null;
        price = 0;
    }

    public Car(string brand) // Constructor ontvangt 1 parameter
    {
        this.brand = brand;
    }

    public Car(string brand, string color) // Constructor ontvangt 2 parameters
    {
        this.brand = brand;
        this.color = color;
    }

    public Car(string brand, string color, decimal price) // Constructor ontvangt 3 parameters
    {
        this.brand = brand;
        this.color = color;
        this.price = price;
    }
}
```

Opgepast, zowel 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 :

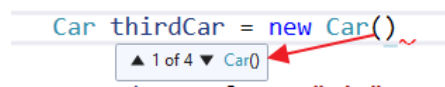
*In de eerste (default) constructor hoefden we dit niet te doen gezien er daar geen conflicten waren met de namen van parameters en fields.*

```
public class Car
{
    public string brand;
    public string color;
    public decimal price;

    public Car() // dit is het equivalent van de
    {
        brand = null;
        color = null;
        price = 0;
    }

    public Car(string brand) // Constructor ontvan
    {
        this.brand = brand;
    }
}
```

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



- We passen de code in onze **console** applicatie aan als volgt:

```
class Program
{
    static void Main(string[] args)
    {
        Car newCar;
        newCar = new Car();

        newCar.color = "zwart";
        newCar.brand = "Hyundai";
        newCar.price = 15000M;

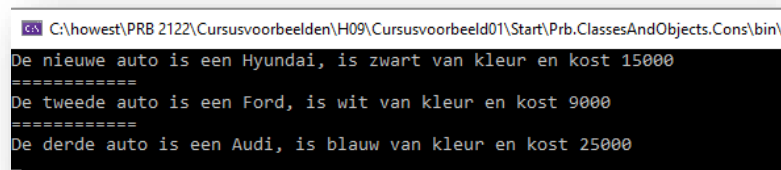
        Car secondCar;
        secondCar = new Car();
        secondCar.color = "wit";
        secondCar.brand = "Ford";
        secondCar.price = 9000M;

        Car thirdCar = new Car("Audi", "blauw", 25000M);

        Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van
        kleur en kost {newCar.price}");
        Console.WriteLine("=====");
        Console.WriteLine($"De tweede auto is een {secondCar.brand}, is {secondCar.color}
        van kleur en kost {secondCar.price}");
        Console.WriteLine("=====");
        Console.WriteLine($"De derde auto is een {thirdCar.brand}, is {thirdCar.color} van
        kleur en kost {thirdCar.price}");

        Console.ReadLine();
    }
}
```

Met volgend resultaat:



```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Cons\bin\
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
```

## 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:

- Declareer een static field met de naam `carCount`. Pas je code aan zoals hieronder :

```
public class Car
{
    public string brand;
    public string color;
    public decimal price;
    public static int carCount;

    public Car() // dit is het equivalent van de default constructor
    {
        brand = null;
        color = null;
        price = 0;
        carCount++;
    }

    public Car(string brand) :this() // Constructor ontvangt 1 parameter
    {
        this.brand = brand;
    }

    public Car(string brand, string color) : this() // Constructor ontvangt 2 parameters
    {
        this.brand = brand;
        this.color = color;
    }

    public Car(string brand, string color, decimal price) : this() // Constructor ontvangt 3
parameters
    {
        this.brand = brand;
        this.color = color;
        this.price = price;
    }
}
```

Een static field van het type `int` met de naam `carCount` werd public gedeclareerd.

In de parameterloze constructor zorgen we ervoor dat `carCount` met 1 wordt opgehoogd als een `Car` object wordt geïntantieerd.

We willen natuurlijk dat dit ook gebeurt wanneer we een andere constructor gebruiken. Hier zou je de opdracht `carCount++;` 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:

Pas de code in de console applicatie als volgt aan :

```
class Program
{
    static void Main(string[] args)
    {
        Car newCar;
        newCar = new Car();

        newCar.color = "zwart";
        newCar.brand = "Hyundai";
        newCar.price = 15000M;

        Car secondCar;
        secondCar = new Car();
        secondCar.color = "wit";
        secondCar.brand = "Ford";
        secondCar.price = 9000M;

        Car thirdCar = new Car("Audi", "blauw", 25000M);

        Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van kleur
en kost {newCar.price}");
        Console.WriteLine("=====");
        Console.WriteLine($"De tweede auto is een {secondCar.brand}, is {secondCar.color} van
kleur en kost {secondCar.price}");
        Console.WriteLine("=====");
        Console.WriteLine($"De derde auto is een {thirdCar.brand}, is {thirdCar.color} van
kleur en kost {thirdCar.price}");
        Console.WriteLine("=====");
        Console.WriteLine($"We hebben nu {Car.carCount} auto's");

        Console.ReadLine();
    }
}
```



#### Opgelet

- We verwijzen hier niet naar een Car variabele (newCar, secondCar of thirdCar) maar naar de klasse Car. Dit omdat het static field (carCount) door alle Car-objecten wordt gedeeld!

Met als resultaat:

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObject
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 Car-objecten, dan zouden we dit **zonder** static methods op volgende manier kunnen oplossen:

- Voeg volgende code toe in de klasse Car:

```
namespace Prb.ClassesAndObjects.CORE
{
    public class Car
    {
        ...

        public decimal PriceDifference(Car someCar)
        {
            return Math.Abs(this.price - someCar.price);
        }
    }
}
```

In de methode PriceDifference wordt als parameter (someCar) een object van het type Car meegegeven. We berekenen het prijsverschil (in absolute cijfers) tussen het huidige object en het object dat als parameter werd doorgegeven en retourneren een decimal-waarde.

- We passen de code aan in de console applicatie:

```
namespace Prb.ClassesAndObjects.CONSOLE
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

            Car thirdCar = new Car("Audi", "blauw", 25000M);
            decimal costDifference = newCar.PriceDifference(thirdCar);

            ...

            Console.WriteLine($"We hebben nu {Car.carCount} auto's");
            Console.WriteLine("=====");
            Console.WriteLine($"Prijsverschil tussen {newCar.price} en {thirdCar.price} is {costDifference}");

            Console.ReadLine();
        }
    }
}
```

Met volgend resultaat:

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Con
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.

- Pas de code in de klasse Car aan als volgt:

```
namespace Prb.ClassesAndObjects.CORE
{
    public class Car
    {
        ...

        public decimal PriceDifference(Car someCar)
        {
            return Math.Abs(this.price - someCar.price);
        }

        public static decimal PriceDifference(Car car1, Car car2)
        {
            return Math.Abs(car1.price - car2.price);
        }
    }
}
```

*We overladen hier de methode PriceDifference, wat geen probleem is gezien deze nieuwe versie 2 parameters verwacht.*

- Pas de code in de console applicatie aan als volgt:

```
namespace Prb.ClassesAndObjects.CONSOLE
{
    class Program
    {
        static void Main(string[] args)
        {
            ...
            Car thirdCar = new Car("Audi", "blauw", 25000M);

            //decimal costDifference = newCar.PriceDifference(thirdCar);
            decimal costDifference = Car.PriceDifference(newCar, thirdCar);

            ...
            Console.WriteLine($"Prijsverschil tussen {newCar.price} en {thirdCar.price} is {costDifference}");

