GODERIK LEFEBVRE

LEREN PROGRAMMEREN

EEN 3-LAGENMODEL, C# IN VISUAL STUDIO

Grote dank aan Marijke Lisabeth, Annick Renders en Dominiek Vandewalle voor de massa's ideeën, feedback en voor het aanbrengen van vele correcties!

Hoofdstuk "2.8 Omdat missen menselijk is" is opgesteld op basis van nota's van Annick Renders.

Merci papa om, zonder enige notie van programmeren, toch alle teksten na te lezen op taalfouten!

Voor het ontwerp van het voorblad, wil ik graag uitgeverij acco bedanken.

Inhoudsopgave

Inhoudsopgave **5**

Woord vooraf **9**

1 Au revoir BlueJ, welcome Visual Studio **13**

1.1 Een klasse _ even opfrissen 14
1.1.1 Header van een klasse 16
1.1.2 Velden van een klasse 16
1.1.3 Constructor van een klasse 16
1.1.4 Accessormethoden van een klasse 17
1.1.5 Mutatormethoden van een klasse 17
1.1.6 Methoden van een klasse 18
1.2 Het class-bestand 21
1.2.1 Using 21
1.2.2 Namespace 22
1.3 Het venster Immediate 22
1.4 C# style 25
1.5 Conclusie 29
1.6 Even oefenen 30

2 Een user interface opbouwen **35**

2.1 Formulieren 37
2.2 Weergaven van een formulier 37
2.2.1 De weergave Design 37
2.2.2 De weergave Code 39
2.2.3 Design Time / Run Time van een formulier 39
2.3 Besturingselementen 40
2.4 De code achter een formulier 42
2.4.1 De startsituatie 42
2.4.2 Samenwerken met de Business laag 44
2.4.3 Methoden in reactie op een gebeurtenis 46
2.4.4 Hulpmethode 49
2.5 Een ander voorbeeldje 51
2.5.1 De Business Layer 52
2.5.2 De Presentation Layer 54
2.6 Program.cs 59

2.7 Even oefenen ... 60
2.8 Omdat missen menselijk is ... 62
2.8.1 Syntaxfouten ... 63
2.8.2 Build Errors ... 67
2.8.3 Logische fouten ... 72
2.8.4 Code debuggen ... 74

3 Properties - part I ... 89

3.1 Properties in een klasse ... 89
3.1.1 Een voorbeeldklasse met accessoren en mutatoren ... 90
3.1.2 Een voorbeeldklasse met Properties ... 92
3.2 Properties in het venster Immediate ... 95
3.3 Properties in de Presentation Layer ... 100
3.3.1 Een Property opvragen ... 103
3.3.2 Een Property wijzigen ... 104
3.3.3 Een Property opvragen én schrijven in één en dezelfde instructie ... 105
3.4 Een ander voorbeeldje ... 106
3.4.1 De Business Layer ... 106
3.4.2 De Presentation Layer ... 109
3.5 Even oefenen ... 114

4 Return of the Properties ... 121

4.1 Kennismaken met het project `Showroom` ... 121
4.1.1 De Business Layer ... 122
4.1.2 De Presentation Layer ... 124
4.2 Objecteigenschappen ... 125
4.2.1 Objecteigenschappen in het Properties venster ... 126
4.2.2 Objecteigenschappen in programmacode ... 127
4.2.3 Objecteigenschappen in de Object Browser ... 129
4.3 Invoercontrole ... 133
4.3.1 Invoercontrole met een `NumericUpDown`-element ... 134
4.4 Even oefenen ... 140
4.5 De Object Browser nader bekeken ... 145
4.5.1 Properties in de Object Browser ... 145
4.5.2 Methoden in de Object Browser ... 148

5 Over Solutions, Projecten en Formulieren ... 159

5.1 Terminologie ... 159
5.2 Een toepassing: from zero to hero ... 160
5.2.1 Een nieuwe solution maken ... 161
5.2.2 Een project toevoegen aan een solution ... 162
5.2.3 De Business Layer opbouwen ... 164

5.2.4 De Presentation Layer opbouwen ... 168
5.3 Een project met meerdere formulieren ... 172
5.4 Een uitsmijter ... 181

6 Collecties in C# ... 189

6.1 De objecten in onze lijst ... 190
6.2 `List`-bewerkingen ... 191
6.2.1 Een `List` declareren ... 191
6.2.2 Een `List` initialiseren ... 191
6.2.3 Een Property van het type `List` ... 192
6.2.4 Een object toevoegen aan een `List` ... 192
6.2.5 Hoeveel objecten in een `List` ... 193
6.2.6 Eén object uit een `List` opvragen ... 194
6.2.7 Met een foreach-lus de objecten in een `List` doorlopen ... 196
6.2.8 Een object uit een `List` wissen ... 197
6.2.9 Meerdere objecten uit een `List` wissen met een lusje ... 199
6.3 Even oefenen ... 200
6.4 Een uitsmijter ... 204

7 Collecties in de presentatielaag ... 207

7.1 De CocktailBar-toepassing ... 208
7.1.1 De Business Layer ... 208
7.1.2 Het formulier `StartForm` ... 209
7.1.3 Het formulier `CocktailInfoForm` ... 211
7.1.4 Het formulier `IngredientForm` ... 212
7.2 `List`-bewerkingen in de presentatielaag ... 213
7.2.1 Een collectie toewijzen aan een `ListBox` ... 213
7.2.2 Het geselecteerde object in de `ListBox` ... 217
7.2.3 Een `ListBox` vernieuwen ... 220
7.3 Even oefenen ... 223

8 Data Access Layer ... 233

8.1 De solution `OpStap` ... 234
8.1.1 De klasse `wandelroute` ... 235
8.1.2 De wandelroutes in een `ListBox` ... 236
8.1.3 De details van een wandelroute tonen ... 237
8.1.4 Een wandelroute wijzigen ... 239
8.1.5 Een nieuwe wandelroute toevoegen ... 240
8.1.6 Een wandelroute verwijderen ... 242
8.1.7 Wat ontbreekt nog ... 243
8.2 De databank `wandelroutes` ... 243
8.3 De Data Access Layer - voorbereiding ... 246

8.3.1 Elke Layer in een apart project ... 246
8.3.2 References ... 248
8.3.3 De Server Explorer ... 251
8.4 De Data Access Layer - programmacode ... 253
8.4.1 De klasse `WandelrouteDA` ... 253
8.4.2 Databankgegevens lezen in de klasse `WandelrouteDA` ... 256
8.4.3 Databankgegevens updaten in de klasse `WandelrouteDA` ... 263
8.4.4 Databankgegevens toevoegen in de klasse `WandelrouteDA` ... 269
8.4.5 Databankgegevens wissen in de klasse `WandelrouteDA` ... 274
8.4.6 Databankgegevens lezen met een filter ... 278
8.4.7 Eén waarde uit de databank lezen ... 284
8.5 Even oefenen ... 289

9 Spiekbriefjes ... 305

9.1 Switch ... 305
9.2 Wiskundige operatoren ... 306
9.3 Conversies met numerieke gegevenstypen en `String` ... 306
9.4 Kleuren (Color) ... 308
9.5 De collectie `List<>` ... 308
9.6 Code in de Data Access Layer ... 309

Woord vooraf

Een leerkracht informatica in het Vlaamse secundair onderwijs mag nooit op zijn lauweren rusten …

In de leerplannen informatica voor de studierichtingen Boekhouden-Informatica en Informaticabeheer werd de roep om meer de nadruk te leggen op **objectgeoriënteerd programmeren** (**OOP**) steeds luider. Als deze leerlingen later in het hoger onderwijs voor een informatica-opleiding kiezen, worden ze vaak al vanaf het prille begin met OOP geconfronteerd.

Ik had al heel wat lesmateriaal samengesteld om de programmeerlessen mee te spekken, maar het onderdeeltje objectgeoriënteerd programmeren bleef toch onderbelicht. Een obligaat hoofdstukje OOP, zo ergens op het einde van het curricum, volstond dus niet meer.

In de lessentabel van de studierichtingen Boekhouden-Informatica en Informaticabeheer staan een pak lesuren die we aan software-ontwikkeling kunnen besteden. Er is dus zeker heel wat tijd beschikbaar om iets op te bouwen. Maar hoe die OOP mooi geïntegreerd te krijgen in de lessen programmeren, was enkele jaren geleden nog een groot vraagteken.

Hieronder mijn twee grootste bedenkingen:

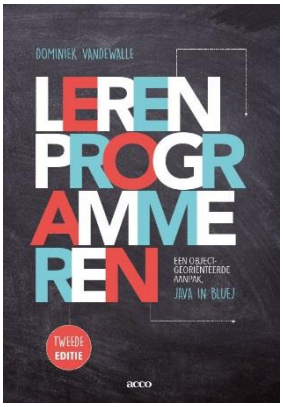
1. Het roer helemaal omgooien en de leerlingen onmiddellijk in contact brengen met OOP, is een optie. Maar leggen we de lat hiermee niet meteen te hoog, want OOP vraagt van de leerlingen ontegensprekelijk een hogere instap? De klassieke 'invoer-berekening-uitvoer'-structuur bood de leerlingen altijd een houvast. Door objectgeoriënteerd te gaan programmeren teken je in op een heel andere filosofie. Het A-B-C van OOP heb je niet in 1-2-3 uitgelegd.

2. Aan het einde van de rit wil je graag het punt bereiken waar de leerlingen een iets grotere toepassing - met alles erop en eraan - zelfstandig kunnen uitdenken en uitwerken. In de ideale wereld zit daar nog één of andere link met een databank bij. Maar is dit nog haalbaar als je alles in een OOP-jasje moet stoppen?

   In de praktijk worden hiervoor vaak frameworks gebruikt, maar door een iets te hoog hocus pocus gehalte (automatisch gegenereerde code door wizards) lijkt dit voor *pedagogische doeleinden* misschien niet de beste keuze.

We zijn ondertussen zo'n drietal jaar verder, dus tijd voor een nieuwe stand van zaken …

Veel van de oorspronkelijke twijfels verdwenen toen een collega zijn proefcursus BlueJ doormailde. BlueJ is een educatieve programmeeromgeving die het levenslicht zag met als voornaamste doel om op een bevattelijke manier de concepten van objectgeörienteerd programmeren aan te brengen.

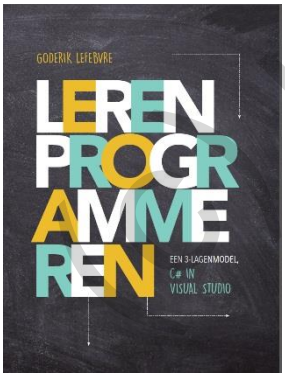


Die proefcursus is ondertussen een heus boek geworden. '**Leren Programmeren, een objectgeoriënteerde aanpak, Java in BlueJ**' door **Dominiek Vandewalle** verscheen al in tweede druk bij uitgeverij **acco**.

Het boek gaat uit van een object first aanpak (met BlueJ kan het ook niet anders). Aan de hand van enkele eenvoudige voorbeelden leer je meteen hoe een **klasse** in elkaar steekt. Via een makkelijke interface heb je van de klasse zo levende **objecten** gemaakt en kan je haast aan de lijve ondervinden op welke manier de code op deze objecten interageert.

Je moet zeker voldoende tijd maken om de lijnen uit te zetten (het boek doet dit ook), maar dan heb je wel een mooi OOP-kader waarbinnen je ook andere thema's als de selectie, lijsten, lussen, … bij de leerlingen kan introduceren. Steeds bekijk/schrijf je code in een **klasse** en test je aan de hand van **objecten** of alles correct werkt. De basisprincipes van OOP worden er m.a.w. voortdurend *ingeslepen*.

Er moest dan echter nog een grote volgende stap gezet worden. Als we willen dat leerlingen *echte* programma's - met alle toeters en bellen - kunnen ontwerpen, zal op een bepaald punt de educatieve programmeeromgeving BlueJ ingewisseld moeten worden voor een meer professionele IDE.



Deze cursus '**Leren Programmeren, een 3-lagenmodel, C# in Visual Studio**' is een poging van enkele sparrende vakcollega's om les-materiaal te ontwikkelen om deze overstap te maken.

Er werd een praktische keuze gemaakt voor de programmeer-omgeving Visual Studio (we hadden hier immers de meeste ervaring mee), maar werden zo ook gedwongen om van Java over te schake-len naar C#.

"Jongens en meisjes, jullie mogen alles vergeten wat je met BlueJ geleerd hebt, vanaf nu doen we het helemaal anders.", was echter absoluut GEEN optie.

De cursus is opgesteld als een waardig **vervolg** bij het boek 'Leren Programmeren, een objectgeoriënteerde aanpak, Java in BlueJ'. Door de juiste voorbeelden te kiezen, lukt het om de overgang van enerzijds BlueJ naar Visual Studio en anderzijds van Java naar C# (de syntax van beide programmeertalen lijkt sterk op elkaar), bijna geruisloos te maken.

Typische Visual Studio - of C#-concepten, zoals Properties, de object browser, solutions … worden gaandeweg, stap voor stap, geïntroduceerd.

OOP blijft echter de rode draad vormen doorheen de volledige cursus!

Elke toepassing vertrekt vanuit één of meerdere (business)**klassen** die de logica van de probleemstelling vatten (= de businesslaag). Het nieuwe is dat we onze toepassingen nu ook gaan voorzien van een user interface (= de presentatielaag). Dat zo'n presentatielaag ook opgebouwd is uit (formulier)**klassen**, die beroep doen op **objecten** uit de businesslaag, past helemaal in ons OOP-plaatje.

Tenslotte bekijken we in een uitgebreid hoofdstuk hoe je de communicatie met een database kan opzetten (= de data-accesslaag). Je leert welke code je in een (data-access)**klasse** nodig hebt om gegevens uit een database in te lezen en om (gewijzigde) gegevens naar een database te laten wegschrijven.

De screenshots in deze cursus en de bron- en oefengegevens bij deze cursus werden gemaakt met/voor de IDE **Visual Studio Community 2017**. Met andere versies van Visual Studio kan je waarschijnlijk ook aan de slag, al zal hier en daar de te volgen weg naar een bepaalde instelling, een demonstratie van een handig Visual Studio-trucje, de beschrijving in een help- of infovenstertje, niet helemaal accuraat zijn.

Als Database Management System bij "hoofdstuk 8 Data Access Layer" hebben we voor **MySQL** gekozen, met als frontend **MySQL Workbench**. Er wordt hierbij verondersteld dat de leerlingen al in staat zijn om MySQL-statements (CRUD) te schrijven en dat ze hun weg al wat kunnen vinden in de gebruiksinterface van MySQL Workbench.

Bij toetsen en proefwerken die wij de leerlingen voorschotelen, heeft de slinger bij ons al enkele keren de weg van "*alle hulpbronnen, inclusief cursus en oefenbestanden, zijn beschikbaar*" tot "*alles gesloten boek, zo zijn jullie wel verplicht om de leerstof goed in te oefenen*" en omgekeerd afgelegd. Op dit moment opteren we meestal voor een tussenoplossing, waar de leerlingen bij een test, een door ons opgesteld *spiekbriefje* mogen gebruiken. Deze spiekbrief vormt het 9de en laatste hoofdstuk van de cursus.

Goderik Lefebvre
goderiklefebvre@sgsintpaulus.be

21 augustus 2018

1 Au revoir BlueJ, welcome Visual Studio

In dit eerste hoofdstuk willen we de brug slaan tussen **BlueJ** - het schitterende educatieve pakket dat we gebruikten om alle basiscomponenten van objectgeoriënteerd programmeren aan te leren - en **Visual Studio** - een professionele Integrated Development Environment (IDE) waarmee we voortaan aan de slag gaan.

Het grootste tastbare verschil dat je vanaf het tweede hoofdstuk zal merken, is dat wij in deze cursus in Visual Studio zullen werken met formulieren met daarop o.a. knoppen (waarmee we acties kunnen opstarten) en tekstvakken (waarin we resultaten kunnen weergeven) en nog zoveel meer. Onze programmacode krijgt t.o.v. onze BlueJ-bouwsels m.a.w. een visuele laag of een **User Interface**. Zoals een zekere leerling ooit zei: "Dan kunnen we *echte* programmaatjes gaan maken".

Maar de eerste stap van alles wat we hier gaan doen, blijft steeds het uitwerken van een **klasse** (of meerdere klassen die met elkaar samenwerken), zoals jullie zo vaak in BlueJ gedaan hebben. Daarboven gaan we dan zo'n visueel laagje zetten.

Een andere klip die we zullen moeten nemen is de overstap van **Java** naar **C#**. In BlueJ ben je steeds klassen in de Java-taal aan het schrijven. Alhoewel Visual Studio meerdere programmeertalen ondersteunt, is Java daar jammer genoeg geen van. Met C# kiezen we wel een taal waar de syntax heel gelijkaardig is aan Java.

Het uitgewerkte voorbeeld in dit hoofdstuk zal tonen hoe sterk Java en C# op elkaar *kunnen* lijken. Ook in deze cursus mogen jullie dus volop {}-accolades; punt-komma's en && of ||-symbolen verwachten.

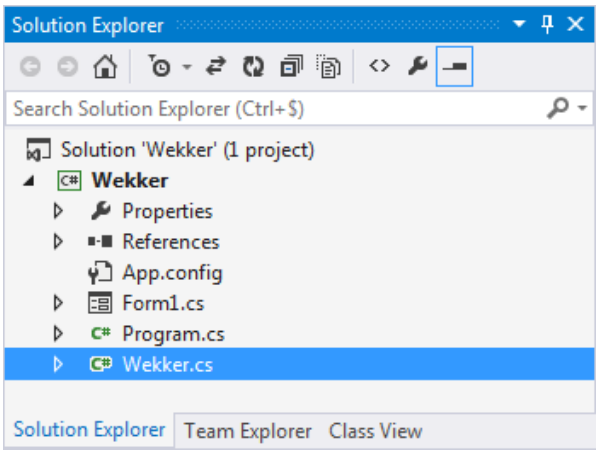
Dat de code er toch iets anders uitziet, komt voor een deel omdat de C#-community wat andere afspraken gebruikt bij de naamgeving van velden, variabelen, methoden, … dan de Java-crowd. We hebben er echter alle vertrouwen in dat na enkele voorbeeldjes en oefeningen de C#-stijl in jullie vingers zal zitten.

Het grootste conceptuele verschil is misschien wel de introductie van **Properties** (die o.a. de **accessor**- en **mutator**methoden zullen vervangen), maar dit reserveren we pas voor het derde hoofdstuk.

1.1 Een klasse _ even opfrissen

Bij de brongegevens van deze cursus vind je in de submap **_hoofdstuk_1/_voorbeelden** een mapje **Wekker** terug. Deze map bevat het eerste Visual Studio project waarmee we aan de slag willen. Door in verkenner te dubbelklikken op het bestand **Wekker.sln** zal het project geopend worden in Visual Studio.

In het rechtervenster (de **Solution Explorer**) zie je het bestandje **Wekker.cs** staan.