            Console.ReadLine();
        }
    }
}
```



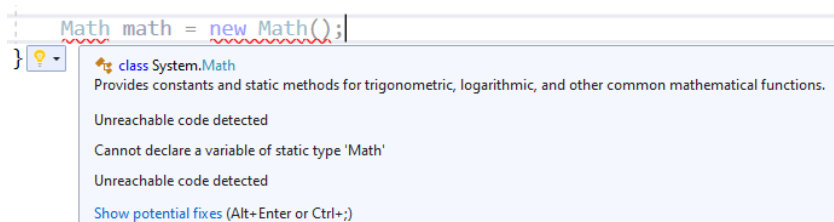
Met volgende  
(identieke) resultaat:

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Con
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
```

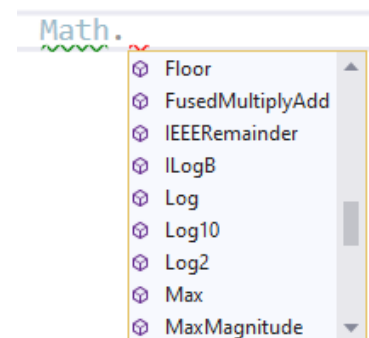
*In de praktijk gebruik je statische methoden niet vaak zoals in het voorbeeld dat hierboven uiteen werd gezet.*

*Je zal eerder klassen tegen komen (of zelf maken) die uit niets anders bestaan dan statische methoden. Bekijk even de .Net klasse die we zopas al even gebruikten, namelijk de klasse Math.*

*Van deze klasse is het zelfs niet mogelijk om een instantie aan te maken :*



*Deze klasse biedt alleen een hele batterij aan functionaliteiten aan (statische methoden) die dan ook rechtstreeks aanspreekbaar zijn.*

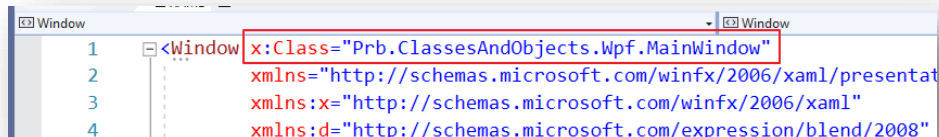


## 4.6 PARTIAL KLASSEN

Een klasse kan veel methoden, velden, constructors en andere items (members) 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**.

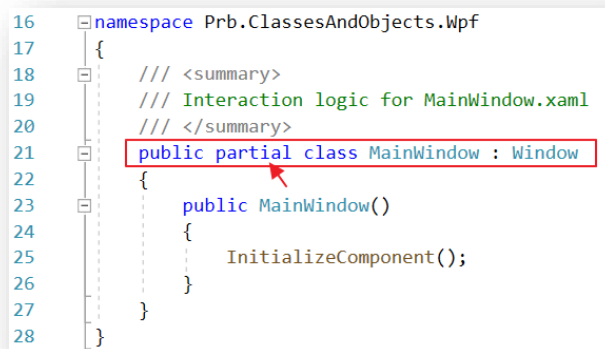
Dit principe wordt eigenlijk gebruikt in elke WPF toepassing of ASP.Net website die je maakt en heet **partial classes**.

- In je WPF toepassing zie je volgende code in de XAML-file:



```
1 <Window x:Class="Prb.ClassesAndObjects.Wpf.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

- In de Code-behind van dit venster zie je:



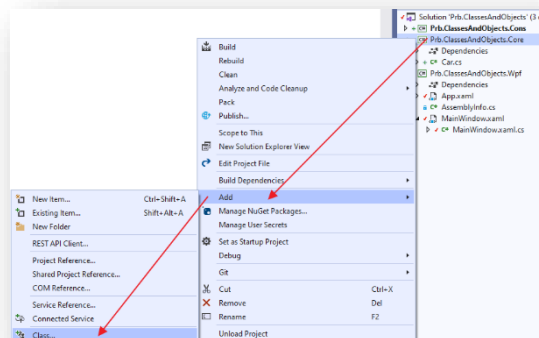
```
16 namespace Prb.ClassesAndObjects.Wpf
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26         }
27     }
28 }
```

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 venster voorziet is direct bruikbaar in de codebehind-file.

We kunnen dit toepassen voor de eerder gemaakte klasse Car.

- Voeg in je Class Library (Prb.ClassesAndObjects.Core) een nieuwe klasse toe
- Geef deze de naam CarExtension



- Wijzig de code in dit nieuwe bestand als volgt (vervang dus “class CarExtension” door “public partial class Car”):

```

Name: CarExtension
CarExtension.cs* x MainWindow.xaml* MainWindow.xaml.cs Microsoft.NET
Prb.ClassesAndObjects.Core
7 namespace Prb.ClassesAndObjects.Core
8 {
9     public partial class Car
10    {
11    }
12 }

```

- In de klasse Car zelf pas je ook de code aan als volgt (vervang dus “public class Car” door “public partial class Car”):

```

CarExtension.cs* Car.cs* x MainWindow.xaml* MainWindow.xaml
Prb.ClassesAndObjects.Core
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Prb.ClassesAndObjects.Core
8 {
9     public partial class Car
10    {
11        public string brand;
12        public string color;
13    }
14 }

```

Als voorbeeld **knippen** we onze 2 PriceDifference methoden weg uit onze originele Car klasse en **plakken** die in onze bijgekomen klasse (dus in het bestand CarExtension.cs):

```

CarExtension.cs* Car.cs* x MainWindow.xaml* MainWindow.xaml.cs Microsoft.NET.Sdk.DefaultItems.targets Main
Prb.ClassesAndObjects.Core
6
7 namespace Prb.ClassesAndObjects.Core
8 {
9     public partial class Car
10    {
11        public decimal PriceDifference(Car someCar)
12        {
13            return Math.Abs(this.price - someCar.price);
14        }
15        public static decimal PriceDifference(Car car1, Car car2)
16        {
17            return Math.Abs(car1.price - car2.price);
18        }
19    }
20 }

```

Na hercompileren merk je geen verschil: beide klassen werken als 1 klasse!

In onze console applicatie merk je dus zelfs niet dat onze klasse Car eigenlijk op de achtergrond over 2 verschillende bestanden verdeeld werd.

```

MainWindow.xaml* MainWindow.xaml.cs Microsoft.NET.Sdk.DefaultItems.targets MainWindow.g.cs Program.cs x
Prb.ClassesAndObjects.Cons.Program
secondCar.color = "wit";
secondCar.brand = "Ford";
secondCar.price = 9000M;

Car thirdCar = new Car("Audi", "blauw", 25000M);

//decimal costDifference = new Car().PriceDifference(thirdCar);
decimal costDifference = Car.PriceDifference(newCar, thirdCar);

```

## 4.7 BESLUITEN TOT DUSVER

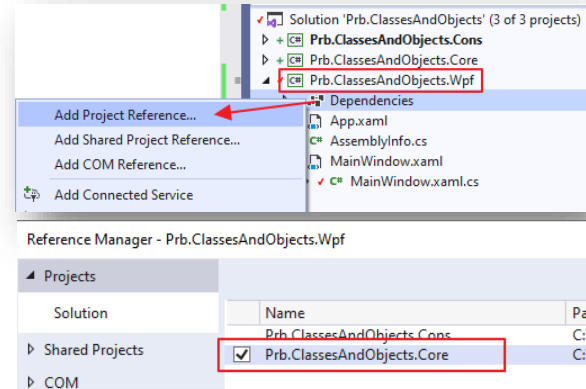
---

- 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 hieronder) en **methodes**. Deze worden **members** genoemd.
- Een **constructor** wordt uitgevoerd tijdens de instantiëring 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,...)
- Met **constructor chaining** kan de ene versie van de constructor bestaande functionaliteiten uit een andere versie van de constructor laten uitvoeren zodat deze niet in elke constructor herhaald te dienen worden.
- **Static** members (velden, methoden, eigenschappen) worden door elk object van een klasse gedeeld.

## 5 GEBRUIK VAN DE KLASSENBIBLIOTHEEK

We hebben reeds een klassenbibliotheek aangemaakt (Prb.ClassesAndObjects.CORE) en hebben deze bibliotheek gebruikt in onze Console applicatie. We gebruiken deze klassenbibliotheek ook voor onze WPF applicatie.

- We maken een referentie vanuit de WPF applicatie naar de klassenbibliotheek zodat we de klasse Car ook kunnen gebruiken in de WPF applicatie:



- Plaats bovenaan in je code behind een using directive die naar de namespace van onze class library verwijst :

```
15 using Prb.ClassesAndObjects.Core;
16
17 namespace Prb.ClassesAndObjects.Wpf
18 {
19     /// <summary>
20     /// Interaction logic for MainWindow.xaml
21     /// </summary>
22     public partial class MainWindow : Window
23     {
24         public MainWindow()
25         {
26             InitializeComponent();
27         }
28     }
29 }
```

- Voorzie een variabele die alle objecten van het type Car zal bijhouden in de code-behind:

```
namespace Prb.ClassesAndObjects.Wpf
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        List<Car> cars;
        public MainWindow()
        {
            InitializeComponent();
            cars = new List<Car>();
        }
    }
}
```

- Wanneer er op de knop Voeg auto toe gedrukt wordt, dienen de textboxes uitgelezen te worden en wordt een nieuw object van het type Car 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 (List) van auto's toegevoegd en de listbox met auto's wordt opgevuld met alle auto's in deze lijst (List):

```

private void btnAddNewCar_Click(object sender, RoutedEventArgs e)
{
    string color = txtColor.Text.Trim();
    string carbrand = txtCarBrand.Text.Trim();
    decimal.TryParse(txtPrice.Text.Trim(), out decimal price);

    Car car = new Car();
    car.color = color;
    car.brand = carbrand;
    car.price = price;

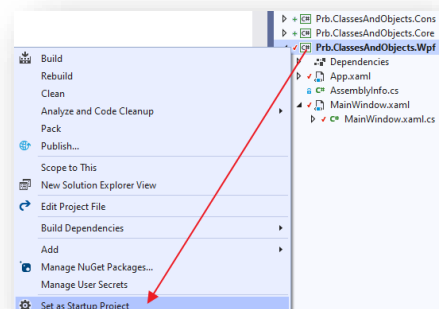
    // OF via constructor:
    //Car car = new Car(carbrand, color, price);

    cars.Add(car);
    UpdateCarListbox();
}
private void UpdateCarListbox()
{
    ///3 mogelijke manieren om de listbox te vullen met de List:

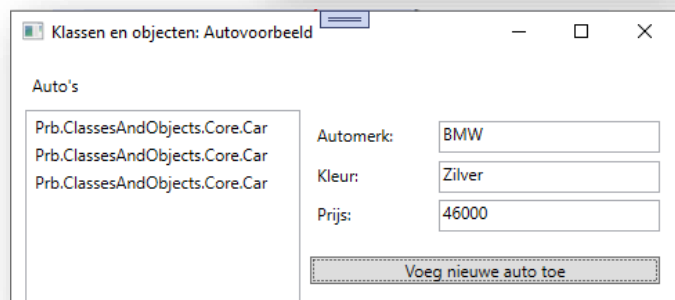
    /// #1
    // lstCars.ItemsSource = cars;
    // lstCars.Items.Refresh();
    /// #2
    // lstCars.ItemsSource = null;
    // lstCars.ItemsSource = cars;
    /// #3
    ///
    lstCars.Items.Clear();
    foreach(Car car in cars)
    {
        lstCars.Items.Add(car);
    }
}

```

Opgepast : vergeet niet om nu je WPF project in te stellen als startup project !



Wanneer we één of meerdere auto's toevoegen, zien we volgend (waarschijnlijk onverwacht) 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 (hier dus Prb.ClassesAndObjects. Core.Car). Dit is voor een gebruiker uiteraard weinig verhelderend.

## 5.1 OVERRIDE STRING TOSTRING()

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 in de klasse Car 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

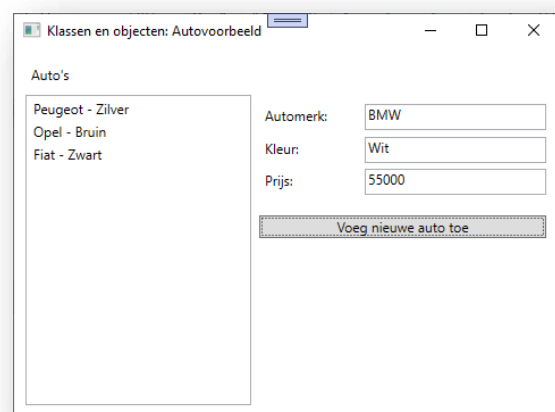
In de methode stellen we dan de tekst samen die we willen tonen op basis van één of meerdere velden (of eigenschappen, zie verder). Deze tekst wordt dan geretourneerd.

Pas de code in de klasse Car (je beslist zelf in welk van de 2 files je deze aanpassing doet) als volgt aan (onderstaande code plaatsten we in het bestand "CarExtension.cs" maar je kan het evengoed toevoegen aan "Car.cs") :

```
namespace Prb.ClassesAndObjects.Core
{
    public partial class Car
    {
        public override string ToString()
        {
            return $"{brand} - {color}";
        }
        public decimal PriceDifference(Car someCar)
        {
            return Math.Abs(this.price - someCar.price);
        }
        public static decimal PriceDifference(Car car1, Car car2)
        {
            return Math.Abs(car1.price - car2.price);
        }
    }
}
```

Via de override ToString() methode geven we aan dat, wanneer we een object van het type Car willen afbeelden, we de inhoud van de velden brand en color zullen tonen.

Voeg enkele auto's toe in de WPF-applicatie en je zult zien dat de listbox met auto's nu als volgt getoond wordt:



## 5.2 EEN OBJECT CASTEN

We zullen er nu nog voor zorgen dat wanneer we in de listbox een auto selecteren, de gegevens van de geselecteerde auto meteen in de corresponderende tekstvakken terecht komt.

Creëer een event-handler op de listbox (SelectionChanged).

Pas je code in de WPF code-behind als volgt aan:

```
private void lstCars_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lstCars.SelectedItem != null)
    {
        Car car = (Car)lstCars.SelectedItem;
        txtCarBrand.Text = car.brand;
        txtColor.Text = car.color;
        txtPrice.Text = car.price.ToString();
    }
}
```

Wanneer de selectie verandert in de listbox dan kijken we eerst na of er wel degelijk een element geselecteerd werd (het event kan zich immers ook voordoen zonder dat een item geselecteerd staat). We laten de code dus enkel uitvoeren wanneer we zeker zijn dat er wel degelijk iets geselecteerd werd.

Via de .SelectedItem eigenschap van de listbox krijgen we het item dat in de listbox geselecteerd werd, alleen, dit is altijd een object (ongeacht van wat je gebruikte om de listbox te vullen : strings, ints, cars ...).

Het geselecteerde item dienen we dus eerst te **casten** naar het type Car : geen probleem, want in de listbox zitten enkel Car objecten.

We hoeven hier geen gebruik te maken van het new keyword omdat car reeds bestaat als object in het geheugen (dit object zit al in de List Cars). We maken dus een referentie vanuit car naar het effectieve object in het geheugen. We komen hier nog op terug in het hoofdstuk Value & references in de module Programming Advanced.

Via het object car vullen we alle textboxen in aan de hand van de fields van het Car-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 textboxen ingevuld worden met de specificaties (fields) van de geselecteerde auto.



## 6 ANONIEME KLASSE

Iets waar je in de cursus (waarschijnlijk) niet meer (rechtstreeks) in aanraking mee zult komen, maar waar je verder in je opleiding vaak mee zult werken, zijn zogenaamde anonieme klassen. In sommige toepassingen binnen .Net, zoals Linq worden ze intensief gebruikt.

Een anonieme klasse is een klasse die (zoals je wellicht vermoedt) geen naam heeft.

Om al een idee te hebben waarover het gaat, werken we in onze CONSOLE applicatie (vergeet deze niet terug als startup in te stellen om de code uit te testen) een eenvoudig voorbeeld.

- Voeg helemaal onderaan je console applicatie volgende code toe :

```
namespace Prb.ClassesAndObjects.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...
            Console.WriteLine($"Prijsverschil tussen {newCar.price} en {thirdCar.price} is {costDifference}");

            var anonymous = new { Name = "Peter", Age = 40 };
            Console.WriteLine("=====");
            Console.WriteLine($"Anonymous heet {anonymous.Name}, is {anonymous.Age} jaar oud en is van het type {anonymous.GetType().Name}");

            Console.ReadLine();        }
    }
}
```

- Na uitvoeren van de code krijgen we volgend resultaat:

```
C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld01\Start\Prb.ClassesAndObjects.Cons\bin\Debug\net
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
=====
Anonymous heet Peter, is 40 jaar oud en is van het type <>f__AnonymousType0`2
```

We maken hier dus als het ware “on the fly” een object aan van een klasse die niet bestaat.

Het is een (nog niet) bestaande klasse, zonder naam, vandaar de term anonieme klasse.

Omdat de klasse (nog) niet bestaat, kan je om het even welke velden (eigenlijk zijn het eigenschappen, zie verder) beschrijven en er waarden aan toekennen tussen de accolades. Variabelen (hier dus anonymous) die je hiervoor gebruikt dienen steeds van het type **var** te zijn.

Het echte nut hiervan zal je dus ontdekken verder in je opleiding.



### Code repository

De broncode van dit cursusvoorbeeld tot dusver is te vinden op

 `git clone` <https://github.com/howest-gp-prb/cu-h9-KlassenEnObjecten-einde>

## 7 PROPERTIES (EIGENSCHAPPEN) : INLEIDING

In het eerste gedeelte van dit hoofdstuk heb je kennisgemaakt met instantievariabelen (velden). 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.

### 7.1 WAT ZIJN PROPERTIES

Properties of eigenschappen stellen de ontwikkelaar in staat om data af te schermen van oneigenlijk gebruik of verkeerd gebruik: we spreken over **data encapsulation**.

Properties en fields (die we eerder zagen) worden in het begin vaak verward.

Beiden hebben ogenschijnlijk hetzelfde effect/functie/gedrag, maar (publieke) properties zijn veel krachtiger dan (publieke) fields : we zullen in de toekomst dan ook quasi altijd de voorkeur geven aan (publieke) properties boven (publieke) fields.

Opgelet : we zeggen dus wel degelijk dat we het gebruik van (publieke) properties zullen verkiezen boven het gebruik van (publieke) fields : private fields daarentegen zullen belangrijk blijven binnen het verhaal van de properties maar hier zometeen meer over.

### 7.2 PRAKTISCH VOORBEELD

- Open de klasse `Car` in de `Prb.ClassesAndObjects.Core` class library.
- Voeg volgende code (field) toe:

```
namespace Prb.ClassesAndObjects.Core
{
    public partial class Car
    {
        public string brand;
        public string color;
        public decimal price;
        public static int carCount;
        public int topSpeed;

        public Car() // Default constructor
        {
            brand = null;
            color = null;
            price = 0;
            carCount++;
        }
        ...
    }
}
```

- Open het `Program.cs` van de console applicatie project `Prb.ClassesAndObjects.Cons`
- Voeg volgende code toe:

```

static void Main(string[] args)
{
    Car newCar;
    newCar = new Car();

    newCar.color = "zwart";
    newCar.brand = "Hyundai";
    newCar.price = 15000M;
    newCar.topSpeed = -190;

    Car secondCar;

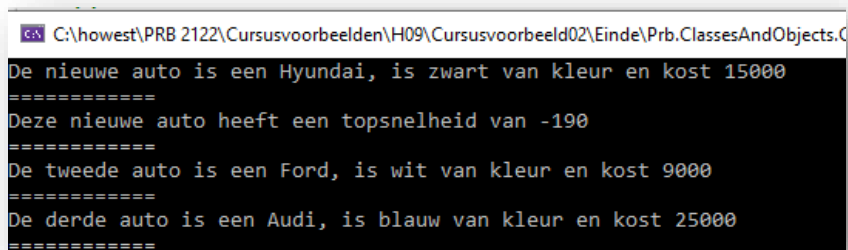
    ...

    Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van kleur
en kost {newCar.price}");
    Console.WriteLine("=====");
    Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van {newCar.topSpeed}");
    Console.WriteLine("=====");

    ...
}

```

Met volgend resultaat:



```

C:\howest\PRB 2122\Cursusvoorbeelden\H09\Cursusvoorbeeld02\Einde\Prb.ClassesAndObjects.C
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
=====

```

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 verkeerdelijk gebruikt wordt. We bespreken enkele mogelijkheden, de één al wat omslachtiger dan de andere.

## 8 DATA AFSCHERMEN MET EEN METHODE (GETTERS & SETTERS)

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.

- Voorzie volgende wijzigingen in de klasse `Auto`.

**We veranderen hier dus het publieke veld (field) `topSpeed` in een private veld (field) `topSpeed`**

:

```
namespace Prb.Properties.Core
{
    Public partial class Car
    {
        public string brand;
        public string color;
        public decimal price;
        public static int carCount;
        private int topSpeed;

        public void SetTopSpeed(int topSpeed)
        {
            if (topSpeed < 0)
                this.topSpeed = 0;
            else
                this.topSpeed = topSpeed;
        }
    }
}
```

Omdat we nu geen publiek veld (field) `topSpeed` meer hebben kunnen we de topsnelheid van onze wagen nu enkel nog instellen via de zonet gecreëerde methode `SetTopSpeed()`.

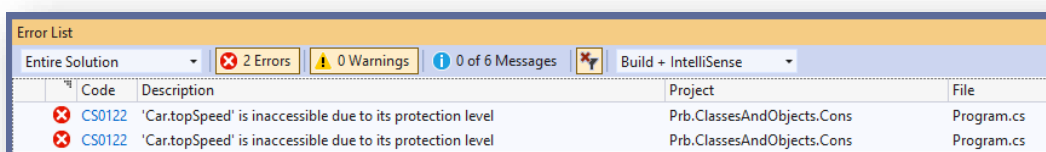
Wanneer de opgegeven topsnelheid die via een argument doorgegeven wordt aan de methode **`SetTopSpeed`** kleiner is dan 0, dan stellen we die in op 0.

*Een andere mogelijkheid zou kunnen zijn dat we een `Exception` gooien met de melding dat de snelheid niet negatief mag zijn.*

```
public void SetTopSpeed(int topSpeed)
{
    if (topSpeed < 0)
        //this.topSpeed = 0;
        throw new Exception("Topsnelheid kan niet kleiner zijn dan 0");
    else
        this.topSpeed = topSpeed;
}
```

*Bovenstaande code is ter illustratie : pas je eigen programma NIET aan.*

Wanneer we na deze wijzigingen onze solution opnieuw builden, krijgen we compileerfouten:



Error List			
Entire Solution			
2 Errors 0 Warnings 0 of 6 Messages Build + IntelliSense			
	Code	Description	File
✖	CS0122	'Car.topSpeed' is inaccessible due to its protection level	Prb.ClassesAndObjects.Cons
✖	CS0122	'Car.topSpeed' is inaccessible due to its protection level	Program.cs

Aangezien we het field `topSpeed` in de klasse `Car` hebben gewijzigd van `public` naar `private` is dit field niet meer toegankelijk vanuit onze console applicatie:

```
newCar.price = 15000M;
newCar.topSpeed = -190;

Car secondCar;
secondCar = new Car();
secondCar.color = "wit";
secondCar.brand = "Ford";
secondCar.price = 9000M;

Car thirdCar = new Car("Audi", "blauw", 25000M);

//decimal costDifference = newCar.PriceDifference(thirdCar);
decimal costDifference = Car.PriceDifference(newCar, thirdCar);

Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van kleur");
Console.WriteLine("=====");
Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van {newCar.topSpeed}");
Console.WriteLine("=====");
```

- Om in onze console applicatie de `topSpeed` in te stellen van een `car` object passen we volgende code aan:

```
namespace Prb.ClassesAndObjects.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Car newCar;
            newCar = new Car();

            newCar.color = "zwart";
            newCar.brand = "Hyundai";
            newCar.price = 15000M;
            newCar.SetTopSpeed(-190);
            ...
        }
    }
}
```

We stellen nu de `topSpeed` in via de methode `SetTopSpeed`.

We hebben nog altijd één compileerfout; omdat het field `topSpeed` `private` is kunnen we dit ook niet meer opvragen.

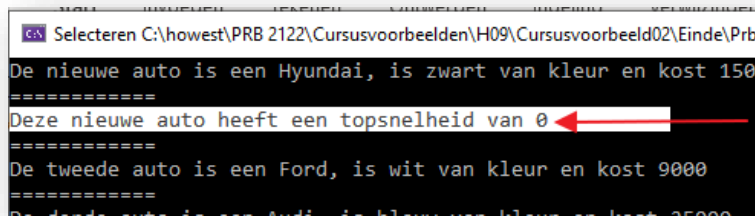
- Omdat het veld (field) `topSpeed` nu `privaat` is kan het ook niet meer zomaar gelezen worden van buiten de klasse. Voorzie daarom de methode `GetTopSpeed()` in de klasse `Car`:

```
namespace Prb.ClassesAndObjects.Core
{
    public partial class Car
    {
        ...
        public void SetTopSpeed(int topSpeed)
        {
            if (topSpeed < 0)
                this.topSpeed = 0;
            else
                this.topSpeed = topSpeed;
        }
        public int GetTopSpeed()
        {
            return topSpeed;
        }
        ...
    }
}
```

- Wijzig in Program.cs volgende code:

```
namespace Prb.Properties.CONST
{
    class Program
    {
        static void Main(string[] args)
        {
            ...
            Console.WriteLine($"De nieuwe auto is een {newCar.brand}, is {newCar.color} van kleur
en kost {newCar.price}");
            Console.WriteLine("=====");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van
{newCar.GetTopSpeed()}");
            Console.WriteLine("=====");
            ...
        }
    }
}
```

- Run de console applicatie en we zien dat de topsnelheid nu weergegeven wordt als 0:



## 8.1 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 wel een beetje **omslachtig**:
  - Je moet voor elk (privaat) field twee methoden toevoegen
  - Om een waarde op te vragen moet je steeds een methodeaanroep uitvoeren:
    - `newCar.topSpeed` was eigenlijk toch gemakkelijker dan `newCar.GetTopSpeed()`
  - Om een waarde aan te passen, moet je ook een methodeaanroep doen:
    - `newCar.topSpeed = -190;`  
werd  
`newCar.SetTopSpeed(-190);`

## 9 DATA AFSCHERMEN MET EEN PROPERTY

### 9.1 EEN EERSTE PROP MAKEN

Een ideale oplossing voor dit probleem vormen properties of eigenschappen: je hebt de afscherming zoals met methoden (de getter en setter van daarnet), maar naar de buitenwereld toe (dus buiten de klasse) kan je de data manipuleren zoals een publiek veld (field).

- Vervang de methoden `GetTopSpeed` en `SetTopSpeed` in de klasse `Car` door onderstaande code (of zet beide methoden in commentaar) :

```
namespace Prb.Properties.Core
{
    public class Car
    {
        public string brand;
        public string color;
        public decimal price;
        public static int carCount;
        private int topSpeed;

        public int topSpeed
        {
            get { return topSpeed; }
            set
            {
                if (value < 0)
                    topSpeed = 0;
                else
                    topSpeed = value;
            }
        }

        //public void SetTopSpeed(int topSpeed)
        //{
        //    if (topSpeed < 0)
        //        this.topSpeed = 0;
        //    //throw new Exception("Topsnelheid kan niet kleiner zijn dan 0");
        //    else
        //        this.topSpeed = topSpeed;
        //}
        //public int GetTopSpeed()
        //{
        //    return topSpeed;
        //}
    }
}
```

Hier hebben we de eigenschap/property `topSpeed` 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 ; de property is dan **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 zou moeten worden. Laat je dit onderdeel weg, dan kan de eigenschap niet gebruikt worden om een waarde te veranderen; de property is dan **read-only**.

De naam van een privaat veld (field) begint met een **kleine letter**, die van properties met een **Hoofdletter**.

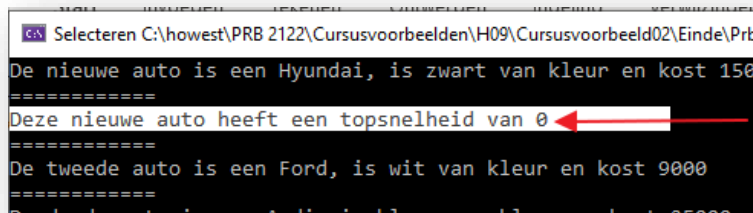
Opgepast, het private veld (field) `topSpeed` is hier nog steeds nodig, de property `TopSpeed` gaat dit (private) field enkel afschermen tegen oneigenlijk gebruik.

- Pas de code als volgt aan in `Program.cs` om de builderrors aan te pakken:

```
namespace Prb.Properties.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            ...
            newCar.price = 15000M;
            newCar.TopSpeed = -190;
            ...