Dit bestandje bevat onze klasse `Wekker`. Een eenvoudige dubbelklik zal je volgende code tonen:

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

namespace Wekker
{
    public class Wekker
    {
        private int uur;
        private int minuut;

        public Wekker()
        {
            uur = 0;
            minuut = 0;
        }

        public int getUur()
        {
            return uur;
        }

        public int getMinuut()
        {
            return minuut;
        }

        public void setUur(int uur)
        {
            this.uur = uur;
        }
```

```
        public void setMinuut(int minuut)
        {
            this.minuut = minuut;
        }

        public void urenPlus()
        {
            uur++;
            if (uur > 23)
            {
                uur = uur - 24;
            }
        }

        public void urenMin()
        {
            uur--;
            if (uur < 0)
            {
                uur = uur + 24;
            }
        }

        public void minutenPlus()
        {
            minuut++;
            if (minuut > 59)
            {
                minuut = minuut - 60;
            }
        }

        public void minutenMin()
        {
            minuut--;
            if (minuut < 0)
            {
                minuut = minuut + 60;
            }
        }

        public String alarmOm()
        {
            String minuutString;
            String uurString;

            minuutString = displayTweeCijfers(minuut);
            uurString = displayTweeCijfers(uur);

            return uurString + ":" + minuutString;
        }

        private String displayTweeCijfers(int getal)
        {
            String displayString;

            if (getal < 10)
            {
                displayString = "0" + getal.ToString();
            }
            else
            {
                displayString = getal.ToString();
            }

            return displayString;
        }
    }
}
```

Met onze eerste voorbeeldklasse `Wekker` willen we bewijzen dat **Java** en **C#** heel sterk op elkaar lijken. Bij de C#-klassendefinitie van de klasse `Wekker` (alle code vanaf de regels "`public class Wekker {`" tot de afsluitende "`}`") is er maar één verschilletje te vinden met de klassendefinitie zoals we deze in BlueJ (met Java) zouden uitwerken.

In plaats van ons nu eerst op dit 'verschilletje' te focussen, willen we liever bovenstaande code gebruiken om de componenten die in een klassendefinitie voorkomen nog eens te duiden … of zie dit m.a.w. als een compacte opfrissing van de belangrijkste bouwstenen van een klasse.

1.1.1 Header van een klasse

De **header** (of het omhulsel) van de klassendefinitie van de klasse `Wekker` bestaat uit de coderegels:

```
public class Wekker
{
}
```

1.1.2 Velden van een klasse

De **toestand** van een object wordt bijgehouden in de **velden** van de klasse. De klasse `Wekker` bevat volgende twee velden:

```
private int uur;
private int minuut;
```

Bij elke **instantie** van de klasse `Wekker` zal dus een 'uur' en een 'minuut' bijgehouden worden. Bedoeling is dat we in deze twee velden bijhouden om hoe laat de wekker moet afgaan.

Het gegevenstype `int` geeft aan dat deze velden gehele getallen zullen bevatten.

1.1.3 Constructor van een klasse

De **constructor** van een klasse bepaalt hoe elk **nieuw object** van die klasse initieel ingesteld moet worden. De constructor herken je aan het feit dat de header (van de constructor) dezelfde naam krijgt als de klasse zelf.

In de klasse `Wekker` staat volgende constructor:

```
public Wekker()
{
    uur = 0;
    minuut = 0;
}
```

Bij elke nieuwe instantie die je aanmaakt van de klasse `Wekker` zullen de velden uur en minuut dus op 0 ingesteld worden.

Opmerkingen:

- Onze constructor bevat geen parameters. Uiteraard zijn parameters in C# ook toegestaan. Omdat echter gekozen is om een nieuw `Wekker`-object initieel op 'middernacht' te zetten waren er hier geen parameters nodig.
- Ook zonder de instructies `uur = 0;` en `minuut = 0;` hadden deze velden bij een nieuw object op 0 gestaan, want ook in C# geldt voor een `int`-variabele de defaultwaarde 0.

  Wij volgen voor de duidelijkheid de policy om expliciet *elk veld* een waarde te geven in de constructor van de klasse.

1.1.4 Accessormethoden van een klasse

Om van een instantie de waarde van een bepaald veld te kunnen opvragen, voorzien we een **accessormethode** in de klasse. De klasse `Wekker` bevat zowel voor het veld `uur` als voor het veld `minuut` zo'n accessormethode.

Om het veld `uur` op te vragen aan een `Wekker`-object definieerden we volgende accessor-methode `getUur`:

```
public int getUur()
{
    return uur;
}
```

Uit de header maak je op dat er bij deze methode een retourwaarde is. Het type van deze retourwaarde is hier `int`. Het retourtype wordt in de header vermeld net vóór de naam van de methode (`public int getUur(){}`).

Een methode een waarde laten retourneren, gebeurt met het sleutelwoord **return**. Je ziet dat de retourwaarde van onze methode het veld `uur` is (`return uur;`).

Een accessormethode neemt geen parameters. Vandaar dat er in de header niets tussen de ronde haakjes staat.

1.1.5 Mutatormethoden van een klasse

Om van een instantie de waarde van een bepaald veld te kunnen instellen, voorzien we een **mutatormethode** in de klasse. De klasse `Wekker` bevat zowel voor het veld `uur` als voor het veld `minuut` zo'n mutatormethode.

Om het veld `uur` in te stellen voor een `Wekker`-object hebben we volgende mutator–methode `setUur`:

```
public void setUur(int uur)
{
    this.uur = uur;
}
```

Een mutator is een methode die geen retourwaarde heeft. Dit wordt aangegeven door het sleutelwoord **void** dat in de header vóór de naam van de methode staat.

Een mutatormethode neemt klassiek één parameter. Bij onze methode is dit de parameter `uur` van het gegevenstype `int` (`public void setUur(int uur){}`).

Met de instructie `this.uur = uur;` laten we de waarde van de parameter `uur` wegschrijven in het veld `uur`. Of anders gezegd, het veld `uur` krijgt de waarde van de parameter `uur`.

Omdat de parameter en het veld hier beiden dezelfde naam **uur** hebben, moesten we de naamsverwarring opheffen. Waar je **this** gebruikt, zal je verwijzen naar het veld.

1.1.6 Methoden van een klasse

Naast de constructor(en), accessoren en mutatoren kan je uiteraard een onbeperkt aantal *andere* **methoden** aan een klasse toevoegen. Laat er ons van de klasse `Wekker` nog een drietal nader bekijken.

De methode `void urenPlus()`

```
public void urenPlus()
{
    uur++;
    if (uur > 23)
    {
        uur = uur - 24;
    }
}
```

Dat de methode `urenPlus` geen parameters neemt en geen retourwaarde heeft (zie **void**), kan je perfect aflezen aan de header van de methode.

Deze methode dient om het veld `uur` met één te verhogen. Denk aan het knopje op je wekker (je wekker-app) waarmee je het uur waar de wekker moet afgaan, laat verhogen.

Ook in C# hebben de drie onderstaande instructies trouwens dezelfde betekenis:

```
uur++;
uur += 1;
uur = uur + 1;
```

Omdat we het uur natuurlijk niet onbeperkt kunnen blijven verhogen, moeten we zorgen dat - eens we de 23 gepasseerd zijn - we terugspringen naar 0. De **if** in de methode regelt dit perfect.

De methoden `void urenMin()`, `void minutenPlus()` en `void minutenMin()` bevatten analoge code.

De methode `String alarmOm()`

```
public String alarmOm()
{
    String minuutString;
    String uurString;

    minuutString = displayTweeCijfers(minuut);
    uurString = displayTweeCijfers(uur);

    return uurString + ":" + minuutString;
}
```

Het retourtype van deze methode is `String`. We weten dus dat de retourwaarde een 'tekst' zal zijn.

De bedoeling van deze methode is om het geselecteerde tijdstip terug te geven, maar dan in het formaat zoals een tijd op een digitale wekker wordt weergegeven. Stel dat het veld `uur` op 7 en het veld `minuut` op 15 staat, dan zou onze methode `alarmOm` volgende retour–waarde moeten hebben: "07:15".

Dat we het uur, een ":"-teken en de minuten aan elkaar plakken, kon je zeker al afleiden uit de instructie:

```
return uurString + ":" + minuutString;
```

Maar wat zijn dan de twee mysterieuze instructies daarvóór, waar we de hulpvariabelen `minuutString` en `uurString` een waarde geven?

```
minuutString = displayTweeCijfers(minuut);
uurString = displayTweeCijfers(uur);
```

`displayTweeCijfers` is een hulpmethode die we zelf in de klasse `Wekker` voorzien hebben.

Als het uur (of de minuut) maar uit één cijfer bestaat, dan moet er in de digitale weergave van de tijd een nulletje vóór geplaatst worden. In ons voorbeeldje waar een wekker afgaat om 7 uur 15 minuten, hebben we dus in de digitale weergave een extra nulletje nodig bij de 7 (--> "07:15").

Kijk maar hieronder hoe we dit in de methode `String displayTweeCijfers(int)` lieten regelen:

De methode `String displayTweeCijfers(int getal)`

```
private String displayTweeCijfers(int getal)
{
    String displayString;

    if (getal < 10)
    {
        displayString = "0" + getal.ToString();
    }
```

```
    else
    {
        displayString = getal.ToString();
    }

    return displayString;
}
```

Deze methode neemt als parameter een getal (gegevenstype `int`) en laat een tekst (gegevenstype `String`) retourneren met ditzelfde getal. Als de parameter maar uit één cijfer bestaat, laten we deze retourwaarde voorafgaan door een extra nulletje.

Bijvoorbeeld:

- de parameter `getal` is 15 -> de retourwaarde van de methode is de tekst "15";
- de parameter `getal` is 7 -> de retourwaarde van de methode is de tekst "07".

Controleren of de parameter `getal` al dan niet uit één cijfer bestaat, kan eenvoudig door te testen of deze al dan niet kleiner is dan 10. Vandaar volgende if-else structuur:

```
if (getal < 10)
{
    // getal bestaat maar uit één cijfer
}
else
{
    // getal bestaat NIET uit maar één cijfer
}
```

En dan moeten we het zeker ook nog eens hebben over **conversies** tussen gegevenstypen.

Als we bij deze methode het *getalletje* 15 meegeven als parameter, moeten we de *tekst* "15" retourneren, of dus ergens zal onze parameter die van het type `int` is, moeten omgezet worden naar het type `String`.

Onthoud 1.1 Om een geheel getal (een `int`) om te zetten naar een tekst (type `String`) gebruik je in C# de methode `ToString()`.

Een voorbeeldje:

```
int getal = 15;
String getalAlsTekst;

getalAlsTekst = getal.ToString();
```

De variabele `getalAlsTekst` zal hierna de tekst "15" bevatten.

We hebben uiteindelijk dus toch een verschil tussen Java- en C#-code gevonden. De conversie van `int` naar `String` gebeurde in Java niet met de `ToString()`-methode.

De onderstaande twee instructies uit de klassendefinitie van de klasse `Wekker`, waarmee we de vereiste conversie regelen, zouden in BlueJ (Java) dus een compileerfout opleveren.

```
displayString = "0" + getal.ToString();
displayString = getal.ToString();
```

Opmerking:

- De aandachtige lezer heeft misschien opgemerkt dat we enkel in de header van de methode `String displayTweeCijfers(int)` het sleutelwoord **private** (i.p.v. **public)** gebruikt hebben.

  Omdat we deze methode enkel nodig hebben als 'hulpje' (om ons in de methode `void alarmOm()` te assisteren om, indien nodig, vóór het uur en de minuut een nulletje te zetten) en het NIET de bedoeling is om de methode `String displayTweeCijfers(int)` toe te passen op aparte instanties van de klasse `Wekker`, kunnen we er een private methode van maken. Men verwoordt het soms ook dat zo'n methode enkel dient voor intern gebruik.

1.2 Het class-bestand

Het bestand **Wekker.cs** bevat naast de klassendefinitie (alle code vanaf de regels "`class Wekker {`" tot de afsluitende "`}`") toch nog wat extra code. We geven jullie hieromtrent een beetje verduidelijking.

1.2.1 Using

Bovenaan in het bestand vind je:

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

Het sleutelwoord **using** in C# kan je een *beetje* vergelijken met de **import** uit Java.

Het komt er eigenlijk op neer dat de vele klassen in Visual Studio in bibliotheken georganiseerd worden (zo'n bibliotheek noemt men in Visual Studio een **namespace**). Door bovenaan in een klasse met using naar een bepaalde namespace te verwijzen, zal je op een handiger (kortere) manier gebruik kunnen maken van de klassen in deze namespace.

Eens we wat verder in de cursus gevorderd zijn, komen zeker nog voorbeelden aan bod van wat we precies bedoelen.

De vijf bovenstaande namespaces (`System`, `System.Collection.Generic`, `System.Linq`, `System.Text` en `System.Threading.Tasks`) worden standaard opgenomen als je een nieuw class-bestand aanmaakt. Wij hebben deze instructies dus niet zelf moeten intikken.

Sommige using-statements worden in de editor in het grijs weergegeven. Dit betekent dat in dat bestand nog geen beroep gedaan wordt op de klassen uit de betreffende namespace. Die using-statements zijn dus eigenlijk (nog) overbodig.

1.2.2 Namespace

Dat klassen in namespaces onderverdeeld worden, kan je ook al in **Wekker.cs** observeren:

```
namespace Wekker
{
    public class Wekker
    {
        // code van de klasse Wekker
    }
}
```

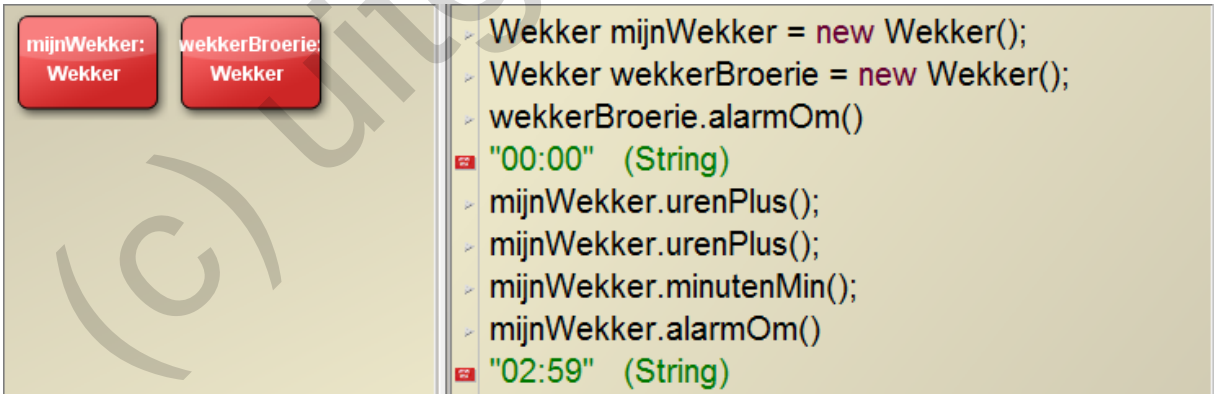
De klassendefinitie van de **klasse** `Wekker` staat in het class-bestand inderdaad binnen de **namespace** `Wekker`.

De naam van de namespace werd hier ontleend aan de naam van het **Visual Studio project** (dit project hebben we dus ook `Wekker` genoemd). Trek dus zeker niet de conclusie dat de namespace en de klasse dezelfde naam moeten hebben.

1.3 Het venster Immediate

Nu we de klasse `Wekker` nader bestudeerd hebben (en jullie hiermee de leerstof van de cursus BlueJ hopelijk weer helemaal geüpdatet hebben), zouden we toch wel graag deze klasse eens aan het werk zien.

In BlueJ kon dit door in het **objectenvak** objecten van de klasse aan te maken (de gekende rode blokjes) om dan allerlei methoden op te roepen bij deze objecten. Een iets moeilijker, maar misschien nog nuttiger, alternatief was om in het **evaluatievak** via tekstinstructies de nieuwe objecten aan te maken om daar dan ook via de juiste expressies allerhande methoden op toe te passen:



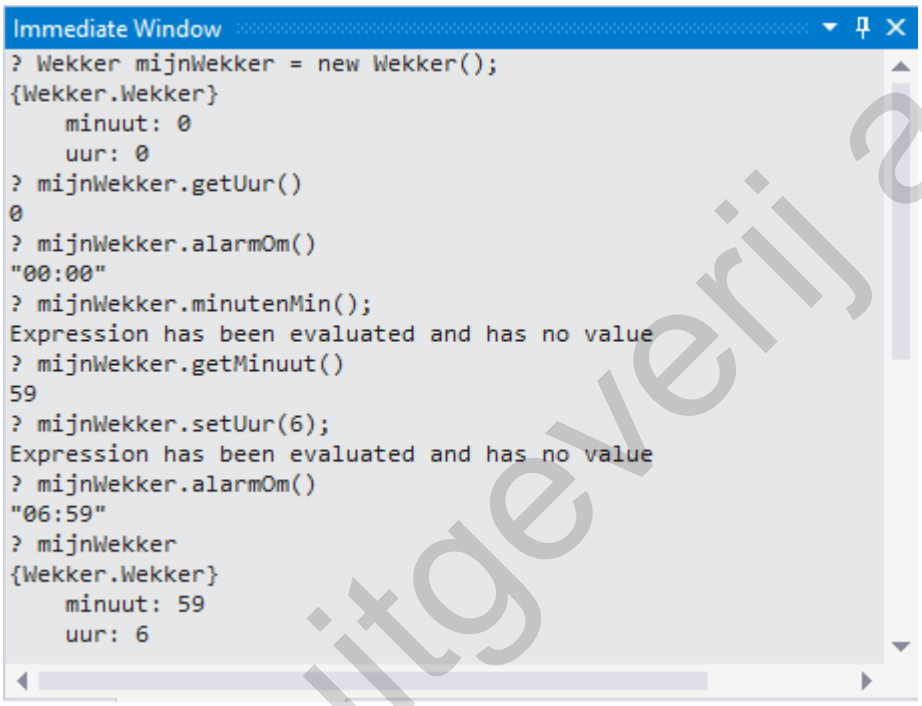
In Visual Studio is het uiteraard NIET meer de bedoeling om op diezelfde manier onze klassen levend te maken (we willen *echte* programmaatjes maken potverdikke). Visual Studio heeft trouwens ook helemaal geen objectenvak.

Toch bestaat er in Visual Studio een gelijkaardig alternatief om op de 'vertrouwde' evaluatievak-manier van BlueJ een klasse uit te testen.

We willen met volle overtuiging deze werkwijze eens uit de doeken doen om zo de link te leggen met wat jullie vorig jaar in BlueJ leerden. Dit blijft trouwens een handige, snelle manier om te testen of een klasse die je programmeerde, wel goed werkt.

We hebben het venster **Immediate** nodig dat je in Visual Studio opent via de menubalk: Debug | Windows | Immediate (of via ctrl+alt+i). Het venster verschijnt onderaan. Misschien moet je bij Tools | Options… | Debugging eerst nog even de optie "**Use Managed Compatibility Mode**" inschakelen om onderstaande demonstratie te laten werken.

Ter informatie: De instructies die we zelf intikken in het venster Immediate laten we steeds voorafgaan door een '**?**' (al is dit eigenlijk niet verplicht). De regels die je hieronder in de screenshot ziet *zonder* een vraagteken, zijn dan de *uitvoer* die Visual Studio genereerde.



In het venster Immediate kan je **nieuwe objecten declareren**. Wij maken een nieuw object `mijnWekker` aan van de klasse `Wekker`. Met behulp van de constructor (via het sleutelwoord **new**) laten we ons object `mijnWekker` ook onmiddellijk **initialiseren**.