            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van {newCar.TopSpeed}");
            Console.WriteLine("=====");
            ...
        }
    }
}
```

- Run de console applicatie en we zien dat de topsnelheid ook nu weergegeven wordt als 0:



```
De nieuwe auto is een Hyundai, is zwart van kleur en kost 15000M
=====
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
```



#### 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.

### Conclusie

We hebben nu dezelfde functionaliteit, maar dan zonder dat we extra methodes moeten aanmaken!



## 9.2 GOOD PRACTICE

### Vanaf nu gebruik je 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; }  
}
```

## 9.3 PROPERTIES : VERKORTE NOTATIE

Bij het gebruik van een private field en public property zou men in feite bovenstaande code (paragraaf 9.2) 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 codefragement maakt de property *Waarde* aan. Intern wordt deze verkorte schrijfwijze vertaald tot het voorbeeld erboven (paragraaf 9.2): het private field en de get en set-implementatie van de property worden dus automatisch voorzien. Het field *waarde* (de private variabele dus) is niet zichtbaar.

De property kan even eenvoudig worden aangemaakt als een field, maar je beschikt over een property die je later kan uitbreiden zonder de notatie in bestaande code te moeten aanpassen.

Starten met een field, en later overschakelen naar een property kan omslachtiger zijn.



#### Tip

Voor het aanmaken van een verkorte property kun je de **code-snippet propfull** gebruiken. Na ingeven van **prop** en **2 \* de tab-toets** indrukken, zorgt die voor een automatische aanmaak van een property met get en set accessor.


## 9.4 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 (je hoeft dit in je applicatie niet aan te maken):

```
public class Calendar
{
    public DateTime Date { get; set; }

    public Calendar () // Default constructor
    {
        Date = 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 Date toch nullable te maken, dan kunnen we dit doen door na het datatype (DateTime) een **vraagteken (?)** te plaatsen:

```
public DateTime? Date { get; set; }
```

Een ander voorbeeld:

```
public int? NumberOfDays { get; set; }
```

## 9.5 AUTO-PROPERTY INITIALIZER

Soms wensen we een property reeds een standaard waarde te geven. Bekijk onderstaand voorbeeld (opnieuw, je hoeft dit niet op te nemen in je applicatie):

```
public class PepperoniPizza
{
    public decimal ExtraPrice { get; set; }

    public PepperoniPizza()
    {
        ExtraPrice = 0.25m;
    }
}
```

In de constructor van deze klasse initialiseren we de property *ExtraPrice* met een waarde van 0,25. Dus elk geïntanceerd object van deze klasse zal een property *ExtraPrice* hebben met een waarde van 0,25.

Sinds C# 6.0 kunnen we dit in deze verkorte versie schrijven:

```
public class PepperoniPizza
{
    public decimal ExtraPrice { 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. Uiteraard kan je deze eigenschap – omwille van de get en set - via het object nog een andere waarde geven (het is dus GEEN constante of GEEN read-only waarde, het gaat hier enkel om een initiele waardetoekenning).

## 9.6 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 ExtraPrice { get; } = 0.25m;
}
```

De property *ExtraPrice* 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.

Een andere manier om tot hetzelfde resultaat te komen is om het set gedeelte privaat te maken :

```
public class PepperoniPizza
{
    public decimal ExtraPrice { get; private set; } = 0.25m;
}
```

Welke manier je ook gebruikt, aan *ExtraPrice* zal je binnen de klasse (eventueel) een andere waarde kunnen toekennen, maar buiten de klasse (het object) zal het niet mogelijk zijn om de waarde rechtstreeks te wijzigen (tenzij je uiteraard in een constructor a.d.h.v. argumenten alsnog de waarde ervan laat instellen).

## 10 AANPASSEN VAN ONZE VOORBEELD APPLICATIE

Aangezien we hier beter kunnen werken met automatische properties zullen we de andere public fields (brand, color, price en carCount) van de klasse Auto ook aanpassen.

- Wijzig en vervang volgende code in de klasse Car:

```
namespace Prb.ClassesAndObjects.Core
{
    public partial class Car
    {
        //public string brand;
        //public string color;
        //public decimal price;
        //public static int carCount;

        public string Brand { get; set; }
        public string Color { get; set; }
        public decimal Price { get; set; }
        public static int CarCount { get; set; }

        private int topSpeed;
        public int TopSpeed
        {
            get { return topSpeed; }
            set
            {
                if (value < 0)
                    topSpeed = 0;
                else
                    topSpeed = value;
            }
        }
        ...

        public Car() // Default constructor
        {
            //brand = null;
            //color = null;
            //price = 0;
            //carCount++;
            Brand = null;
            Color = null;
            Price = 0;
            CarCount++;
        }
        public Car(string brand) : this() // Constructor ontvangt 1 parameter
        {
            //this.brand = brand;
            Brand = brand;
        }
        public Car(string brand, string color) : this() // Constructor ontvangt 2 parameters
        {
            //this.brand = brand;
            //this.color = color;
            Brand = brand;
            Color = color;
        }
        public Car(string brand, string color, decimal price) : this() // Constructor ontvangt 3
parameters
        {
            //this.brand = brand;
            //this.color = color;
            //this.price = price;
            Brand = brand;
            Color = color;
            Price = price;
        }
        ...
    }
}
```

Aangezien we nog steeds een negatieve topsnelheid willen opvangen, kunnen we de property `TopSpeed` niet in zijn verkorte versie schrijven (`public int TopSpeed { get; set; }`). Deze property heeft nog steeds een zichtbaar private field nodig.

- De klasse `Car` is over 2 bestanden verspreid (ter herinnering : partial class). In de klasse `Car` in het bestand `CarExtensions.cs` 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 Prb.ClassesAndObjects.Core
{
    public partial class Car
    {
        public override string ToString()
        {
            return $"{Brand} - {Color}";
        }
        public decimal PriceDifference(Car someCar)
        {
            return Math.Abs(this.Price - someCar.Price);
        }
        public static decimal PriceDifference(Car car1, Car car2)
        {
            return Math.Abs(car1.Price - car2.Price);
        }
    }
}
```