```
? Wekker mijnWekker = new Wekker();
```

In het venster Immediate wordt onder de coderegel waar je een object declareert/initialiseert automatisch de toestand van het nieuwe object geprint. Dit betekent concreet dat wij in onze situatie onmiddellijk de waarden te zien krijgen van de velden `uur` en `minuut`:

```
{Wekker.Wekker}
    minuut: 0
    uur: 0
```

En inderdaad, je ziet dat onze constructor ervoor zorgt dat bij nieuwe `Wekker`-objecten de velden `uur` en `minuut` op 0 geïnitialiseerd worden.

Eens we ons object hebben, kunnen we hierop de beschikbare methoden toepassen. Dit met de gekende 'punt-notatie': **object.methode(parameters)**.

Als je in het venster Immediate bij een object een methode oproept **die een resultaat teruggeeft**, krijg je eronder die retourwaarde te zien.

Bij de voorbeeldcode zien we de accessor `getUur` aan het werk:

```
? mijnWekker.getUur()
0
```

Of nog een testje met de methode `alarmOm`. Je ziet dat `String`-waarden hier ook tussen dubbele aanhalingstekens geprint worden:

```
? mijnWekker.alarmOm()
"00:00"
```

In bovenstaande screenshot van het venster Immediate vind je ook nog enkele voorbeeldjes waar we een **methode** oproepen **die geen retourwaarde heeft**. De methoden `setUur` en `minutenMin`, die beiden de toestand van het object wijzigen, zijn hier een voorbeeld van.

```
? mijnWekker.minutenMin();
? mijnWekker.setUur(6);
```

Omdat deze methoden geen resultaat retourneren, krijgen we van Visual Studio de boodschap dat de actie wel uitgevoerd is, maar dat er niets te printen valt:

```
Expression has been evaluated and has no value
```

In het venster Immediate heb je trouwens de mogelijkheid om op elk moment de toestand van een object (of van een variabele) op te vragen. Als je wilt weten hoe de toestand van het object `mijnWekker` eruitziet, doe je dit heel eenvoudig met:

```
? mijnWekker
{Wekker.Wekker}
    minuut: 59
    uur: 6
```

Je hebt misschien opgemerkt dat instructies die we intikten in het venster Immediate soms wel en soms niet afgesloten werden met een puntkomma. We hebben ons gebaseerd op de regeltjes zoals die gelden in BlueJ, maar eigenlijk is het hier iets minder strikt.

- Instructies waar je een object (of een variabele) declareert: -> moet je afsluiten met een puntkomma.
- Instructies waar je een methode oproept MET EEN retourwaarde: -> laten wij niet afsluiten met een puntkomma (maar dit mag wel).
- Instructies waar je een methode oproept die GEEN retourwaarde heeft: -> laten wij afsluiten met een puntkomma (maar dit is eigenlijk niet verplicht).

Opmerking:

- Door rechts te klikken in het venster Immediate en dan **Clear All** te kiezen in het snelmenu maak je het venster weer leeg. De objecten (en variabelen) die je er reeds gedeclareerd had, blijven dan echter in het geheugen bestaan!

  Een trucje waarmee je het geheugen van het venster Immediate toch laat wissen, is door een wijziging aan te brengen aan de code van een klasse. Ergens een onbeduidende spatie intikken betekent ook dat je de klasse gewijzigd hebt.

1.4 C# style

Je hebt ondertussen al door dat de Java- en de C#-syntax heel gelijkaardig zijn. De code van onze klasse `Wekker` kon bijna rechtstreeks vanuit de cursus BlueJ gekopieerd zijn. We moeten echter eerlijk bekennen dat we toch een beetje ons best moesten doen om zo'n *spiegelvoorbeeld* samen te kunnen stellen.

In de loop van de cursus zullen nog heel wat verschillen opduiken tussen Java en C#. We denken bijvoorbeeld aan de **Properties** (hoofdstuk 3) of de **ArrayList** uit Java die zal ver–vangen worden door de **List** in C# (hoofdstuk 7).

Ook als je C#-code te zien krijgt - zoek maar naar voorbeeldcode op internet - zal deze voor jullie nu toch iets minder vertrouwd lijken. C#-programma's hebben immers een *eigen stijl* die wat verschilt van Java-code.

Met **naming conventions** bedoelen we de afspraken die in een community gemaakt werden in verband met de **naamgeving** van klassen, objecten, methoden, velden, parameters, variabelen, ... Java- en C#-programmeurs houden er hierbij wat andere gewoonten op na.

Niet dat deze naming conventions zulke strikte regels zijn die zullen verhinderen dat onze programma's werken als we hiertegen zondigen, maar waarom ons nu niet onmiddellijk conformeren aan de gewoontes die veel andere Visual Studio / C# - programmeurs ook volgen. *Trouwens ... op fora wordt er wel nog eens ruzie gemaakt over welke conventies nu eigenlijk gevolgd moeten worden.*

In deze context willen we eerst de begrippen **camelCase** en **PascalCase** introduceren.

camelCase naamgeving:

- woorden worden aan elkaar geschreven;
- het eerste woord start met een kleine letter;
- alle volgende woorden beginnen met een hoofdletter.

Bijvoorbeeld:

- `maxInschrijving`
- `slimsteLeerlingKlas`

PascalCase naamgeving:

- woorden worden aan elkaar geschreven;
- het eerste woord start met een hoofdletter;
- alle volgende woorden beginnen met een hoofdletter.

Bijvoorbeeld:

- `ToonAantalLeden`
- `MooisteMeisjeKlas`

De naming conventions die we in deze cursus vanaf nu zullen volgen:

klasse	PascalCase bijvoorbeeld: `GemotoriseerdVoertuig`
velden in een klasse	camelCase met een underscore als prefix bijvoorbeeld: `_typeBrandstof`
methoden in een klasse	PascalCase bijvoorbeeld: `GaNaarBenzinestation()`
parameters van een methode	camelCase bijvoorbeeld: `inhoudTank`
variabelen die je declareert binnen een methode	camelCase bijvoorbeeld: `prijsPerLiter`
objecten die je declareert van een klasse	camelCase bijvoorbeeld: `mijnWagen`, `wagenBroerie`

Uiteraard hoeft een naam niet uit meerdere woorden te bestaan. Het tabelletje hierboven vertelt dan ook of je zo'n "enkelvoudige" naam al dan niet met een hoofdletter start.

De code van de klassendefinitie van de klasse `Wekker` in **Wekker.css** staat momenteel nog in de Java-stijl die we in de cursus BlueJ hanteerden. Om deze code om te zetten volgens de *nieuwe* naming conventions van C#, moeten we volgende aanpassingen aanbrengen:

- De velden (`uur` en `minuut`) moeten we laten voorafgaan met een underscore.
- De methoden moeten we laten starten met een hoofdletter.

Hieronder nog eens de volledige code van de klasse `Wekker` maar deze keer in C#-stijl. De wijzigingen plaatsten we in het vet.

```
public class Wekker
{
    private int _uur;
    private int _minuut;

    public Wekker()
    {
        _uur = 0;
        _minuut = 0;
    }

    public int GetUur()
    {
        return _uur;
    }

    public int GetMinuut()
    {
        return _minuut;
    }

    public void SetUur(int uur)
    {
        _uur = uur;
    }

    public void SetMinuut(int minuut)
    {
        _minuut = minuut;
    }

    public void UrenPlus()
    {
        _uur ++;
        if (_uur > 23)
        {
            _uur = _uur - 24;
        }
    }

    public void UrenMin()
    {
        _uur--;
        if (_uur < 0)
        {
            _uur = _uur + 24;
        }
    }

    public void MinutenPlus()
    {
        _minuut ++;
        if (_minuut > 59)
        {
            _minuut = _minuut - 60;
        }
    }
```

```
    public void MinutenMin()
    {
        _minuut --;
        if (_minuut < 0)
        {
            _minuut = _minuut + 60;
        }
    }

    public String AlarmOm()
    {
        String minuutString;
        String uurString;

        minuutString = DisplayTweeCijfers(_minuut);
        uurString = DisplayTweeCijfers(_uur);

        return uurString + ":" + minuutString;
    }

    private String DisplayTweeCijfers(int getal)
    {
        String displayString;

        if (getal < 10)
        {
            displayString = "0" + getal.ToString();
        }
        else
        {
            displayString = getal.ToString();
        }

        return displayString;
    }
}
```

Oefening 1.1 Voer in **Wekker.cs** de hierboven aangeduide wijzigingen aan de klasse `Wekker` door, zodat de code voldoet aan de naming conventions van C#.

Opmerking:

- Volgend trucje kan goed van pas komen als je achteraf een naam van een veld of een variabele wilt aanpassen.

  Net nadat je een veld/variabele van naam veranderde, wordt er rond dat element een stippellijn getekend:

```
private int _uur;
private int minuut;
```

  Ga je met de cursor op dit veld of deze variabele staan, dan verschijnt er een icoontje met een lampje dat je vervolgens kan openklappen:

```
private int _uur;
private  minuut;

public W
```

Rename 'uur' to '_uur'

Met deze optie **Rename 'uur' to '_uur'** wordt in ons geval in de programmacode **overal** de naam 'uur' gewijzigd naar '_uur'. Het grote voordeel: zo hoef je niet je volledige code te doorploeteren om overal handmatig de naam van het veld/variabele te wijzigen.

Nu we in de C#-stijl (enkel) de velden laten voorafgaan door een underscore krijgen we een leuke bijkomstigheid cadeau. De verwarring tussen velden en parameters met dezelfde naam doet zich nu niet meer voor!

In Java-stijl zag de mutatormethode om het veld `uur` in te stellen er als volgt uit:

```
public void setUur(int uur)
{
    this.uur = uur;
}
```

Om in de body van deze mutatormethode naar het veld `uur` te verwijzen, is het sleutelwoord **this** noodzakelijk. Met **this.uur** spreken we het veld aan, met **uur** de parameter.

In C#-stijl hebben het veld (`_uur`) en de parameter (`uur`) niet meer dezelfde naam. Er is m.a.w. geen naamsverwarring meer, waardoor **this** achterwege mag gelaten worden:

```
public void SetUur(int uur)
{
    _uur = uur;
}
```

1.5 Conclusie

In dit eerste hoofdstuk van deze cursus wilden we voornamelijk de link leggen naar wat jullie in de cursus BlueJ (met de programmeertaal Java) geleerd hebben. Een leuke boodschap hierbij is dat - behoudens de naamgeving van elementen - de syntax van Java en C# heel gelijkaardig zijn.

Dat we nog steeds klassen met hun velden, constructoren, methoden, ... gaan programmeren, zou jullie moeten geruststellen dat deze cursus wel degelijk een mooi vervolg is op wat jullie in BlueJ geleerd hebben.

Met het venster Immediate hebben we in Visual Studio zelfs een mogelijkheid om onze klassen te testen net zoals we dit met het evaluatievak van BlueJ deden.

Je vraagt je ondertussen misschien wel af wat we nu eigenlijk anders gaan doen in Visual Studio! Wat maakt dat Visual Studio een Professionele IDE genoemd wordt om software te ontwikkelen en BlueJ 'slechts' een educatief pakket is om objectgeoriënteerd te leren programmeren?

Als we in dit eerste hoofdstuk de gelijkenissen wilden benadrukken, dient het volgende hoofdstuk volop om nieuwe mogelijkheden in Visual Studio te exploreren. Je zal er een user interface leren uitwerken!

Maar misschien eerst met enkele verplichte oefeningen nagaan of na dit eerste hoofdstuk jullie kennis over objectgeoriënteerd programmeren terug op peil staat.

1.6 Even oefenen

Oefening 1.2 - Reisbureau. Ken je de verschillende gegevenstypen nog die je in code kan gebruiken? We laten de belangrijkste in deze oefening nog eens de revue passeren. Je zal merken dat er tussen Java en C# weinig verschillen zijn!

We bouwen deze oefening op rond de klasse `Groepsreis` die jullie in het project `Reisbureau` (zie **_hoofdstuk_1/_oefeningen**) mogen uitwerken. In de Solution Explorer open je hiervoor het bestand **Groepsreis.cs**. Houd je bij deze oefening aan de naming conventions van C#.

Declareer in de klasse `Groepsreis` de volgende velden:

- Het veld `_bestemming` van het type `String` om de hoofdbestemming van de groepsreis bij te houden.
- Het veld `_maxDeelnemers` van het type `int`. In dit veld houden we bij hoeveel personen maximaal kunnen inschrijven voor de groepsreis.
- Het veld `_inschrijvingen` van het type `int`. `_inschrijvingen` houdt bij hoeveel personen al ingeschreven zijn voor de groepsreis.
- Het veld `_prijsPerPersoon` van het type `double`.

  Het gegevenstype `double` laat toe om cijfers na de komma op te slaan!
- Het veld `_kinderenToegelaten` van het type `bool`.

  Je hebt het waarschijnlijk al geraden: het gegevenstype `bool` is de C#-variant van het gegevenstype `boolean` uit Java. Variabelen van het type `bool` kunnen enkel de waarde true en false aannemen.
- Het veld `_klasseVerblijf` van het type `char`. De luxe resorts zullen de klasse 'A' toegewezen krijgen. Bij meer bescheiden hotels zal dit een 'B' zijn.

  Ook in C# plaatst men letterlijke `char`-waarden tussen **enkele** aanhalingstekens, in tegenstelling tot `String`-waarden die tussen **dubbele** aanhalingstekens komen!

Voorzie in de klasse `Groepsreis` een constructor met drie parameters, waarmee we in deze volgorde de bestemming, het maximum aantal deelnemers en de prijs per persoon instellen bij nieuwe `Groepsreis`-objecten. Verder merken we nog op dat bij een groepsreis kinderen standaard toegelaten zijn, dat we initieel voor een klasse 'B' verblijf gaan en dat er onmiddellijk één persoon voor de reis ingeschreven is (nl. de groepsleider).

Programmeer accessormethoden voor alle velden.

Voeg mutatormethoden toe voor de velden `_prijsPerPersoon`, `_kinderenToegelaten` en `_klasseVerblijf`.

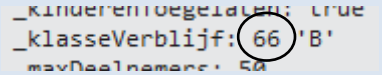
Voorzie de methode `void VolwasseneInschrijven()`. Deze methode laat – als er minstens nog één plaats vrij is - een persoon extra inschrijven voor de groepsreis.

Programmeer de methode `void KinderenInschrijven(int)`. Het aantal kinderen dat je wilt inschrijven geef je mee als parameter. Enkel als er nog voldoende plaats vrij is voor **alle** kinderen, laat je deze inschrijven (optellen bij het veld `_inschrijvingen`); zo niet laat je niemand inschrijven. Jullie moeten zelf expliciet controleren of de groepsreis wel kinderen toegelaten is.

In het venster Immediate kan je experimenten opzetten met de klasse `Groepsreis`. Zo bewijst onderstaande screenshot dat onze constructor; enkele accessoren/mutatoren en de methode `VolwasseneInschrijven()` wel degelijk doen wat moet:

```
Immediate Window
? Groepsreis g = new Groepsreis("Barcelona", 50, 374.99);
{Reisbureau.Groepsreis}
    _bestemming: "Barcelona"
    _inschrijvingen: 1
    _kinderenToegelaten: true
    _klasseVerblijf: 66 'B'
    _maxDeelnemers: 50
    _prijsPerPersoon: 374.99
? g.SetPrijsPerPersoon(399.49);
Expression has been evaluated and has no value
? g.SetKlasseVerblijf('A');
Expression has been evaluated and has no value
? g.VolwasseneInschrijven();
Expression has been evaluated and has no value
? g.GetPrijsPerPersoon()
399.49
? g
{Reisbureau.Groepsreis}
    _bestemming: "Barcelona"
    _inschrijvingen: 2
    _kinderenToegelaten: true
    _klasseVerblijf: 65 'A'
    _maxDeelnemers: 50
    _prijsPerPersoon: 399.49
```

Heb je enig idee wat de betekenis is van het getalletje dat naast het veld `_klasseVerblijf` komt te staan bij de print van de toestand van het object?



Kan je zelf in het venster Immediate een uitvoerige test opzetten om te controleren of de methode `KinderenInschrijven()` zich in elke situatie foutloos gedraagt!

Oefening 1.3 - Aftelklok. In het project `Aftelklok` (**_hoofdstuk_1/_oefeningen**) vind je in het bestand **Aftelklok.cs** de header van de klasse `Aftelklok`. De body van deze klasse is nog helemaal leeg. We zullen in deze klasse een aftelklok proberen te modelleren, die we op een gegeven aantal minuten/seconden kunnen instellen en die dan seconde per seconde kan aftellen tot het klokje weer op nul staat.

Voorzie in de klasse de velden `_minuten` en `_seconden` die respectievelijk het aantal minuten en seconden bijhouden waarop het aftelklokje staat.

Bij de constructor `Aftelklok(int,int)` geef je als parameter de starttijd mee, opgesplitst in het aantal minuten en seconden.

In de klasse `Aftelklok` plaats je zowel voor het veld `_minuten` als voor het veld `_seconden` een accessor- en een mutatormethode. Deze vormen samen dus de vier methoden `int GetMinuten()`, `int GetSeconden()`, `void SetMinuten(int)` en `void SetSeconden(int)`.

Met de methode `void TijdInstellen(int,int)` kan je ineens het veld `_minuten` (via de eerste parameter) en het veld `_seconden` (via de tweede parameter) instellen.

De methode `void Aftellen()` laat de klok met één seconde naar beneden gaan. Als alle seconden van een minuut weggetikt zijn, moet het veld `_minuten` uiteraard met één verminderen (of bijvoorbeeld na *2 minuten 0 seconden* komt *1 minuut 59 seconden*). Als het klokje op *0 minuten 0 seconden* staat, mag deze methode niets meer veranderen aan de velden of anders gezegd, het aftelklokje kan niet op negatieve getallen terechtkomen.

De methode `String ResterendeTijd()` heeft als retourwaarde de tijd waarop het klokje staat; dit in **mm:ss-formaat** of dus de notatie waarop de tijd in een digitaal klokje zou weergegeven worden. Staat het klokje bijvoorbeeld op *2 minuten 5 seconden* levert deze methode dus de `String` *"02:05"* op.

Je ziet bij bovenstaande methode `String ResterendeTijd()` zeker de analogie met de methode `String AlarmOm()` bij de klasse `Wekker`. Het is dus nuttig om ook de private methode `String DisplayTweeCijfers(int)` uit de klasse `Wekker` naar de klasse `Aftelklok` te kopiëren als hulpmethode.

Oefening 1.4 In de vorige oefening stelde je in het project `Aftelklok` de klasse `Aftelklok` op. Jouw opdracht hier is om via het venster Immediate te testen of je deze klasse goed geprogrammeerd hebt.

Geef hiervoor in het venster Immediate de instructies in om onderstaande acties uit te voeren. Controleer telkens kritisch of de aftelklok zich wel goed gedraagt.

Noteer telkens de betreffende instructie.

- Declareer een `Aftelklok`-object met de naam `a` en laat dit klokje hierbij onmiddellijk instellen op *2 minuten* en *3 seconden*.

- Controleer met de methode `ResterendeTijd` of de aftelklok `a` nu inderdaad op *"02:03"* staat.

- Laat vier maal de methode `Aftellen` toepassen op het object `a`. Met de pijltjestoetsen (↑ en ↓) kan je trouwens vlug een eerdere instructie naar boven halen in het venster Immediate.

- Controleer of onze aftelklok `a` nu op *"01:59"* staat.

- Met de methode `TijdInstellen` laat je de tijd van de klok overschrijven naar *0 minuten* en *3 seconden*.

- Laat nog eens vier maal de methode `Aftellen` toepassen op het object `a`.

- Controleer of onze aftelklok `a` nu op *"00:00"* kwam en bleef staan.

2 Een user interface opbouwen

Bij objectgeoriënteerd programmeren probeer je om de werkelijkheid te **modelleren** aan de hand van bepaalde **klassen**. Hoe omvangrijker of complexer de opdracht, hoe meer klassen (die dan onderling zullen samenwerken) je zal moeten uitwerken. De code in de klassen–definities van deze klassen beschrijft hoe de instanties (de objecten) van deze klasse zich zullen gedragen.

De verzameling van deze klassen, die de logica (of de intelligentie) van de toepassing bevatten, noemt met de **businesslaag** (**Business Layer**).

Alles wat we in de cursus BlueJ deden - of elke klasse die we er bij de vele voorbeeldjes, oefeningen en toetsen bestudeerd of geprogrammeerd hebben - hoorde bij deze Business Layer.

Als je kant-en-klare software wil schrijven, zal je echter ook een **visuele laag** moeten toevoegen aan je programma's. In deze cursus zullen we dit in de praktijk brengen via formulieren met *knoppen* (waarmee je acties kan opstarten), *tekstvakken* (waarmee je invoer aan het programma kan leveren of resultaten kan tonen), *ListBoxen* (waarin je collecties kan weergeven), ... en nog zoveel meer.

We zorgen m.a.w. voor een **user interface** waarmee de gebruiker in interactie kan komen met de objecten uit de Business Layer. De code waarmee we deze user interface regelen noemen we de **presentatielaag** (**Presentation Layer**) van de toepassing.

Deze presentatielaag wordt in een toepassing ook soms de **GUI** (Graphical User Interface), of - met wat poëtisch gevoel - het "**laagje mooi**" genoemd.

Onthoud 2.1 Het **tweelagenmodel** beschrijft dat code in een softwaretoepassing onderverdeeld wordt in volgende twee lagen:

- De **Business Layer**, of de verzameling van de klassen die de eigenlijke logica van de toepassing bevatten.
- De **Presentation Layer**, of de **user interface** waarmee een gebruiker in interactie kan komen met objecten uit de Business Layer.

In hoofdstuk 8 schakelen we nog een versnelling hoger en introduceren we **een derde laag** of de **data-accesslaag** (**Data Access Layer**). In die laag leren jullie hoe je vanuit de programmacode een verbinding kan maken met een **externe databank**.

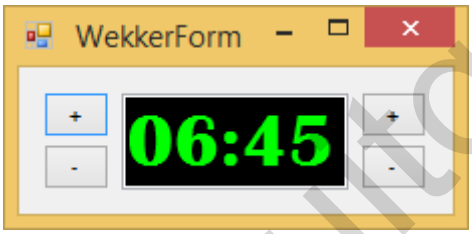
Pas in dat 8e hoofdstuk krijgt het kernwoord **"drielagenmodel"** uit de titel van deze cursus (DRIELAGENMODEL in C# met VISUAL STUDIO) zijn volle betekenis. Tot dan blijven we spreken over het **tweelagenmodel** waar een onderscheid gemaakt wordt tussen de businesslogica en de user interface.

Laat ons eens kijken wat dit tweelagenmodel concreet zou kunnen betekenen voor onze wekker uit het eerste hoofdstuk.

De klasse `Wekker` uit het project `Wekker` hoort duidelijk bij de Business Layer. In deze klasse probeerden we immers een wekker te modelleren:

- We bepaalden er dat het tijdstip waarop de wekker moet afgaan, bijgehouden wordt met aparte getalletjes voor het uur en de minuten (zie de velden van de klasse).
- We beschreven er hoe het tijdstip gewijzigd kan worden (met o.a. de methoden `UrenPlus`, `UrenMin`, `MinutenPlus` en `MinutenMin`).
- We legden er het "uu:mm"-formaat van een wekkerdisplay vast (met behulp van de methoden `AlarmOm` en `DisplayTweeCijfers`).

De Presentation Layer van onze toepassing zou er dan bijvoorbeeld als volgt kunnen uitzien:

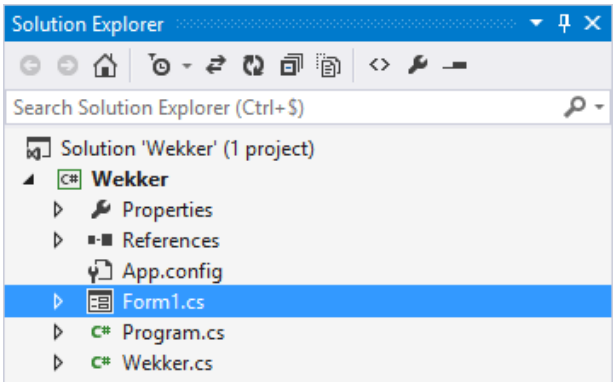


We hebben hier een formuliertje waar men het tijdstip wanneer de wekker afgaat - in het juiste formaat - ziet. Met de knopjes '+' en '-' kan men respectievelijk het uur en de minuten verhogen en verlagen.

Hoe we zo'n user interface uitwerken of dus hoe we dit formuliertje aanmaken en welke programmacode we daarbij op welke plaats moeten voorzien, wordt de inzet van dit hoofdstuk.

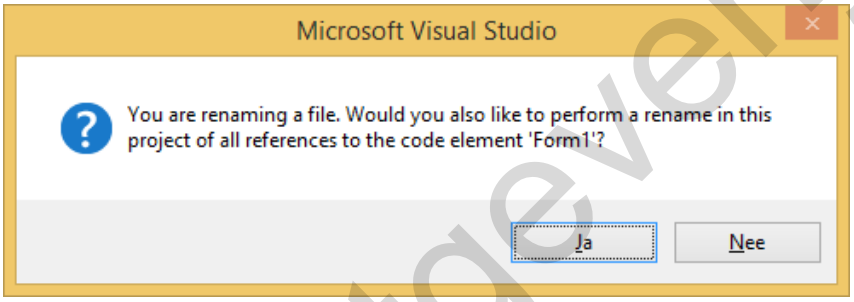
2.1 Formulieren

In het Visual Studio Project `Wekker` (**_hoofdstuk_2/_voorbeelden**) was er standaard al een formulier voorzien. In de Solution Explorer vind je dit formulier terug onder de naam **Form1**.



Form1 vinden wij geen goede naam. Via het snelmenu (=rechtermuisknop) | Rename kan je het formulier hernoemen. We kiezen voor **WekkerForm**.

Visual Studio vraagt ons of we alle verwijzingen in het project naar dit formulier willen aanpassen. Wij beamen volmondig (dus antwoord **Ja**).



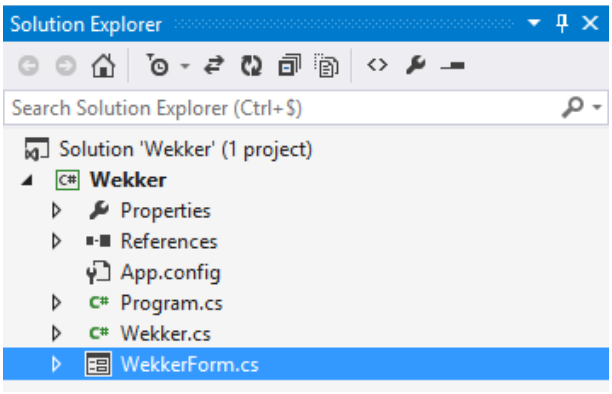
> **Onthoud 2.2** Bij de naamgeving van **formulieren** maken we de afspraak om de **PascalCase**-notatie te gebruiken en achteraan steeds de **suffix Form** toe te voegen.
>
> bijvoorbeeld: `WekkerForm`

2.2 Weergaven van een formulier

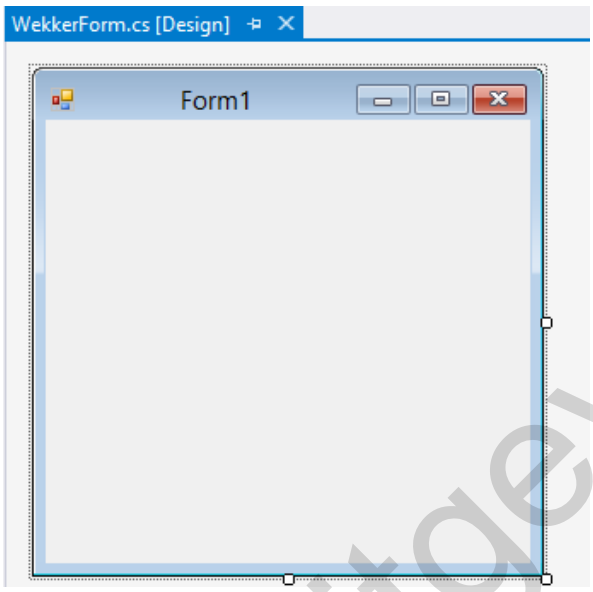
2.2.1 De weergave Design

Als je in de Solution Explorer dubbelklikt op een formulier kom je in de **ontwerpweergave** van het formulier terecht. Wij noemen dit in deze cursus ook de **weergave Design** van het formulier.

Om de weergave Design op te roepen, dubbelklik je op **WekkerForm** in de Solution Explorer.

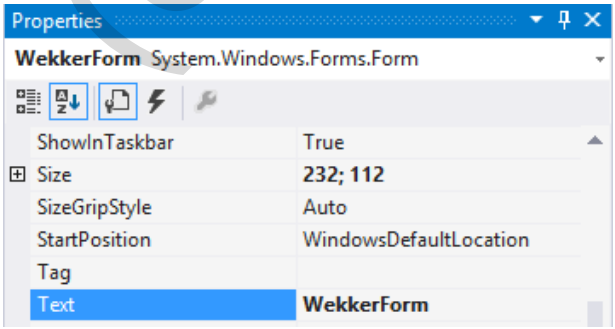


We krijgen een blanco formuliertje te zien:



Misschien onmiddellijk een instelling verbeteren die ons al op de zenuwen werkt. In de titelbalk zien we nog de 'oude' naam Form1 staan. Deze titel kunnen we gemakkelijk aanpassen via het venster **Properties**. Dit venster vind je rechts onderaan in Visual Studio.

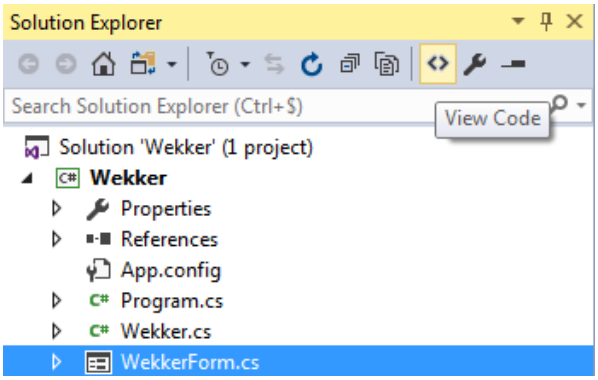
Zorg eerst dat in de weergave Design het formulier geselecteerd staat; in het venster Properties zoek je de eigenschap `Text`. Overschrijf er Form1 door "**WekkerForm**":



Straks zullen we in deze weergave Design alle knopjes en het tekstvak op het formulier zetten en instellen.

2.2.2 De weergave Code

Onze formulieren zullen niet zomaar werken; dit zal toch ook wat programmatie vragen. Om de **programmacode** achter ons formulier te zien, ga je in de Solution Explorer op **WekkerForm.cs** staan en gebruik je het knopje View Code () of de toets **F7**.



Volgende code werd er standaard al voorzien:

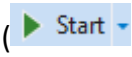
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wekker
{
    public partial class WekkerForm : Form
    {
        public WekkerForm()
        {
            InitializeComponent();
        }
    }
}
```

Dit venster noemen we de **weergave Code** van een formulier. Later (veel) meer over deze standaardcode en hoe er onze user interface te programmeren.

Trouwens, als je vanuit de weergave Design met de toets F7 naar de weergave Code kan, dan lukt het om met **Shift F7** het omgekeerde te doen (van de weergave Code naar de weergave Design).

2.2.3 Design Time / Run Time van een formulier

Als we in de weergave Design of de weergave Code aan het werken zijn, dan zijn we eigenlijk bezig met het *ontwerpen* of het *programmeren* van het formulier. Men zegt dat het formulier (eigenlijk het project) zich dan in **Design Time** bevindt.

Met de knop **Start** () uit de Visual Studio werkbalk laat je het project **opstarten**. Dit betekent concreet dat het **startformulier** van het project geopend wordt. Bij ons project is dit het (enige) formulier `WekkerForm`.

Je zou kunnen zeggen dat ons formulier nu effectief *draait* of dat het programma *loopt*. Men stelt dat het formulier (eigenlijk het project) zich dan in **Run Time** bevindt. Omdat we hier te maken hebben met een leeg formulier, waar we nog niets programmeerden, heeft de Run Time van het formulier ons nog niets te bieden.

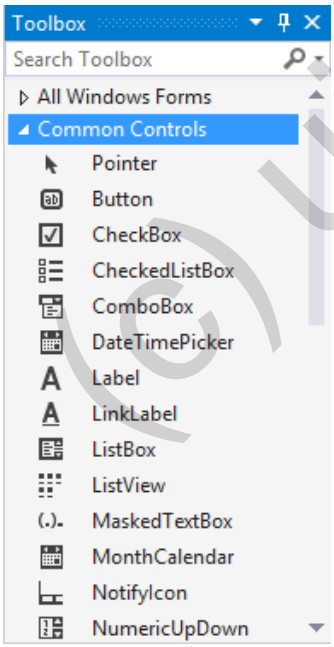
Door het *draaiende* formulier `WekkerForm` te sluiten, of door op het knopje **Stop** () uit de Visual Studio werkbalk te klikken, laat je het project weer stoppen. Je project valt dan m.a.w. terug op Design Time.

2.3 Besturingselementen

We gaan hieronder de nodige elementen op ons formulier laten zetten. We hebben een **tekstvak** nodig (waar we het tijdstip waarop de wekker afloopt tonen) en vier **knoppen** waarmee we respectievelijk het uur en de minuten kunnen verhogen/verlagen.

Tekstvakken, knoppen en andere elementen die je op een formulier kan zetten, noemt men **besturingselementen** (of in het Engels: **controls**).

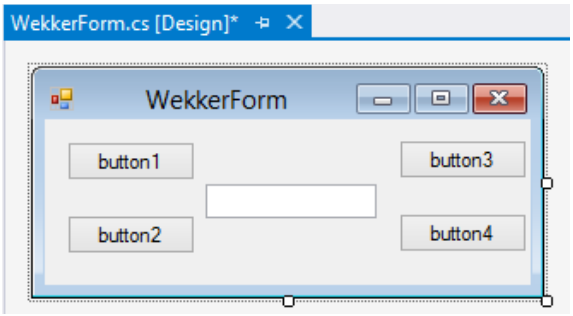
Aan de linkerkant van Visual Studio kan je het venster **Toolbox** laten openen. Als je je in de weergave Design van een formulier bevindt, wordt de Toolbox met heel wat categorieën bevolkt. Klap er de categorie **Common Controls** open:



Opmerking:

- Door rechts bovenaan op het punaise-icoontje te klikken, zet je de Toolbox vast in je Visual Studio environment. Dat lijkt ons de handigste manier van werken.

Sleep één **TextBox** en vier **Buttons** op het formulier:



Elk van deze vijf besturingselementen kunnen we nu via het venster **Properties** nader instellen. Het principe is simpel: je selecteert op het formulier het gewenste besturingselement; in het venster Properties pas je de instellingen aan.

Voor ons tekstvak:

(Name)	**alarmOmTextBox**
BackColor	Black
Font | Name	Britannic Bold
Font | Size	28
ForeColor	Lime
TextAlign	Center

Voor het eerste knopje (knopje links bovenaan):

(Name)	**uurOmhoogButton**
Text	+

Voor het tweede knopje (knopje links onderaan):

(Name)	**uurOmlaagButton**
Text	-

Voor het derde knopje (knopje rechts bovenaan):

(Name)	**minuutOmhoogButton**
Text	+

Voor het laatste knopje (knopje rechts onderaan):

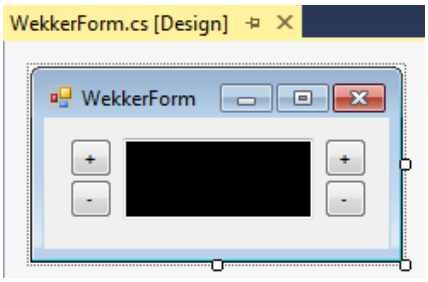
(Name)	**minuutOmlaagButton**
Text	-

Bij de eigenschap `Name` hebben we elk elementje een unieke naam gegeven. Voor besturingselementen (controls) op een formulier houden we ons aan volgende naming convention:

> **Onthoud 2.3** Bij de naamgeving van **besturingselementen** (**controls**) op een formulier gebruiken we de **camelCase**-notatie, waar we als **suffix** nog eens **het type** van het besturingselement vermelden.
>
> Bijvoorbeeld: `alarmOmTextBox`, `uurOmhoogButton`

Door de besturingselementen consequent op deze manier te benoemen, kunnen we aan de naam van het element steeds onmiddellijk zien of het om een tekstvak, een knop, … gaat. Dit zal ons bij het programmeren zeker helpen.

Als je dan nog even de moeite doet om de grootte van de besturingselementen op het formulier aan te passen (via de formaatgrepen) en de elementen op de juiste plaats te slepen, kan je in een handomdraai volgend resultaat namaken:



2.4 De code achter een formulier

2.4.1 De startsituatie

Volgende programmacode kregen we er bij ons formulier `WekkerForm` gratis bij:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wekker
{
    public partial class WekkerForm : Form
    {
        public WekkerForm()
        {
            InitializeComponent();
        }
    }
}
```

Het grootste deel van deze code kunnen jullie al wat plaatsen. Voor de *volledigheid* willen we elk onderdeeltje kort benoemen. Weet echter dat we hier en daar toch enkele zaken verzwijgen of slechts heel minimaal behandelen, omdat we jullie in deze fase nog niet met te veel ballast willen opzadelen:

- Met de **using**-commando's wordt er naar heel wat namespaces verwezen zodat de klassen uit deze namespaces handiger aangesproken kunnen worden.

- Een formulier is een **klasse**!!! Je herkent hier de header van de klasse `WekkerForm`.

```
public partial class WekkerForm : Form
{
}
```

Twee extra elementen zorgen dat deze header toch wat vreemd aanvoelt:

partial Het sleutelwoord **partial** wordt gebruikt om code van een klasse over verschillende locaties (verschillende bestanden) **te verdelen**.

In dit code-bestand zullen wij *onze eigen* code bij de klasse `WekkerForm` kunnen schrijven.

Andere code, *die Visual Studio reeds gegenereerd heeft* om het formulier te laten werken, staat ergens in een ander bestandje in het project 'verborgen'. Toch behoort die 'andere code' ook bij de klasse `WekkerForm`.

overerving Het '**:**' geeft aan dat de klasse `WekkerForm` **overerft** van de klasse `Form`.

In het laatste hoofdstuk van de cursus BlueJ kwam overerving uitvoerig aan bod. In Java wordt het sleutelwoord **extends** gebruikt i.p.v. het ':' om de overervingsrelatie aan te geven.

- In de klasse `WekkerForm` is er reeds een **constructor** voorzien:

```
public WekkerForm()
{
    InitializeComponent();
}
```

Iedere keer als er een nieuw object van de klasse `WekkerForm` aangemaakt wordt, zal deze constructor aangeroepen worden. Zo'n object is dus een instantie van het for–mulier `WekkerForm`.

De methode `InitializeComponent()` zal er o.a. voor zorgen dat alle knopjes en het tekstvak correct op dit formulier gezet worden. De code van deze methode zit in een ander bestandje verborgen (wat mogelijk werd gemaakt via het sleutelwoord Partial).

❖ Klassen worden in Visual Studio onderverdeeld in **namespaces**. Onze klasse `WekkerForm` werd dus onder de namespace `Wekker` gecatalogeerd:

```
namespace Wekker
{
    public partial class WekkerForm : Form
    {
        // ...
    }
}
```

2.4.2 Samenwerken met de Business laag

Het formulier `WekkerForm` is het **visuele laagje** (of de **Presentation Layer**) van onze wekkertoepassing. De eigenlijke intelligentie (of de **Business Layer**) van onze wekker zit echter in de klasse `Wekker`.

Ons formulier (of dus de formulierklasse `WekkerForm`) zal dus noodzakelijk moeten samen–werken met de klasse `Wekker`. We voegen hiervoor volgende (vette) code toe aan de klasse `WekkerForm`:

```
public partial class WekkerForm : Form
{
    private Wekker _wekker;  // het veld _wekker declareren

    public WekkerForm()
    {
        InitializeComponent();

        // het veld _wekker initialiseren als een nieuw Wekker-object
        _wekker = new Wekker();

        // tijdstip van de wekker in het tekstvak zetten
        alarmOmTextBox.Text = _wekker.AlarmOm();
    }
}
```

We laten bovenaan in de klasse `WekkerForm` een veld `_wekker` declareren van de klasse `Wekker`:

```
private Wekker _wekker;  // het veld _wekker declareren
```

De klasse `WekkerForm` zal dus beschikken over één `Wekker`-object. We volgen hierbij natuurlijk de naming convention om veldnamen te laten starten met een underscore.

Onze vaste vuistregel is om bij de constructor van een klasse, alle velden van die klasse te laten initialiseren. Onze klasse `WekkerForm` beschikt over één veld (met de naam `_wekker`). Daarom vind je in onze constructor volgende instructie terug:

```
// het veld _wekker initialiseren als een nieuw Wekker-object
_wekker = new Wekker();
```

Met deze programmaregel laten we ons veld `_wekker` initialiseren door de constructor van de klasse `Wekker` op te roepen. *Mocht je nog eens de code van de constructor van de klasse `Wekker` erop naslaan, zal je begrijpen dat ons veld `_wekker` op 'middernacht' zal staan.*

Omdat `_wekker` een object is uit de businesslaag (dat weliswaar dienst doet in de presentatielaag), noemen we zo'n object verder in de cursus soms een **businessobject**.

Een belangrijke taak van de user interface is om de nodige gegevens "te presenteren"! Zo willen we dat de gebruiker onmiddellijk op het formulier het tijdstip te zien krijgt waarop de wekker ingesteld is. We hebben hiervoor het tekstvak `alarmOmTextBox` voorzien.

Het tijdstip van de wekker in het tekstvak plaatsen, gebeurt in de constructor met:

```
// tijdstip van de wekker in het tekstvak zetten
alarmOmTextBox.Text = _wekker.AlarmOm();
```

De methode `AlarmOm()` van de klasse `Wekker` retourneert het tijdstip van de wekker in uu:mm-formaat. *Mocht je dit niet meer precies weten, kan je steeds nog eens de code van deze methode `String AlarmOm()` in de klasse `Wekker` bekijken.*

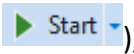
`alarmOmTextBox.Text` is de expressie waarmee je verwijst naar **de inhoud** van het tekstvak `alarmOmTextBox`. Verder in de cursus veel meer over de objecteigenschap `Text` (en objecteigenschappen in het algemeen). Nu is het belangrijk om gewoon te onthouden dat `alarmOmTextBox.Text` betekent: "de tekst die in het tekstvak `alarmOmTextBox` staat".

Onthoud 2.4 Stel je hebt een tekstvak met de naam `vakTextBox`. Met de expressie **`vakTextBox.Text`** verwijs je naar **de inhoud** van dit tekstvak (of m.a.w. welke tekst er in dit tekstvak staat).

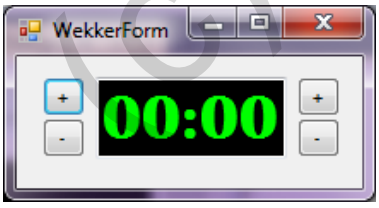
Een andere inhoud naar dit tekstvak wegschrijven, gebeurt met een instructie van de vorm:

```
vakTextBox.Text = "andere inhoud"
```

De instructie "`alarmOmTextBox.Text = _wekker.AlarmOm();`" betekent dus dat je aan het object `_wekker` met de methode `AlarmOm()` opvraagt op welke tijd deze ingesteld staat; dit tijdstip schrijf je weg in het tekstvak `alarmOmTextBox`.

Als je het project opstart (knop 

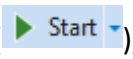
), krijg je nu volgend formulier te zien:



Opmerkingen:

- We merkten net op dat ons formulier - of de visuele laag van onze toepassing - ook gewoon een **klasse** is (de klasse `WekkerForm`)! De principes van objectgeoriënteerd programmeren zullen dus even goed van toepassing zijn voor onze Presentation Layer!

Het gebruik van de constructor in `WekkerForm` is daar al een duidelijk voorbeeld van. Je weet dat een constructor de code bevat die uitgevoerd wordt als er een nieuwe instantie (een object) van de klasse aangemaakt wordt.

Op het moment dat we het project opstarten (

), krijgen we een wekker-formulier te zien. Dat venster is een **object (**van de klasse `WekkerForm`).

Bij het aanmaken van die instantie, zal de code bij de constructor van de klasse `WekkerForm` ervoor zorgen dat er hierbinnen een `Wekker`-object geïnitialiseerd wordt (ons businessobject `_wekker`) en dat het initiële tijdstip van dit `Wekker`-object in het tekstvak geschreven wordt.

- Aan de constructor van de klasse `WekkerForm` werd standaard volgende instructie toegevoegd: "`InitializeComponent();`". Deze methode-oproep zorgt ervoor dat alle besturingselementen (waaronder het tekstvak) aangemaakt worden en op de juiste plaats op het formulier terecht komen.

  Maak niet de fout om deze instructie "per ongeluk" te wissen. Zonder deze instructie zal er immers geen tekstvak `alarmOmTextBox` meer zijn.

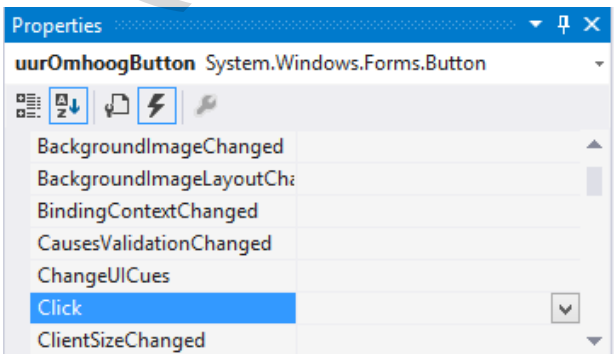
  Snap je dat onze eigen instructie "`alarmOmTextBox.Text = _wekker.AlarmOm();`" in de constructor ná de "`InitializeComponent();`" moet volgen? Een tekstvak moet immers eerst aangemaakt zijn vooraleer we er een tekst naartoe kunnen schrijven.

2.4.3 Methoden in reactie op een gebeurtenis

De volgende stap bij het opbouwen van ons formulier is de knoppen werkend te krijgen. De bedoeling is dat we met het '+' en '-'-knopje aan de linkerzijde het uur verhogen/verlagen. De knopjes aan de rechterzijde dienen om de minuten te besturen.

We beginnen met het '+'-knopje aan de linkerzijde. Dit knopje luistert naar de naam `uurOmhoogButton`.

Selecteer in de weergave **Design** van het formulier onze knop `uurOmhoogButton`; in het venster **Properties** vraag je de **gebeurtenissen** (**Events**) op via het knopje met het bliksem–schichtje ().



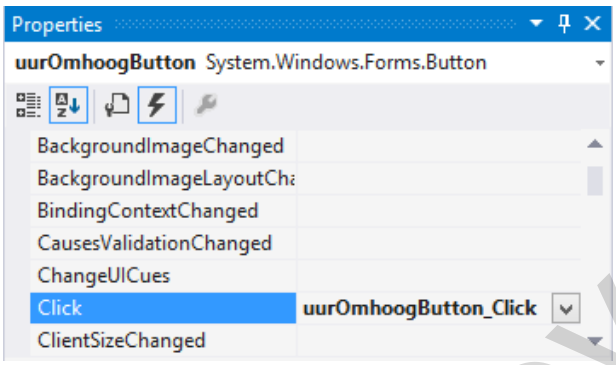
Wij willen dat er straks iets gebeurt als je **klikt** op de knop `uurOmhoogButton` (nl. het uur van onze klok moet dan met één verhoogd worden). Jullie moeten daarom in het venster Properties dubbelklikken in het vakje naast de gebeurtenis `Click`.

Het effect is dat in de weergave **Code** van ons formulier de header van een nieuwe methode `uurOmhoogButton_Click()` toegevoegd werd:

```
private void uurOmhoogButton_Click(object sender, EventArgs e)
{
}
```

De code die je in deze methode schrijft, zal uitgevoerd worden telkens er op de knop `uurOmhoogButton` geklikt wordt.

De koppeling tussen de gebeurtenis `Click` van het knopje en de methode `uurOmhoogButton_Click` wordt als volgt in het venster Properties aangegeven:



Onthoud 2.5 Je kan een methode laten koppelen aan een bepaalde **gebeurtenis** (**Event**). Dergelijke methode zal **getriggerd** (opgestart) worden als die gebeurtenis zich voordoet.

Een klassiek voorbeeld is een methode die opgestart wordt als er geklikt wordt op een knop op een formulier. "Klikken op die knop" is hier dus de trigger (de gebeurtenis) om de methode op te roepen.

Dergelijke methode kan je invoegen door in het venster Properties van het betreffende besturingselement te dubbelklikken naast de gebeurtenis.

Deze methode krijgt dan van Visual Studio automatisch een naam van de volgende vorm:

naamobject_naamgebeurtenis

We plaatsen de volgende code bij deze methode:

```
private void uurOmhoogButton_Click(object sender, EventArgs e)
{
    _wekker.UrenPlus();

    // tijdstip in het tekstvak updaten
    alarmOmTextBox.Text = _wekker.AlarmOm();
}
```

Eerst zorgen we dat we in ons businessobject `_wekker` het uur met één laten verhogen. De methode `UrenPlus()` van de klasse `Wekker` regelt dit. Vandaar de instructie:

```
_wekker.UrenPlus();
```

Hiermee hebben we ons veld `_wekker` wel bijgewerkt, maar het tekstvak op het formulier weet nog van niets. We laten daarom nadien het tijdstip van ons `Wekker`-object nog eens wegschrijven in het tekstvak:

```
// tijdstip in het tekstvak updaten
alarmOmTextBox.Text = _wekker.AlarmOm();
```

Start het project nog maar eens op en controleer of het '+'-knopje aan de linkerzijde zijn werk doet.

Het zou nu geen probleem mogen zijn om ook de drie overige knoppen op het formulier te programmeren. Zorg dat je hiervoor bij elk van deze drie knoppen een methode die reageert op de gebeurtenis `Click` laat aanmaken (door in het venster Properties bij deze knoppen te dubbelklikken naast de gebeurtenis `Click`). Met onderstaande evidente code laten we het formulier dan helemaal werken:

```
private void uurOmlaagButton_Click(object sender, EventArgs e)
{
    _wekker.UrenMin();

    // tijdstip in het tekstvak updaten
    alarmOmTextBox.Text = _wekker.AlarmOm();
}

private void minuutOmhoogButton_Click(object sender, EventArgs e)
{
    _wekker.MinutenPlus();

    // tijdstip in het tekstvak updaten
    alarmOmTextBox.Text = _wekker.AlarmOm();
}

private void minuutOmlaagButton_Click(object sender, EventArgs e)
{
    _wekker.MinutenMin();

    // tijdstip in het tekstvak updaten
    alarmOmTextBox.Text = _wekker.AlarmOm();
}
```

Aan de hand van deze korte methoden kunnen we al perfect het **tweelagenmodel** duiden!

Bijvoorbeeld de methode `uurOmhoogButton_Click` (die zich bevindt in de klasse `WekkerForm`) reageert als de gebruiker klikt op een knop en laat in het tekstvak aan de gebruiker een bijgewerkt tijdstip (één uur later) zien. Dit hoort dus helemaal bij de **user interface** of de **Presentation Layer** van onze toepassing.

Een wekker een uur laten verhogen (en vooral de intelligentie wat te doen als de wekker op 23 uur staat), gebeurt echter binnen de methode `UrenPlus()`. Deze methode staat uitgewerkt in de klasse `Wekker`. De klasse `Wekker` hoort bij de **Business Layer** van onze toepassing.

Door in de klasse `WekkerForm` (= de Presentation Layer) een object van de klasse `Wekker` (= de Business Layer) te gebruiken, komen beide lagen samen. Je mag dit dus zeker zien als een samenwerkingsverband tussen verschillende klassen waarbij één klasse zich in de presentatielaag en een andere klasse zich in de businesslaag situeert.

Opmerkingen:

- We nemen graag nog eens de specifieke header van zo'n methode die reageert op een gebeurtenis onder de loep:

```
private void uurOmhoogButton_Click(object sender, EventArgs e)
{
}
```

  De **void** vertelt ons dat de methode geen retourwaarde heeft. In de methode voorzien we dus ook geen **return**.

  Er worden bij zo'n methoden blijkbaar twee parameters meegegeven, respectievelijk van het type `object` en `EventArgs`. Omdat wij in deze cursus niets aanvangen met deze parameters, gaan we deze gewoon straal negeren!

- De vier methoden die elk reageren op de gebeurtenis `Click` bij respectievelijk één van de knoppen, kregen door onze werkwijze automatisch de naam: *`naamobject`*`_Click` (bijvoorbeeld `uurOmlaagButton_Click`).

  Eigenlijk zondigen we hiermee tegen onze eigen **naming convention** voor methoden, waar we stelden dat methoden in **PascalCase** genoteerd worden, of dus met een hoofdletter moeten starten.

  Het *verbeteren* van de namen van dergelijke methoden (het eerste teken omzetten naar een hoofdletter) levert echter ongewenste complicaties op, waardoor we onszelf met extra werk zullen opzadelen.

  We zijn voor deze keer dus liever pragmatisch en schuiven *in deze concrete situatie* onze principes (of de naming convention) aan de kant. Methoden die reageren op een gebeurtenis zullen in deze cursus *consequent* met een kleine letter starten.

2.4.4 Hulpmethode

Als er in een klassendefinitie een moeilijkere berekening moet uitgevoerd worden, of als we in de code van een klasse een bepaalde bewerking meerdere keren moeten **herhalen**, gebruiken we vaak het handige trucje om bepaalde code in (een) logische **hulpmethode**(n) onder te brengen.

Code **opsplitsen** over verschillende methoden vergelijken wij graag met de succesvolle "verdeel en heers"-tactiek waarmee Julius Caesar onze uitgestrekte regio's kon veroveren (ook al waren wij nog zo dapper). Het is soms handiger om *meerdere* 'makkelijke' problemen op te lossen, dan *één* 'complexe' taak te moeten uitvoeren.

We merkten al op dat de **Presentation Layer** van onze wekkertoepassing, of dus het formulier `WekkerForm`, ook **een klasse** is (de klasse `WekkerForm`). Als het ons nuttig lijkt, kunnen we het principe van hulpmethoden dus ook in deze klasse `WekkerForm` toepassen.

En wat zien we: Elke methode (inclusief de constructor) in de klasse `WekkerForm` eindigt met de instructie om de inhoud van het tekstvak `alarmOmTextBox` te vernieuwen:

```
// tijdstip in het tekstvak updaten
alarmOmTextBox.Text = _wekker.AlarmOm();
```

Je doet niets verkeerd om die instructie, die verschillende keren voorkomt, onder te brengen in een logische aparte **hulpmethode**.

Onderaan onderstaande oplossing zie je de nieuwe methode `void UpdateWekkerInFormulier()`, waar we het tekstvak `alarmOmTextBox` laten vernieuwen.

Bij onze methoden (inclusief de constructor) hoeven we dan enkel deze hulpmethode op te roepen:

```
public partial class WekkerForm : Form
{

    private Wekker _wekker;  // het veld _wekker declareren

    public WekkerForm()
    {
        InitializeComponent();

        // het veld _wekker initialiseren als een nieuw Wekker-object
        _wekker = new Wekker();

        // tijdstip van de wekker in het tekstvak zetten
        UpdateWekkerInFormulier();
    }

    private void uurOmhoogButton_Click(object sender, EventArgs e)
    {
        _wekker.UrenPlus();

        // tijdstip in het tekstvak updaten
        UpdateWekkerInFormulier();
    }

    private void uurOmlaagButton_Click(object sender, EventArgs e)
    {
        _wekker.UrenMin();

        // tijdstip in het tekstvak updaten
        UpdateWekkerInFormulier();
    }

    private void minuutOmhoogButton_Click(object sender, EventArgs e)
    {
        _wekker.MinutenPlus();

        // tijdstip in het tekstvak updaten
        UpdateWekkerInFormulier();
    }
```

```
    private void minuutOmlaagButton_Click(object sender, EventArgs e)
    {
        _wekker.MinutenMin();

        // tijdstip in het tekstvak updaten
        UpdateWekkerInFormulier();
    }

    private void UpdateWekkerInFormulier()
    {
        // tijdstip in het tekstvak updaten
        alarmOmTextBox.Text = _wekker.AlarmOm();
    }
}
```

Hulpmethoden zetten we in principe **private**. Zo ook dus `UpdateWekkerInFormulier()`.

2.5 Een ander voorbeeldje

In het project `Monopoly` (**_hoofdstuk_2/_voorbeelden**) geven we jullie een tweede voorbeeld van een Visual Studio project met een user interface.

We vragen ons even af of jullie, de jeugd van tegenwoordig, het bordspel **Monopoly** ook gespeeld hebben. In pre-iPad-tijden behoorden de spelregels van Monopoly alleszins tot het collectief geheugen.

Onderstaande moet je weten om met ons Monopoly-voorbeeld aan de slag te kunnen:

De meeste vakken op het Monopoly-spelbord stellen "straten" voor. De spelers kunnen deze straten kopen. Als een tegenspeler met zijn/haar pion op één van jouw straten landt, dan moet hij/zij aan jou een huur betalen. De hoogte van de huurprijs is afhankelijk van hoeveel huizen of hotels je eerder op dat vak gebouwd hebt.

Straten bebouwen is onderhevig aan enkele regels:

- Er kunnen maximaal vier huizen op een straat gezet worden.
- Een hotel kan je pas bouwen als er al vier huizen op de straat stonden. Komt er een hotel op de straat, dan moeten die vier huizen er weg.
- Eens er een hotel staat, mag er op die straat niets extra's meer gebouwd worden.

Vóór de Monopoly-adepten ons gaan lynchen: we weten dat we de spelregels hiermee een ietsje vereenvoudigd hebben. Vergeef het ons a.u.b.!

De bedoeling is nu om een formulier uit te werken, waarmee we de toestand van zo'n Monopoly-straat kunnen simuleren. In dit formulier kunnen we voor een gegeven straat – volgens de spelregels – huizen en hotels kopen. We krijgen steeds te zien hoeveel huur ons deze straat, in zijn huidige staat, oplevert.

We maken het wat bevattelijker door jullie al een screenshot te geven van de eindsituatie die we voor ogen hebben:

2.5.1 De Business Layer

De businesslaag, waar we de logica van een Monopoly-straat vastleggen, is in het project al helemaal af.

Over welke gegevens er bij zo'n Monopoly-straat moeten bijhouden worden (=de velden); over welke gegevens opgevraagd moeten kunnen worden en welke acties op zo'n straat moeten kunnen uitgevoerd worden (=de methoden), heeft iemand anders al goed nagedacht. Dit alles resulteerde in de klasse `MonopolyStraat`.

Om met Monopoly-straten aan de slag te kunnen, leggen we hieronder uit op welke manier je een nieuw `MonopolyStraat`-object aanmaakt en geven we een overzicht van welke methoden voorhanden zijn.

Hoe de constructor en de verschillende methoden precies geprogrammeerd zijn, is eigenlijk niet van belang om straks het formulier uit te werken. De code van de klasse `MonopolyStraat` zullen we daarom zelfs NIET bekijken.

De constructor

Bij een nieuw `MonopolyStraat`-object moeten we heel wat gegevens (parameters) aanleveren! De header van de constructor `MonopolyStraat(String, String, int, int, int, int, int, int)` is gigantisch.

In volgorde geef je volgende acht parameters mee:

- de naam van de straat;
- de stad waarin de straat ligt;
- huur als de straat onbebouwd is;
- huur voor de straat met één huis;
- huur voor de straat met twee huizen;
- huur voor de straat met drie huizen;
- huur voor de straat met vier huizen;
- huur voor de straat met één hotel.

In het venster Immediate ziet het er zo uit:

```
Immediate Window
? MonopolyStraat meir = new MonopolyStraat("Meir", "Antwerpen", 35, 175, 500, 1100, 1300, 1500);
```

Een nieuw `MonopolyStraat`-object zal standaard nog geen huizen/hotels bevatten.

De methoden

Er zijn vier accessormethoden voorzien waarmee we respectievelijk de straatnaam, de stad, het aantal huizen en het aantal hotels kunnen opvragen:

```
int GetStraat()
int GetStad()
int GetAantalHuizen()
int GetAantalHotels()
```

In het venster Immediate geeft dit bijvoorbeeld:

```
? meir.GetStraat()
"Meir"
? meir.GetAantalHuizen()
0
```

Met de methoden `void KoopHuis()` en `void KoopHotel()` kunnen we huizen en hotels in een straat bouwen.

Opgelet - dit zijn 'slimme' methoden: Bijvoorbeeld de methode `KoopHotel()` zal enkel **één** hotel op de straat toestaan mits alle vereisten voldaan zijn (er moeten al vier huizen staan, die dan ook nog eens vernietigd worden bij het bouwen van het hotel).

Onderstaande screenshot uit het venster Immediate bewijst dat je nog geen hotel op de Meir mocht zetten.