- Uiteraard dienen we ook onze console applicatie aan te passen. Wijzig onderstaande code in `Program.cs`:

```
namespace Prb.ClassesAndObjects.Cons
{
    class Program
    {
        static void Main(string[] args)
        {
            Car newCar;
            newCar = new Car();

            newCar.Color = "zwart";
            newCar.Brand = "Hyundai";
            newCar.Price = 15000M;
            newCar.TopSpeed = -190;

            Car secondCar;
            secondCar = new Car();
            secondCar.Color = "wit";
            secondCar.Brand = "Ford";
            secondCar.Price = 9000M;

            Car thirdCar = new Car("Audi", "blauw", 25000M);

            //decimal costDifference = newCar.PriceDifference(thirdCar);
            decimal costDifference = Car.PriceDifference(newCar, thirdCar);

            Console.WriteLine($"De nieuwe auto is een {newCar.Brand}, is {newCar.Color} van
            kleur en kost {newCar.Price}");
            Console.WriteLine("=====");
            Console.WriteLine($"Deze nieuwe auto heeft een topsnelheid van
            {newCar.TopSpeed}");
            Console.WriteLine("=====");

            Console.WriteLine($"De tweede auto is een {secondCar.Brand}, is {secondCar.Color}
            van kleur en kost {secondCar.Price}");
            Console.WriteLine("=====");
            Console.WriteLine($"De derde auto is een {thirdCar.Brand}, is {thirdCar.Color} van
            kleur en kost {thirdCar.Price}");
        }
    }
}
```

```

        Console.WriteLine("=====");
        Console.WriteLine($"We hebben nu {Car.CarCount} auto's");

        Console.WriteLine("=====");
        Console.WriteLine($"Prijsverschil tussen {newCar.Price} en {thirdCar.Price} is {costDifference}");

        var anonymous = new { Name = "Peter", Age = 40 };
        Console.WriteLine("=====");
        Console.WriteLine($"Anonymous heet {anonymous.Name}, is {anonymous.Age} jaar oud en is van het type {anonymous.GetType().Name}");

        Console.ReadLine();
    }
}

```

- Ook in onze WPF applicatie dienen we aanpassingen te maken. Doe de nodige wijzigingen in de 2 hieronder afgebeelde event handlers :

```

private void BtnAddNewCar_Click(object sender, RoutedEventArgs e)
{
    string color = txtColor.Text.Trim();
    string carbrand = txtCarBrand.Text.Trim();
    decimal.TryParse(txtPrice.Text.Trim(), out decimal price);

    Car car = new Car();
    car.Color = color;
    car.Brand = carbrand;
    car.Price = price;

    // OF via constructor:
    //Car car = new Car(carbrand, color, price);

    cars.Add(car);
    UpdateCarListbox();
}

```

```

private void LstCars_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lstCars.SelectedItem != null)
    {
        Car car = (Car)lstCars.SelectedItem;
        txtCarBrand.Text = car.Brand;
        txtColor.Text = car.Color;
        txtPrice.Text = car.Price.ToString();
    }
}

```



### Code repository

De broncode van dit hoofdstuk tot nu toe is te vinden op

git clone <https://github.com/howest-gp-prb/cu-h9-Properties-einde.git>

## 11 COMPOSITIE

Je kan voor deze paragraaf best even je Visual Studio afsluiten en een nieuwe repo binnenhalen (zie hieronder). De nieuwe repo bevat min of meer dezelfde code als deze die we tot nu toe gemaakt hebben, maar is wat afgeslankt om het volgende onderwerp duidelijk aan bod te laten komen :



### Code repository

De broncode van de startsituatie van deze paragraaf kan je hier terug vinden

 `git clone https://github.com/howest-gp-prb/cu-h9-Compositie-start.git`

Met compositie bedoelen we een object in een ander object gebruiken.

We leerden al dat, wanneer we een klasse maken (bv Car), we deze klasse kunnen gebruiken als een type.

We konden in onze code behind bijvoorbeeld schrijven :

```
Car car = new Car();
```

We maken hier dus een variabele aan (car met kleine c) die van het type Car (met hoofdletter C) is.

Behalve het feit dat we deze dienen te instantiëren (= new Car() ) vooraleer we die kunnen gebruiken, is er eigenlijk niets nieuws : we maakten al vele variabelen aan van het type string, int, decimal ...

In de vorige paragraaf leerden we werken met properties in een klasse : bij elke property dienden we te vermelden welk datatype deze heeft :

```
public string Brand { get; set; }
public string Color { get; set; }
public decimal Price { get; set; }
```

Niets belet ons om voor een property een datatype te kiezen die een klasse is die we zelf gemaakt hebben.

Maak in je class library (Prb.ClassesAndObjects.Core) een nieuwe klasse aan met als naam **CarType**.

Neem onderstaande (eenvoudige) code over in deze klasse :

```
namespace Prb.ClassesAndObjects.Core
{
    public class CarType
    {
        public string TypeName { get; set; }
        public bool IsCommercial { get; set; }
        public CarType(string typeName, bool isCommercial)
        {
            TypeName = typeName;
            IsCommercial = isCommercial;
        }
        public override string ToString()
        {
            string symbol = "NC"; // NC = not commercial
            if (IsCommercial)
                symbol = "C"; // C = commercial;
            return $"{TypeName} - {symbol}";
        }
    }
}
```

Een klasse dus met 2 properties (TypeName en IsCommercial), met 1 constructor die de waarde van de property TypeName en IsCommercial ontvangt en aan de properties toekent, en tenslotte een override ToString() die de waarde van de beide eigenschappen afbeeldt. Niets nieuw dus.

Neem terug je klasse Car er even bij.

Voeg onderstaande property toe aan de reeds aanwezige properties :

```
...  
public string Brand { get; set; }  
public string Color { get; set; }  
public decimal Price { get; set; }  
public CarType CarType { get; set; }  
...
```

We hebben hier dus een extra property aan toegevoegd van het type CarType (in plaats van een string, een decimal, een int ...).

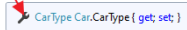
De prop zelf hebben we—ook al hoeft dit niet noodzakelijk—een naam gegeven die dezelfde is als de klasse. In Visual Studio zelf zie je aan de hand van het kleurverschil in de tekst beter het onderscheid tussen het type (CarType) en de property name (CarType) :

```
public CarType CarType { get; set; }
```

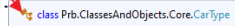
Er zal geen conflict ontstaan wanneer we het type en de property op dezelfde manier schrijven : de compiler zal duidelijk het onderscheid kunnen maken.

Waar nodig zal de intellisense van Visual Studio je trouwens helpen (let op de verschillende icoontjes) :

De eigenschap CarType      `public CarType CarType { get; set; }`



Het type CarType      `public CarType CarType { get; set; }`



Wanneer we een nieuw car-object zullen aanmaken (en we alle properties willen vullen) dan zullen we naast de (klassieke) string-waarden (Brand, Color), int-waarden (TopSpeed) en decimal-waarden (Price) nu ook een CarType-waarde moeten toekennen.

Dit betekent dat we—vooraleer we aan de property CarType van een Car-object een waarde kunnen toekennen—over een CarType-object zullen moeten beschikken. We demonstreren hoe je dit kunt doen terug aan de hand van een voorbeeld.