```
? meir.KoopHuis();
? meir.KoopHotel();
? meir.GetAantalHuizen()
1
? meir.GetAantalHotels()
0
```

De laatste methode `int GeefHuur()` vertelt hoeveel huur er van toepassing is. Uiteraard verandert de retourwaarde van deze methode als er huizen / een hotel bijkomen in de straat:

```
? meir.GeefHuur()
175
? meir.KoopHuis();
? meir.GeefHuur()
500
```

2.5.2 De Presentation Layer

Het **formulier** `StraatForm` wordt de visuele laag van onze toepassing. Wij hebben reeds alle nodige tekstvakken/knoppen op dit formulier geplaatst. We tonen later wel eens hoe je een afbeelding op een formulier krijgt.

Als je de weergave **Code** opvraagt van dit formulier, waar je in de **klasse** `StraatForm` terechtkomt, merk je dat er echter nog niets geprogrammeerd werd. Dat brengen we nu in orde …

Velden in de klasse `StraatForm`

Je weet dat je in het formulier de gegevens van één Monopoly-straat moet tonen (en dus zal moeten bijhouden). Je begrijpt ook dat dit formulier de spelregels van Monopoly moet snappen inzake het kopen van immobiliën (we willen dat de knopjes 'Koop huis' en 'Koop hotel' correct werken).

Dit alles bereiken we door in de klasse `StraatForm` een veld van het type `MonopolyStraat` bij te houden!!!

```
namespace Monopoly
{
    public partial class StraatForm : Form
    {
        private MonopolyStraat _straat;

        public StraatForm()
        {
            InitializeComponent();
        }
    }
}
```

Het formulier `StraatForm` (of dus eigenlijk de klasse `StraatForm`), beschikt nu over een `MonopolyStraat`-object. Dit veld `_straat` kan uiteraard de gegevens van één straat bijhouden. De vele methoden uit de klasse `MonopolyStraat` (bijv. om een huis of hotel te kopen in een straat) zijn allen beschikbaar via ons object `_straat`.

Het veld `_straat` stelt ons m.a.w. in staat om functionaliteit uit de Bussines Layer van de toepassing beschikbaar te hebben in de Presentation Layer! Het veld `_straat` noemen we een businessobject in ons formulier.

De constructor in de klasse `StraatForm`

In de constructor van een *formulierklasse* gaan we gewoonlijk twee zaken regelen:

1. We initialiseren alle velden (dat doen we immers sowieso in *elke* klasse)!!!
2. We bepalen wat we initieel (bij het openen) op het formulier willen zien.

Puntje 1 betekent dat we in de constructor van de klasse `StraatForm` ons veld `_straat` zullen moeten initialiseren. Omdat `_straat` een object van het type `MonopolyStraat` is, moeten we hiervoor de constructor van de klasse `MonopolyStraat` oproepen.

Stel dat we eens met het 'Veldstraat'-vakje uit Monopoly willen werken, dan moeten we ons businessobject als volgt, met alle correcte huurprijzen, initialiseren:

```
public partial class StraatForm : Form
{
    private MonopolyStraat _straat;

    public StraatForm()
    {
        InitializeComponent();

        _straat = new MonopolyStraat("Veldstraat", "Gent", 28, 150,
                                     450, 1000, 1200, 1400);
    }
}
```

Puntje 2 betekent dat we even moeten nadenken over welke gegevens we in de tekstvakken van het formulier zullen zetten.

Niet zo moeilijk: in elk van de vijf tekstvakken printen we de overeenkomstige informatie van ons businessobject (van onze straat).

```
public partial class StraatForm : Form
{
    private MonopolyStraat _straat;

    public StraatForm()
    {
        InitializeComponent();

        _straat = new MonopolyStraat("Veldstraat", "Gent", 28, 150,
                                     450, 1000, 1200, 1400);

        straatTextBox.Text = _straat.GetStraat();
        stadTextBox.Text = _straat.GetStad();
        aantalHuizenTextBox.Text = _straat.GetAantalHuizen().ToString();
        aantalHotelsTextBox.Text = _straat.GetAantalHotels().ToString();
        huurTextBox.Text = _straat.GeefHuur().ToString();
    }
}
```

We pikken uit het lijstje van vijf nieuwe (vette) instructies er eventjes twee uit.

Om – bij het openen van het formulier – de *stad* te tonen in het tekstvak `stadTextBox`:

```
stadTextBox.Text = _straat.GetStad();
```

Wat valt er op te merken:

- Met de methode `GetStad()` vragen we aan ons businessobject `_straat` de bijhorende stad op.
- `stadTextBox.Text` is de expressie waarmee we de *inhoud* van het tekstvak aanduiden.

We printen m.a.w. de stad van ons object `_straat` in het overeenkomstige tekstvak `stadTextBox`.

De instructie om bijvoorbeeld de *huurprijs* van de straat in het betreffende tekstvak te zetten, is iets langer:

```
huurTextBox.Text = _straat.GeefHuur().ToString();
```

Bij deze instructie moesten we een conversie-issue oplossen. Waar zit het probleem:

- Het retourtype van de methode `GeefHuur()` is `int`.
- De inhoud van een tekstvak wordt altijd aanzien van het type `String`.

Visual Studio is streng, en laat niet toe dat we rechtstreeks een `int`-waarde in een tekstvak zetten.

Bij 'Onthoud 1.1' (zie pagina 20) hebben we al geleerd hoe we met de methode `ToString()` een getal (een `int`) kunnen omzetten naar een tekst (een `String`).

Door bij deze instructie met `ToString()` de huurprijs eerst naar een tekst te converteren, kunnen we die alsog in het tekstvak `huurTextBox` laten wegschrijven.

De methoden in de klasse `StraatForm` die reageren op een gebeurtenis

De interactie in het formulier `StraatForm` zit bij de knopjes 'Koop huis' en 'Koop hotel'. Klikken op deze knoppen zou respectievelijk, conform de spelregels, een extra huis of een hotel op onze straat moeten zetten.

Om de knopjes te laten reageren als erop geklikt wordt, voegen we aan de klasse `StraatForm` zowel voor `koopHuisButton` als voor `KoopHotelButton` een methode toe die reageert op de gebeurenis `Click`.

Deze methoden zullen volgende header krijgen:

```
private void koopHuisButton_Click(object sender, EventArgs e){}

private void koopHotelButton_Click(object sender, EventArgs e){}
```

Het denk- of analysewerk welke code in deze methoden moet komen, laten we in onderstaande oefening even over aan jullie.

Oefening 2.1 Op het formulier `StraatForm` willen we, bij het klikken op het knopje 'Koop huis', te zien krijgen wat er – volgens de spelregels – zal gebeuren als je voor de gegeven straat een huis *probeert* te kopen.

Omschrijf welke acties er in de betreffende methode nodig zijn om ons formulier correct te laten reageren. Tip: het businessobject `_straat` speelt een cruciale rol!

Oefening 2.2 Op het formulier `StraatForm` willen we, bij het klikken op het knopje 'Koop hotel', te zien krijgen wat er – volgens de spelregels – zal gebeuren als je voor de gegeven straat een hotel *probeert* te kopen.

Omschrijf welke acties er in de betreffende methode nodig zijn om ons formulier correct te laten reageren.

Check maar eens of de analyse die jullie hierboven gemaakt hebben, overeenkomt met volgende uitwerking:

```
private void koopHuisButton_Click(object sender, EventArgs e)
{
    _straat.KoopHuis();

    aantalHuizenTextBox.Text = _straat.GetAantalHuizen().ToString();
    huurTextBox.Text = _straat.GeefHuur().ToString();
}

private void koopHotelButton_Click(object sender, EventArgs e)
{
    _straat.KoopHotel();

    aantalHuizenTextBox.Text = _straat.GetAantalHuizen().ToString();
    aantalHotelsTextBox.Text = _straat.GetAantalHotels().ToString();
    huurTextBox.Text = _straat.GeefHuur().ToString();
}
```

Hulpmethoden in de klasse `StraatForm`

Hulpmethoden in een klasse kunnen helpen om code beter te structureren, om code overzichtelijker te houden, om te voorkomen dat je steeds dezelfde code moet herhalen, …

In de klasse `StraatForm` kunnen wij wel enkele bewerkingen aanduiden die meerdere keren uitgevoerd worden en bijgevolg in een logische hulpmethode zouden passen. Zien jullie het ook?

Oefening 2.3 Kunnen jullie hieronder een voorstel formuleren van *welke* hulpmethode(n) je in de klasse `StraatForm` *waarom* zou voorzien.

In onze eindoplossing voor de klasse `StraatForm` zie je welke keuzes wij gemaakt hebben:

```
public partial class StraatForm : Form
{
    private MonopolyStraat _straat;

    public StraatForm()
    {
        InitializeComponent();

        _straat = new MonopolyStraat("Veldstraat", "Gent", 28, 150, 450, 1000,
                  1200, 1400);

        straatTextBox.Text = _straat.GetStraat();
        stadTextBox.Text = _straat.GetStad();

        UpdateBebouwingEnHuur();
    }

    private void koopHuisButton_Click(object sender, EventArgs e)
    {
        _straat.KoopHuis();

        UpdateBebouwingEnHuur();
    }

    private void koopHotelButton_Click(object sender, EventArgs e)
    {
        _straat.KoopHotel();

        UpdateBebouwingEnHuur();
    }

    private void UpdateBebouwingEnHuur()
    {
        aantalHuizenTextBox.Text = _straat.GetAantalHuizen().ToString();
        aantalHotelsTextBox.Text = _straat.GetAantalHotels().ToString();
        huurTextBox.Text = _straat.GeefHuur().ToString();
    }
}
```

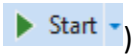
2.6 Program.cs

We willen ook nog eens vlug kijken naar het bestandje **Program.cs** dat in ons Visual Studio project een heel specifieke taak inneemt.

Dubbelklik op **Program.cs** in de Solution Explorer:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wekker
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new WekkerForm());
        }
    }
}
```

De klasse `Program` is een bijzondere klasse in een project.

De voornaamste functie van deze klasse is dat als je het project opstart (knop 

) de methode `void Main()` van deze klasse automatisch opgeroepen wordt.

Onze interesse gaat vooral uit naar de laatste instructie van deze methode:

```
Application.Run(new WekkerForm());
```

Wat doet deze instructie:

- "`new WekkerForm()`" betekent dat er een nieuwe instantie van de klasse `WekkerForm` aangemaakt wordt. Of m.a.w. er wordt een nieuw formulierobject aangemaakt.
- "`Application.Run()`" zal dit nieuwe formulier op het scherm tonen.

Of m.a.w. deze instructie is er dus voor verantwoordelijk dat als je het project opstart er dan (een instantie van) het formulier `WekkerForm` geopend wordt.

Opmerkingen:

- Als je snapt wat er hier in feite gebeurt, heb je misschien ook door dat je deze ene programmaregel kan opsplitsen in volgende twee, iets meer vertrouwde, instructies.

```
WekkerForm formuliertje = new WekkerForm();
Application.Run(formuliertje);
```

Enerzijds het nieuwe formulierobject - dat wij `formuliertje` noemen - aanmaken en anderzijds dit formulier op het scherm laten tonen, gebeurt nu in twee aparte stappen.

- Als we later projecten hebben met *meerdere* formulieren en we willen een ander formulier instellen als **startformulier**, dan kan je dit via deze bewuste instructie makkelijk regelen.

```
Application.Run(new AnderFormulier());
```

2.7 Even oefenen

Oefening 2.4 - Aftelklok. Het project `Aftelklok` bij de brongegevens voor de oefeningen (mapje **_hoofdstuk_2/_oefeningen**) bevat:

- De klasse `Aftelklok`. Met deze klasse simuleren we een aftelklok.

  Zo'n aftelklok kunnen we op een bepaalde tijd instellen (minuten/seconden). Met de methode `Aftellen()` kunnen we de klok één seconde laten aftellen. Bij 'Oefening 1.3' (zie pagina 32) vind je een gedetailleerde beschrijving van deze klasse.

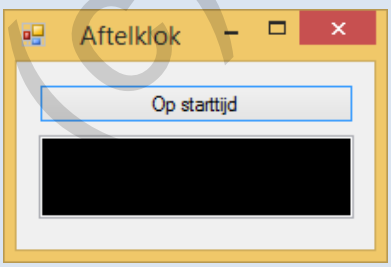
  De klasse `Aftelklok` vormt de **Business Layer** van onze toepassing.

- Het formulier `AftelklokForm`. In dit - nog lege - formulier willen we een **visuele** versie van een aftelklok maken. Het wordt jullie taak om in dit formulier de **Presentation Layer** van de toepassing uit te werken.

We willen ons formulier `AftelklokForm` effectief als een aftelklok kunnen gebruiken. We maken de voorafname dat ons klokje steeds vanaf **1 minuut 30 seconden** aftelt.

Stap voor stap bouwen we de user interface op:

- Plaats in de weergave **Design** op het formulier een knop (`Button`) en een tekstvak (`TextBox`) en noem deze via het venster Properties (eigenschap `Name`) respectievelijk `opnieuwOpStarttijdButton` en `displayTextBox`.



Via het venster Properties zal je toch nog één en ander moeten instellen:

a) In de knop laat je de tekst 'Op starttijd' zetten (eigenschap `Text`).

b) Het tekstvak geef je een zwarte achtergrond (eigenschap `BackColor`), met een andere dan zwarte tekstkleur (eigenschap `ForeColor`). Wij stellen verder een groter

lettertype (eigenschap `Font`) en een gecentreerde uitlijning (eigenschap: `TextAlign`) voor.

❖ De logica van een aftelklok zit in de klasse `Aftelklok`. Ons formulier heeft daarom een `Aftelklok`-object nodig.

Ga hiervoor naar de weergave **Code** van het formulier (je komt in de klasse `Aftelklok-Form` terecht). Declareer binnen deze klasse `AftelklokForm` een veld `_klokje` van de klasse `Aftelklok`.

Binnen de presentielaag zal `_klokje` dienst doen als een object uit de Business Layer. `_klokje` is dus het businessobject in het formulier.

❖ Bij de **constructor** van de klasse `AftelklokForm` zullen we het formulier instellen zoals het eruit moet zien als het formulier geopend wordt.

In twee stappen:

a) In de constructor moeten de velden **geïnitialiseerd** worden. Laat het veld `_klokje` initialiseren op 1 minuut en 30 seconden. Zo geven we onze aftelklok de gevraagde beginwaarde.

b) Om ook iets te zien te krijgen in het formulier laat je in de display (het tekstvak) de tijd van het klokje weergeven. Zoek in de klasse `Aftelklok` op met welke methode je de tijd (in mm:ss-formaat) kan opvragen aan zo'n object.

❖ En dan een nieuwtje ... als we een echte aftelklok willen, moeten we zorgen dat deze **elke seconde** aftelt. Dit is nieuwe leerstof!

In de **Toolbox** vind je bij de categorie **Components** de **Timer** terug (zorg dat je in het formulier in de weergave **Design** staat). Sleep zo'n `Timer`-object op het formulier. In het componentenvak onder het formulier zie je deze timer dan staan.

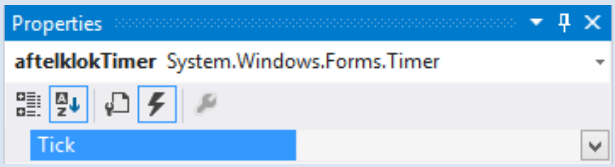
In het venster **Properties** maken we volgende instellingen voor deze timer.

(Name):	**aftelklokTimer**
Interval:	1000
Enabled:	True

De eigenschap `Interval` geeft aan om de hoeveel milliseconden de timer zal 'tikken'. Omdat wij het interval instellen op **1000** milliseconden, betekent dit dat de timer elke seconde tikt.

De eigenschap `Enabled` geeft aan of de timer al dan niet ingeschakeld is. Door deze eigenschap op **true** te zetten is de timer ingeschakeld.

In het venster Properties kan je via de knop met het bliksemschichtje de gebeurtenissen van de timer opvragen. Eén gebeurtenis is beschikbaar, nl. `Tick`.



Dubbelklik naast deze gebeurtenis `Tick` om de bijhorende methode aan te maken. Je moet nu gewoon weten dat deze nieuwe methode `aftelklokTimer_Tick()` uitgevoerd wordt elke keer als de timer 'tikt'. M.a.w. de "timer die tikt" is hier de gebeurtenis die deze methode zal triggeren.

Omdat onze timer elke seconde tikt (zie het timerinterval), zal de code uit de methode `aftelklokTimer_Tick()` dus elke seconde eens uitgevoerd worden.

Jullie moeten volgende twee acties uitvoeren bij deze methode `aftelklok-Timer_Tick()`:

a) Laat het veld `_klokje` (ons `Aftelklok`-object) één seconde aftellen. Zoek in de klasse `Aftelklok` met welke methode dit kan.

b) De bijgewerkte tijd van ons klokje zet je in het tekstvak op het formulier.

❖ Met de knop `opnieuwOpStarttijdButton` willen we het klokje weer op de starttijd van *1 minuut 30 seconden* zetten.

Laat daarom een methode die reageert op de gebeurtenis "klikken op de knop `opnieuwOpStarttijdButton`" invoegen via het venster Properties en schrijf er de nodige programmacode om de tijd van ons veld `_klokje` weer op 1:30 in te stellen en dit uiteraard ook te updaten in het tekstvak.

Het formulier zou nu moeten werken zoals we vooropgesteld hebben. Al missen we misschien een extra knopje waarmee we het aftellen kunnen stoppen (en opnieuw opstarten). Nog even geduld (tot hoofdstuk 4) en ook dit zal voor jullie een piece of cake zijn.

2.8 Omdat missen menselijk is

Misschien maken we jullie door volgende stelling te poneren een illusie armer, maar wij zijn overtuigd dat **foutloos programmeren onmogelijk is**!