- Ga naar de code-behind van je WPF venster.
- Voeg helemaal bovenaan (naast de bestaande List<Car>) een tweede List toe; deze keer een List<CarType>:

```
public partial class MainWindow : Window  
{  
    List<Car> cars;  
    List<CarType> carTypes;  
    ...  
}
```



- We vullen deze nieuwe List tijdens het opstarten van ons programma manueel met een aantal types. Voeg aan de constructor van het venster (public MainWindow() ) onderstaande code toe :

```
public MainWindow()
{
    InitializeComponent();
    cars = new List<Car>();
    carTypes = new List<CarType>();
    carTypes.Add(new CarType("Sedan", false));
    carTypes.Add(new CarType("Break", false));
    carTypes.Add(new CarType("SUV", false));
    carTypes.Add(new CarType("SUV", true));
    carTypes.Add(new CarType("Van", true));
}
```

We instantiëren dus onze nieuwe List carTypes en we vullen die meteen met een aantal CarType-objecten. We doen dit op een verkorte manier. De code hierboven hadden we bijvoorbeeld ook als volgt kunnen schrijven (wat we niet gaan doen) :

```
CarType carType1 = new CarType("Sedan", false);
carTypes.Add(carType1);
CarType carType2 = new CarType("Sedan", false);
carTypes.Add(carType2);
...
```

Of ook nog als volgt (wat we ook niet gaan doen) :

```
CarType carType = new CarType("Sedan", false);
carTypes.Add(carType);
carType = new CarType("Sedan", false);
carTypes.Add(carType);
...
```

Wat nog een kortere manier van schrijven is – met exact hetzelfde resultaat – is het volgende :

```
public MainWindow()
{
    InitializeComponent();
    carTypes = new List<CarType>() {
        new CarType("Sedan", false),
        new CarType("Break", false),
        new CarType("SUV", false),
        new CarType("SUV", true),
        new CarType("Van", true)
    };
}
```

- In onze XAML code bevindt zich nu ook een combobox (cmbCarType) die we (net zoals we onze ListBox vulden) zullen vullen met deze CartType-objecten. Voeg aan de constructor van het venster (public MainWindow() ) onderstaande code toe :

```
public MainWindow()
{
    InitializeComponent();
    cars = new List<Car>();
    carTypes = new List<CarType>();
    carTypes.Add(new CarType("Sedan", false));
    carTypes.Add(new CarType("Break", false));
    carTypes.Add(new CarType("SUV", false));
    carTypes.Add(new CarType("SUV", true));
    carTypes.Add(new CarType("Van", true));
}
```

```

    cmbCarType.ItemsSource = carTypes;
}

```

Of:

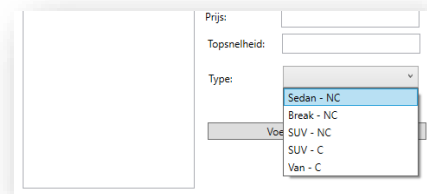
```

public MainWindow()
{
    InitializeComponent();
    carTypes = new List<CarType>() {
        new CarType("Sedan", false),
        new CarType("Break", false),
        new CarType("SUV", false),
        new CarType("SUV", true),
        new CarType("Van", true)
    };

    cmbCarType.ItemsSource = carTypes;
}

```

Test even je programma uit : de combobox zal nu inderdaad gevuld zijn met onze verschillende CarType-objecten.



- Wanneer een nieuwe wagen (een nieuw Car-object) wordt aangemaakt, dienen we nu ook de eigenschap CarType van het Car-object in te vullen.

Wijzig de code van de event-handler BtnAddNewCar\_Click als volgt:

```

private void BtnAddNewCar_Click(object sender, RoutedEventArgs e)
{
    if(cmbCarType.SelectedItem == null)
    {
        MessageBox.Show("Geef een type op", "Fout", MessageBoxButton.OK,
        MessageBoxImage.Error);
        cmbCarType.Focus();
        return;
    }
    string color = txtColor.Text.Trim();
    string carbrand = txtCarBrand.Text.Trim();
    decimal.TryParse(txtPrice.Text.Trim(), out decimal price);
    int.TryParse(txtTopSpeed.Text.Trim(), out int topSpeed);
    CarType carType = (CarType)cmbCarType.SelectedItem;
    Car car = new Car();
    car.Color = color;
    car.Brand = carbrand;
    car.Price = price;
    car.TopSpeed = topSpeed;
    car.CarType = carType;
    cars.Add(car);
    UpdateCarListbox();
}

```

Als we nu een nieuwe wagen wensen toe te voegen, willen we er zeker van zijn dat er een type in de combobox werd geselecteerd. We kijken dus eerst na of er wel iets is geselecteerd, en is dat niet zo dan tonen we een foutmelding en gaan we gewoon niet verder.

Vervolgens vragen we op welk type er geselecteerd werd : cmbCarType.SelectedItem.

Hoewel onze combobox gevuld is met CarType-objecten moeten we er rekening mee houden dat de

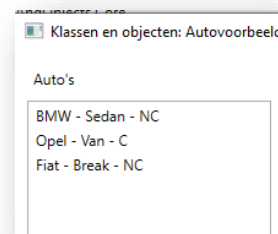
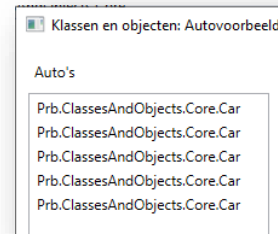
SelectedItem eigenschap altijd een waarde van het type object zal retourneren. We zullen dit object dus eerst moeten casten naar een CarType en vervolgens kennen we deze waarde toe aan een variabele van het type CarType : `CarType carType = (CarType)cmbCarType.SelectedItem;`

Tenslotte kennen we de waarde van deze variabele toe aan de eigenschap CarType van ons nieuwe Car-object : `car.CarType = carType;`

- Test even je applicatie uit. Het toevoegen lukt, maar je merkt dat we nog de override ToString() methode van de Car-klasse zelf ontbreken. Open terug de Car klasse en voeg onderstaande code toe :

```
public override string ToString()
{
    return $"{Brand} - {CarType}";
}
```

Test opnieuw even uit en je merkt dat het toevoegen nu perfect lukt.



Denk even na over de code die we zonet toegevoegd hebben. In onze klasse Car beelden we in de override ToString() methode de (string) waarde Brand af maar beelden we eveneens de (CarType) waarde CarType af : zoals je merkt in het resultaat wordt bij het afbeelden van de CarType waarde meteen ook de override ToString() methode van de CarType-klasse uitgevoerd.

- We moeten er nog enkel nog voor zorgen dat wanneer in de Listbox een Car-object wordt geselecteerd, er in de combobox het overeenkomstig CarType-object wordt geselecteerd. Dat kan heel eenvoudig via de eigenschap CarType van ons Car-object (dat uiteraard een CarType-object is) toe te kennen aan de SelectedItem-eigenschap van de combobox : dit werkt omdat onze combobox immers gevuld is met CarType objecten.

```
private void LstCars_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lstCars.SelectedItem != null)
    {
        Car car = (Car)lstCars.SelectedItem;
        txtCarBrand.Text = car.Brand;
        txtColor.Text = car.Color;
        txtPrice.Text = car.Price.ToString();
        txtTopSpeed.Text = car.TopSpeed.ToString();
        cmbCarType.SelectedItem = car.CarType;
    }
}
```



### Code repository

De eindsituatie van deze paragraaf is te vinden op :

 `git clone` <https://github.com/howest-gp-prb/cu-h9-Compositie-einde.git>