Eén zwak moment en een programmeerfout is zo gemaakt. Achteraf dergelijk foutje moeten opsporen en verbeteren daarentegen kan hard en langdurig labeur zijn. En hoeveel fouten blijven er niet ongemerkt in programma's staan omdat deze in een (te korte) testfase niet bloot kwamen te liggen, om dan achteraf toch genadeloos toe te slaan?

Daarom willen we, nog aan het prille begin van deze cursus, even stilstaan bij waar het tijdens het programmeren allemaal spaak kan lopen.

Visual Studio is een **professionele programmeeromgeving**. Dat betekent dat Visual Studio steeds een oogje in het zeil zal houden als we aan het programmeren zijn en ons enkele heel sterke tools aanreikt om fouten efficiënt op te sporen.

Als teaser bij dit stukje leerstof willen we jullie echter eerst een *vijf minuten challenge* voorschotelen.

Bij de brongegevens van het 2e hoofdstuk vind je een project `GekkeWekker`. Als je dit project opstart, wordt een formulier geopend met de welbekende wekker-toepassing. Als je met het **'+'-knopje aan de rechter zijde** de minuten laat opklimmen, zal je plots een heel vreemde sprong merken: Van 00:09 gaat het naar 00:01 en dan pas weer correct naar 00:11. Ook bij het **'-'-knopje** rechts in het formulier, loopt het op dat tijdstip verkeerd.

Jullie krijgen exact vijf minuten om in de code de corrupte instructie(s) aan te wijzen die verantwoordelijk is (zijn) voor deze fout!

Start!

…

Voilà, de vijf minuten zijn voorbij.

Heb je de fout nog niet gevonden(?), dan laten we je nog even op je honger zitten; het antwoord volgt pas later. Kon je het probleem wel aanwijzen, dan vragen we om de foute instructie nog even ongemoeid te laten. Verder willen we via het **debuggen** van code immers nog eens naar deze fout speuren.

2.8.1 Syntaxfouten

De **syntax** (of **syntaxis**) zijn de *taalregels* van een programmeertaal. Voorbeelden van regels in de C# programmeertaal die je strikt moet navolgen, zijn:

- gebruik puntkomma's om je statements af te sluiten;
- een haakje dat je geopend hebt, zal je ook ooit moeten sluiten;
- een accolade die je geopend hebt, zal je ook ooit moeten sluiten;
- geef het juiste aantal parameters op bij het aanroepen van een methode;
- gebruik de juiste gegevenstypen voor je parameters.

Je hebt dan natuurlijk ook nog de *stomme* tikfouten. Omdat C# hoofdlettergevoelig is, betekent een verkeerde hoofd- of kleine letter al onmiddellijk een (*spel*)fout tegen de syntax.

Het leuke nieuws is dat Visual Studio meestal goed in staat is om dergelijke syntaxfouten op te merken. We worden bijna bij het handje genomen om deze fouten uit onze code te halen. Syntaxfouten bezorgen ons dus zelden kopzorgen. Dit zullen niet de harde noten om te kraken zijn.

We geven aan de hand van het project `VerrekteWekker` wat duiding (zie het mapje **_hoofdstuk_2/_voorbeelden**).

Dit project is een nieuwe kopie van onze wekker-toepassing. We hebben echter de code een beetje gesaboteerd. Lang gaan we deze keer niet naar de fouten moeten zoeken, want Visual Studio drukt ons meteen met de neus op de feiten!

In de Code weergave van het formulier `FormWekker` valt één methode onmiddellijk op:

```
44      private void minuutOmhoogButton_Click(object sender, EventArgs e)
45      {
46          _wekker.minutenPlus()
47
48          // tijdstip in het tekstvak updaten
49          UpdateWekkerInFormulier);
50      }
```

Met rood gekronkelde lijnen worden een drietal fouten aangeduid.

Oefening 2.5 Kan je, *zonder de fouten al te verbeteren*, noteren wat de drie mankementen zijn in bovenstaand stukje code:

Visual Studio is behulpzaam, want als we met de cursor op zo'n rode lijn te gaan staan, krijgen we bijkomende informatie waarom de instructie nog niet wordt goedgekeurd.

We geven twee voorbeeldjes:

- Als je bij de eerste instructie, met de cursor op het rode lijntje na de ronde haken gaat staan, slaat het infovenstertje de nagel op de kop.

  ```
  _wekker.minutenPlus()
                       ; expected
  // tijdstip in het tekstvak up
  ```

  We waren inderdaad de puntkomma vergeten.

- Het zal echter niet altijd zo duidelijk zijn. Het infokadertje bij de rode lijn onder de `minutenPlus()`-methode, vraagt bijvoorbeeld al heel wat meer leeswerk.

  ```
  _wekker.minutenPlus()
  // tijds
  UpdateWe
  ```

  'Wekker' does not contain a definition for 'minutenPlus' and no extension method 'minutenPlus' accepting a first argument of type 'Wekker' could be found (are you missing a using directive or an assembly reference?)
  Show potential fixes (Alt+Enter or Ctrl+;)

  Misschien moeten we deze te lange boodschap, eens in aparte stukjes gaan ontleden …

- "*'Wekker' does not contain a definition for 'minutenPlus'*"

  Dit kunnen wij vertalen naar: "De klasse `Wekker` bevat geen definitie voor de methode `minutenPlus`".

  Dit is inderdaad ons probleem. Wij hebben de klasse `Wekker` immers voorzien van de methode `MinutenPlus` met hoofdletter M. We maakten hier dus een *stom* tikfoutje.

- *"and no extension method 'minutenPlus' …"*

  "Extention methoden" zijn een bijzondere vorm van methoden binnen een klasse. Met onze, nog beperkte programmeerkennis, is dit voor ons eigenlijk Chinees. *Dit soort methoden zullen trouwens niet aan bod komen in deze cursus.*

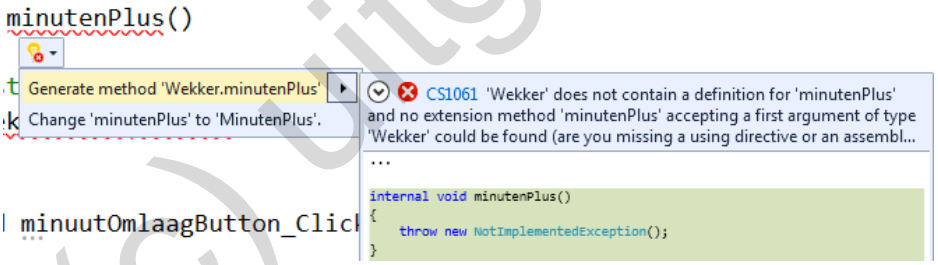
- *"(are you missing a using directive …)"*

  Dit stukje van de foutboodschap tipt ons dat we misschien een using-statement vergeten zijn bovenaan in de codepagina, wat hier dus niet het geval is.

We concluderen dat de helpteksten die Visual Studio biedt, af en toe wat dubbelzinnig, of voor ons moeilijker te interpreteren, zullen zijn. Visual Studio moet soms een beetje *raden* naar wat we fout gedaan hebben. Hoe meer ervaring je opdoet, hoe beter je deze foutteksten zal kunnen lezen.

Dan nog een klein trucje:

Als je met de cursor op het eerste rode lijntje gaat staan, verschijnt er een knop met een lampje, waarmee je een menu kan openklappen:



Hiermee kunnen we Visual Studio zelf de reparatie laten uitvoeren.

❖ Door op de tweede optie "*Change 'minutenPlus' to 'MinutenPlus'.*" te klikken, zal Visual Studio automatisch onze tikfout verbeteren.

❖ Opgelet! Met het eerste voorstel "*Generate method 'Wekker.minutenPlus'.*" zou er aan de klasse `Wekker` een nieuwe methode `minutenPlus()` (met kleine letter m) toegevoegd worden. Dit is hier natuurlijk helemaal niet de bedoeling!

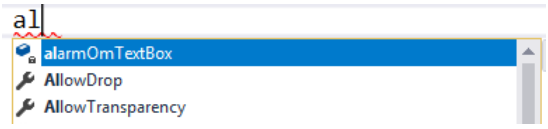
Wees dus voorzichtig met deze automatische reparaties! Omdat Visual Studio niet altijd correct raadt wat we fout deden, worden soms verkeerde remedies voorgesteld.

Onthoud 2.6 Fouten die je maakt tegen de regels van de programmeer*taal*, noemt men **syntaxfouten**. Deze fouten worden door Visual Studio meestal al ontdekt en aangeduid tijdens het programmeren.

Om ons te assisteren bij het programmeren, om ons te helpen fouten te vermijden, doet Visual Studio natuurlijk nog veel meer dan enkel rode strepen onder onze code zetten.

Een kleine greep:

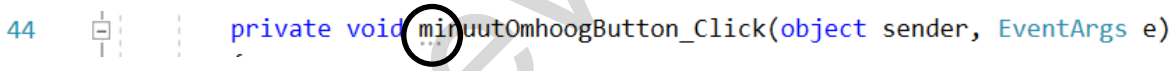
- Als we ergens een rond haakje (of een accolade, of …) openen, krijgen we er het sluitende haakje (of accolade, of …) gratis bij.
- Je hoeft nog maar enkele lettertjes ingetikt te hebben, of er popt al een lijstje op met mogelijke suggesties om je tekst aan te vullen.



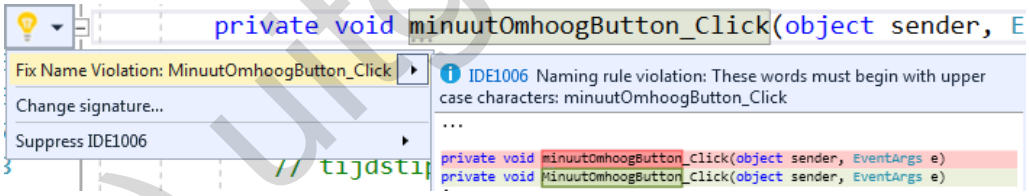
De **IntelliSense**-functie van Visual Studio regelt dit.

- Ook straf is dat we soms opmerkingen krijgen over onze **programmeerstijl**.

De drie grijze puntjes onder de methode `minuutOmhoogButton_Click` (in de Code weergave van `WekkerForm`) wijzen erop dat Visual Studio iets te melden heeft.



Als je met de cursor in deze regel gaat staan, kan je links opnieuw een menu uitklappen:



Visual Studio blijkt m.a.w. op de hoogte van het feit dat methoden in C# PascalCase genoteerd worden (dus met een starthoofdletter) en vraagt ons of hij dit mag fixen!

Wij hebben voor onszelf al goedgepraat dat methoden die reageren op een Event (hier de gebeurtenis `Click`), met een kleine letter mogen beginnen, dus we negeren dit.

Dat Visual Studio een professionele IDE is, merk je ook aan het feit dat je de programmeer-omgeving helemaal kan finetunen. Bij Tools | Options | Text Editor | C# kan je heel wat hulpmiddelen die we hier besproken hebben, in- of uitschakelen en daarenboven nog ongeveer *duizend* andere zaken instellen.

2.8.2 Build Errors

We gaan ervan uit dat je in het project `VerrekteWekker` bij de Code weergave van `FormWekker`, de syntaxfouten bij de methode `minuutOmhoogButton_Click` nog niet verbeterd hebt.

Je hebt er dus volgende situatie, met drie rode golvende markeringen:

```
44      private void minuutOmhoogButton_Click(object sender, EventArgs e)
45      {
46          _wekker.minutenPlus()
47
48          // tijdstip in het tekstvak updaten
49          UpdateWekkerInFormulier);
50      }
```

We vragen ons af wat er zal gebeuren als we toch proberen om dit project op te starten.

Tuurlijk dat we nog problemen krijgen! De volgende boodschap verschijnt:

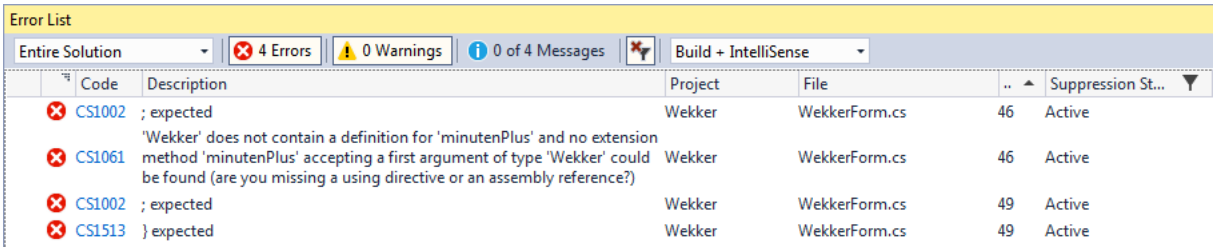
In dit venster is er sprake van **build errors**.

Wanneer je je toepassing wil uitvoeren, dan moet er eerst een uitvoerbare versie van je code gebouwd worden. Een onderdeel van Visual Studio, meer bepaald de compiler, vertaalt hiervoor je C#-code naar code die rechtstreeks door de computer kan uitgevoerd worden. Dit proces noemen we het **compileren** van code of het **builden** van de toepassing.

De compiler kan slechts vertalen wat hij zelf begrijpt en volgt hierbij strikt de syntax van de C#-taal toe. Als jij die regels ergens overtreedt, kan de compiler de vertaalslag niet maken, die dit laat weten in de vorm van build errors.

In zo'n situatie moet je altijd eerst de errors uit de code halen. In bovenstaand dialoogvenster **klik je dus op No**.

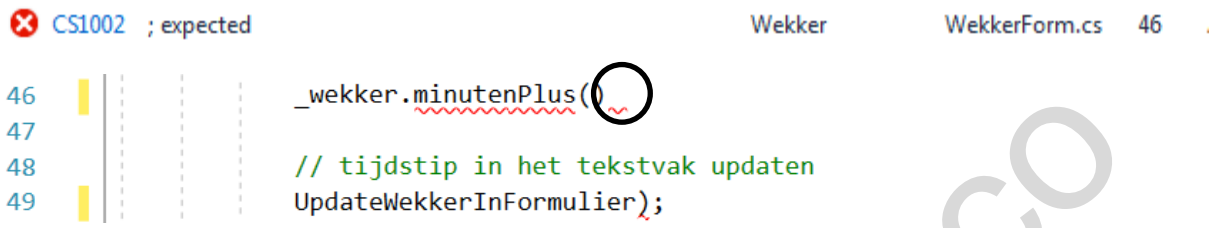
Onderaan in Visual Studio zal een **Error List venster** verschijnen met alle opgemerkte build errors:



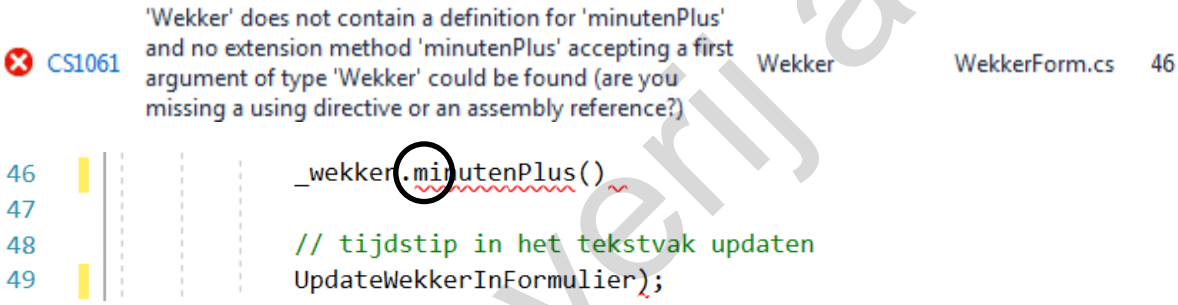
Deze build errors kan je natuurlijk linken aan de syntaxfouten die met rode lijntjes aangeduid werden in onze programmacode. Handig is dat je in het Error List venster kan dubbelklikken op een build error, waarmee je in je code onmiddellijk naar de betreffende instructie springt.

We maken even het rondje van de build errors in de Error List.

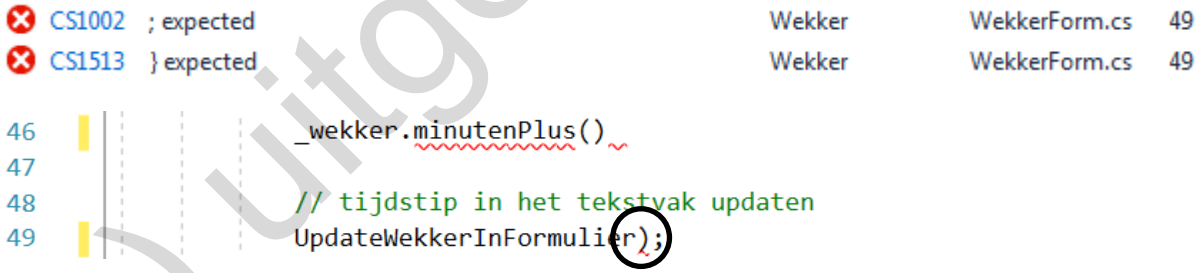
- De eerste build error verwijst natuurlijk naar de ontbrekende puntkomma.



- De tweede build error gaat over onze tikfout bij het oproepen van de methode `MinutenPlus()`. Deze methode moet immers met een hoofdletter starten.



- De twee laatste build errors verwijzen beiden naar programmaregel 49.



  Voor ons is duidelijk dat in de betreffende instructie een openend haakje ontbreekt.

  Misschien een beetje vreemd, maar in de Error List kon Visual Studio dus blijkbaar niet de vinger op de juiste wonde leggen. Daar wordt immers gemeld dat er een puntkomma en een sluitende accolade ontbreekt.

Voor Visual Studio is het niet altijd makkelijk in te schatten wat je waar precies verkeerd deed. Houd er dus rekening mee dat je niet klakkeloos mag geloven wat er in de Error List staat.

Soms maak je op regel x een fout, maar wordt die pas op regel y gedetecteerd en aangeduid. Soms wordt door een enkele syntaxfout, de code die erna volgt ook niet meer correct geïnterpreteerd, waardoor er plots meerdere build errors in het venster verschijnen.

Jij zou alvast geen probleem mogen hebben om de drie syntaxfouten nu definitief te verbeteren. Als je dan de toepassing start, mogen geen build errors meer opdoemen.

Dit betekent dan meteen dat de compiler het project nu zonder problemen kon builden en dat het project (of het formulier `FormWekker`) kan gestart worden!

Onthoud 2.7 Om een Visual Studio toepassing te kunnen starten, moet de C#-programmacode eerst omgezet worden naar een door de computer uitvoerbaar bestand. Dit heet het **compileren** van de code of het **builden** van de toepassing.

Dit proces kan niet met succes beëindigd worden als er nog (syntax)fouten in je code staan.

Dergelijke fouten, die men dan **build errors** noemt, zullen in het **Error List venster** opgelijst worden. Zorg dat je eerst en vooral deze fouten verbetert!

Opmerking:

- We raden het absoluut niet aan, maar wat gebeurt er eigenlijk als je bij het build error-dialoogvenster op de knop **Yes** klikt?

  De build errors in je code worden dan genegeerd en er wordt nagekeken of de compiler er vroeger eens in geslaagd is om een build te maken (zie "*the last successful build*"). Het uitvoerbaar bestand dat die *oude* build opleverde, wordt dan opgestart (zie "*run*").

  Kort samengevat: omdat je huidige code fouten bevat, wordt er gekeken of er misschien ergens een oude versie beschikbaar is die wel opgestart kan worden.

  Bij het schrijven en testen van code, kan je dus weinig aanvangen met die Yes-knop. Alle nieuwe code die je toevoegt, wordt zo immers straal genegeerd (tot je alle build errors in je code verbeterd hebt).

  Dus één vuistregel bij bovenstaand dialoogvenster: **Nooit Yes; Altijd No -> en dan zo snel mogelijk alle build errors uit je code halen**.

Een bijzondere build error

Er zijn van die stomme stoten die iedereen in zijn leven wel één keer meemaakt. Tegen een gesloten - kraaknet gepoetste - glazen deur aanlopen, is zo de eerste die bij ons opkomt.

Bij het werken met formulieren in Visual Studio bestaat er ook zo'n klassieke *stomme stoot* variant. Omdat deze een build error oplevert die wat moeilijker op te heffen valt, is dit hier misschien de ideale plaats om jullie hiertegen te wapenen.

Het gebeurt wel eens dat je tijdens het ontwerpen van een formulier (per ongeluk) op één van de besturingselementen dubbelklikt. Het effect is dat je meteen naar de Code weergave van het formulier gaat en dat er daar een gebeurtenismethode werd gegenereerd.

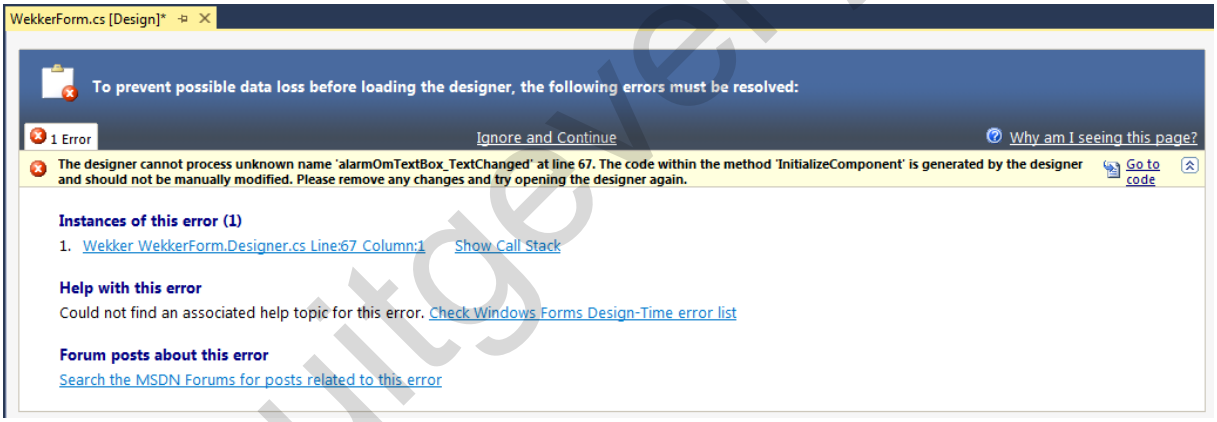
Open bijvoorbeeld eens het formulier `WekkerForm` in de Design weergave (zie het project `VerrekteWekker`) en kijk wat er gebeurt als je **dubbelklikt** op het tekstvak `alarmOmTextBox`.

De header van volgende gebeurtenismethode wordt automatisch aan de code in de klasse `WekkerForm` toegevoegd:

```
66      private void alarmOmTextBox_TextChanged(object sender, EventArgs e)
67      {
68
69      }
```

Omdat we die methode helemaal niet nodig hebben en we niet graag overbodige code zien staan, wissen we die vier lijntjes terug uit de klasse `WekkerForm`.

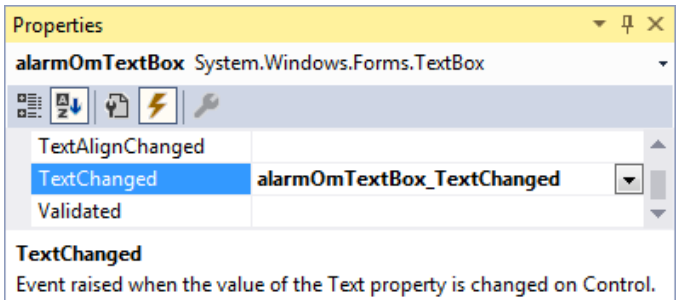
Er lijkt geen vuiltje aan de lucht; nergens rode kronkellijntjes te bespeuren, … maar als je nu het formulier `WekkerForm` probeert te openen in de Design weergave is het dikke miserie:



We krijgen dus impliciet een melding dat er een dringende fout moet opgelost worden. Zolang dit niet gebeurt, kan het formulier niet meer bekeken worden in de Design weergave!

We leggen even uit wat er zich hier allemaal (achter onze rug) afgespeeld heeft:

- Door te dubbelklikken op het tekstvak `alarmOmTextBox`, werd er in het Properties venster (knopje ) aan de gebeurtenis `TextChanged` een methode gekoppeld:



Dit is uiteraard diezelfde gebeurtenismethode `alarmOmTextBox_TextChanged()` waarvoor daarnet automatisch een header aan de klasse werd toegevoegd.

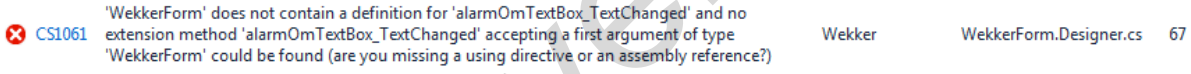
Ter informatie: `TextChanged` is een Event dat geactiveerd wordt elke keer als iets gewijzigd wordt in het tekstvak. De reden dat uit de lange lijst Events net voor deze gebeurtenis gekozen werd, is omdat `TextChanged` voor een `TextBox` het default Event is.

❖ Door in de Code weergave de header van deze gebeurtenismethode te wissen, ontstaat er een probleem!

In het Properties venster staat immers nog steeds aangegeven dat aan de gebeurtenis `TextChanged` een gebeurtenismethode `alarmOmTextBox_TextChanged()` gekoppeld is, maar die gebeurtenismethode zelf bestaat nu NIET meer.

Visual Studio raakt hiervan zo van streek, dat het zelfs niet meer mogelijk is om het formulier (in Design weergave) te openen. We kunnen m.a.w. ook niet meer aan het Properties venster om het probleem daar op te lossen!

❖ Het project opstarten, lukt zeker niet meer. In het Error List venster krijgen we hiervoor volgende build error:



De vraag is nu natuurlijk hoe we uit deze catch 22 raken. Wij doen het gewoonlijk met de volgende strategie:

In het foutvenster dat verschijnt als je de Design weergave van het formulier probeert te openen, klik je op de link 'Go to code'.

Je komt in een voor ons nog onbekend stukje code terecht, waar een rood gemarkeerde instructie staat:

```
66      this.alarmOmTextBox.TabIndex = 2;
67      this.alarmOmTextBox.TextChanged += new System.EventHandler(this.alarmOmTextBox_TextChanged);
68      //
```

Wis die betreffende instructie. In onze screenshot moet dus de **volledige regel** 67 weg!

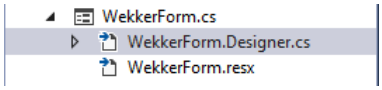
Opmerking:

○ (Enkel) voor de meerwaardezoeker die het plaatje nog wat vollediger wil hebben.

Op pagina 43 hadden we het over het sleutelwoord **partial** bij de header van een klasse, waarmee je de code die een klasse definieert over meerdere bestanden kan spreiden.

Elke formulierklasse (dus ook de klasse `WekkerForm`) maakt gebruik van deze techniek.

De instructie die we net gewist hebben, stond in het bestand **WekkerForm.Designer.cs**. Dit bestand is zo'n uitbreiding van de klasse `WekkerForm`. Om dit bestand in de Solution Explorer te vinden, zal je het menu bij het formulier `WekkerForm` even open moeten klappen:



De gewiste instructie hoorde in het bestand **WekkerFormDesigner.cs** bij de methode `InitializeComponents()`. Dit is de methode die altijd opgeroepen wordt bij de constructor van een formulierklasse. Op pagina 43 vertelden we dat deze - automatisch gegenereerde - methode `InitializeComponents()` verantwoordelijk is om het formulier op te bouwen (bijvoorbeeld de tekstvakken, knoppen, … op het formulier te zetten).

Nu geweten is waar de methode `InitializeComponents()` te vinden is, mogen de geïnteresseerden zeker de code er al eens bestuderen. Misschien kan je wel al een beetje raden wat deze instructies doen, maar eigenlijk moet je hoofdstuk 4 gerond hebben, om deze code beter te kaderen.

Je mag in principe de code in de methode `InitializeComponents()` aanpassen, maar als je nog niet goed weet wat je doet, blijf je er beter af. Een complicatie in deze code kan immers ernstige gevolgen hebben voor je formulier!

2.8.3 Logische fouten

Dat er tijdens het builden (compileren) van een toepassing geen build errors opduiken, is nog geen garantie dat je een foutloos programma afgeleverd hebt.

Als je een formulier ontwerpt waarmee je de prijs van één of andere aankoop laat bepalen, maar je hebt bij de achterliggende berekening een verkeerde formule gehanteerd, dan krijg je als eindresultaat natuurlijk niet het correcte bedrag te zien.

In deze situatie zal er bij onze code waarschijnlijk geen *rood lijntje* onder de foute formule getekend worden. Hoe zou Visual Studio moeten vermoeden dat we bijvoorbeeld een verkeerd BTW-percentage toegepast hebben?

Of nog een voorbeeldje: Stel dat de regering, om te verhinderen dat tientonners in een schoolomgeving te snel gaan rijden, volgend stukje code verplicht laat inbouwen bij de tachograaf van vrachtwagens:

```
if (zone30 == true || snelheid > 30) {
    boete = 100;
}
```

Heb je de fout opgemerkt?

Met deze code zal elke keer als de vrachtwagen gewoon maar een zone 30 binnenkomt, er automatisch al een boete van 100 euro aan vasthangen. Ook telkens de snelheid boven de 30 gaat (al gebeurt dit op een autostrade) wordt de chauffeur sowieso beboet.

Beide voorwaarden in de if mochten hier niet aan elkaar gekoppeld worden met een **||** (de **or**-operator), maar dit moest natuurlijk met een **&&** (de **and**-operator) gebeuren. De boete mag immers enkel uitgereikt worden als aan beide voorwaarden voldaan is (of je bevindt je in zone30 **EN** je rijdt er sneller dan 30 km/uur).

Omdat er in bovenstaand stukje code echter geen fout tegen de *taalregels* van C# gemaakt is, zal er voor de compiler, die de code moet builden, NIETS aan de hand zijn. Tegen de chauffeur, die onder boetes bedolven werd, zal je daarentegen niet kunnen volhouden dat er *geen fouten* in de code staan!

Als er iets mis is met de logica van je code, dan spreekt men van een **logische fout**. Omdat dergelijke fouten NIET door Visual Studio (door de compiler) ontdekt kunnen worden, zullen ze het de programmeur veel moeilijker maken dan de *onschuldige* syntaxfoutjes.

De enige goede remedie tegen logische fouten is uitgebreid testen, dan nog eens testen, dan eens de speciale gevallen testen, …

Onthoud 2.8 Als je niet zondigt tegen de taalregels (de syntax) van de programmeertaal, maar er is iets mis met de logica van je code, dan spreekt men van een **logische fout**.

Visual Studio zal logische fouten per definitie niet herkennen en dus ook niet aanduiden.

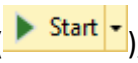
Je vermijdt dergelijke fouten door zorgvuldig je code (= je klassen) uit te denken, geconcentreerd te programmeren en je toepassing uitvoerig te testen. Analyse, implementatie en testen zijn dan ook de drie noodzakelijke fasen die bij een succesvolle softwareontwikkeling gelijkwaardig aan de beurt moeten komen.

Maar ook al heb je bij het testen gemerkt dat jouw programma soms fout reageert, daarmee zal niet altijd automatisch duidelijk zijn *waar* in je code je *wat* fout gedaan hebt (laat staan dat je zomaar een oplossing kan bedenken). Mocht het zo gemakkelijk zijn, dan haalde iedereen op een toets programmeren altijd alle punten.

Weet ook dat professionele toepassingen vlug heel veel code zullen bevatten. In de digitale wereld schrikt men niet van programma's die uit duizenden instructies bestaan. Haal daar als broekje maar de fouten uit.

Visual Studio (en elke andere programmeeromgeving die naam waardig), is daarom uitgerust met enkele tools waarmee je grondig en gedetailleerd je code kan inspecteren. In het laatste onderdeel van dit hoofdstuk laten we jullie in dit kader al eens kennismaken met de mogelijkheden om je code te debuggen!

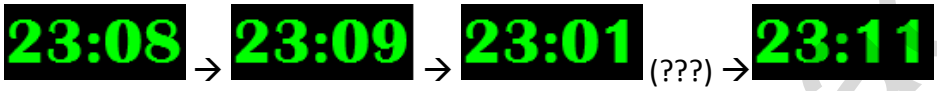
2.8.4 Code debuggen

Als je het project `GekkeWekker` (zie het mapje **_hoofdstuk_2/_voorbeelden**) opent en dan opstart (

), dan wordt het ons bekende formulier `FormWekker` mooi geopend. Een eerste conclusie is dat er in dit project dus geen sprake is van *syntaxfouten* of *build errors*.

Als je daarna een beetje met de knoppen op het formulier gaat spelen, valt toch op dat één en ander niet pluis is:

- Het '+'-knopje aan de linkerzijde om de uren te verhogen, doet blijkbaar net het omgekeerde, namelijk de uren verminderen.
- Als je met het '+'-knopje aan de rechterzijde de minuten laat opklimmen, wordt plots een heel vreemde sprong gemaakt:

  Dit is trouwens de *vijf minuten challenge* die we jullie op pagina 63 al eens voorlegden.

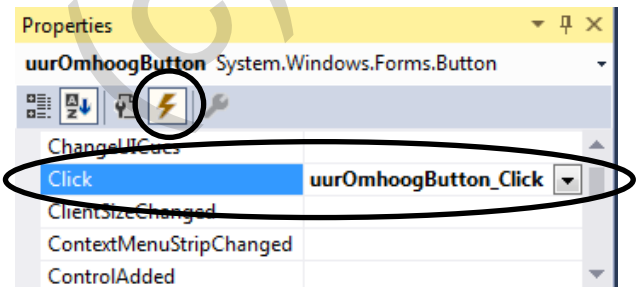
We trekken de nieuwe conclusie dat het project `GekkeWekker` dus niet vrij is van **logische fouten**.

Omdat het hier maar gaat om een relatief miniproject, is het zeker te doen om zonder bijkomende tools, rechtstreeks in de code, de fouten op te sporen en te verbeteren. Wij willen deze case echter aangrijpen om uit te leggen hoe je code moet **debuggen**, dus we laten de fouten nog even ongemoeid (en komen die straks wel weer ergens tegen).

Een onderbrekingspunt in je code plaatsen

We weten dat er iets fout is met het '+'-knopje aan de linkerzijde. Deze knop zou het uur moeten verhogen, maar om één of andere reden doet die net het omgekeerde.

Aan deze knop `uurOmhoogButton` hebben we bij het Event `Click` een methode laten koppelen. Toch eerst even controleren in het Properties venster (bij de Events) of dit wel zo is:



Onderstaande methode zal dus opgestart worden bij het klikken op deze knop `uurOmhoog-Button`:

```
28      private void uurOmhoogButton_Click(object sender, EventArgs e)
29      {
30          _wekker.UrenMin();
31
32          // tijdstip in het tekstvak updaten
33          UpdateWekkerInFormulier();
34      }
```

We willen stap voor stap nagaan wat er hier precies gebeurt en daarom plaatsen we een **onderbrekingspunt** (**breakpoint**) bij deze methode.

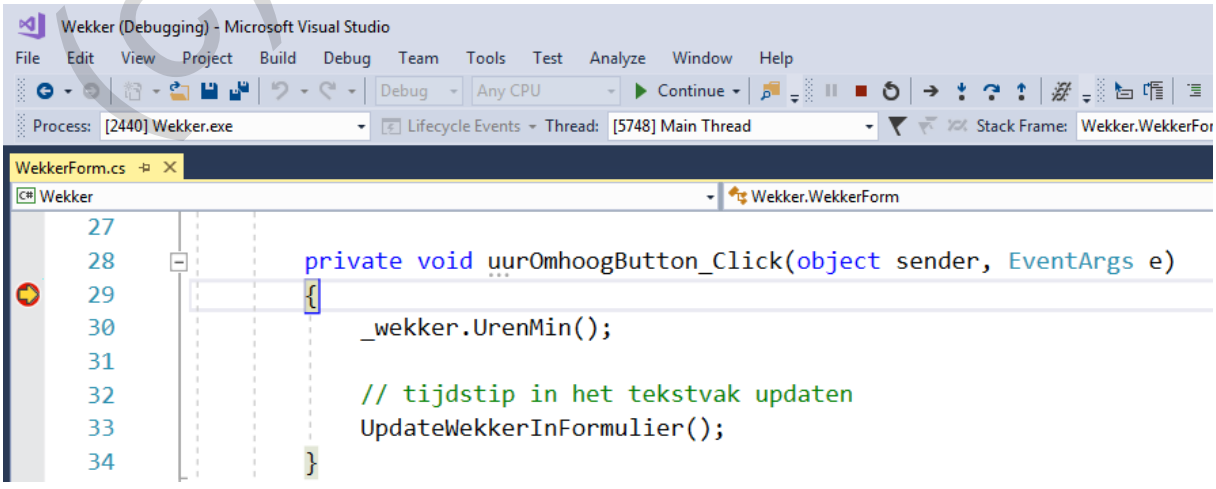
Je doet dit door te klikken in het grijze balkje in de linkermarge bij de regel met de header van deze methode. Er wordt daar dan een rode bol gezet:

```
28      private void uurOmhoogButton_Click(object sender, EventArgs e)
29      {
30          _wekker.UrenMin();
31
32          // tijdstip in het tekstvak updaten
33          UpdateWekkerInFormulier();
34      }
```

De betekenis hiervan is dat als bij het uitvoeren van de code, aan zo'n onderbrekingspunt gekomen wordt, het programma daar gepauzeerd wordt. De rode bol mag je dus interpreteren als een rood licht waar het programma niet voorbij mag.

Start het project opnieuw op (Start) en klik zeven maal op de '-'-knop om het uur te verminderen. De wekker komt op 17:00 staan, maar verder gebeurt er nog niets bijzonders. Ons breakpoint staat dan ook niet bij de code die uitgevoerd wordt als je op de '-'-knop `uurOmlaagButton` klikt.

Maar zodra je aan de linkerzijde van het formulier op de '+'-knop `uurOmhoogButton` klik, geschiedt het wonder:

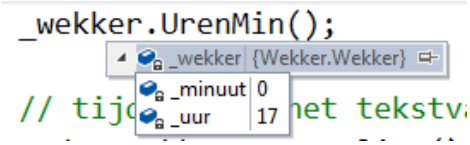


Er gebeurt van alles:

- Visual Studio spring terug naar de programmacode van je toepassing.
- De titelbalk van Visual studio "Wekker (**Debugging**)" geeft aan dat we in de **Debug modus** terecht gekomen zijn. Op dit moment neemt de **debugger** het commando over.
- Binnenin onze rode bol verschijnt een gele pijl. Met deze gele pijl geeft de debugger aan dat hij de uitvoering van de code op die plaats onderbroken heeft.

Terwijl Visual Studio in de Debug modus staat, kan je door op een variabele, een veld, een object, … te gaan staan, de **huidige waarde** van dat element opvragen.

Zet de cursor bijvoorbeeld eens op het businessobject `_wekker` (en laat via het pijltje het menu openklappen) en je krijgt wonderwel de huidige toestand van dit object te zien:



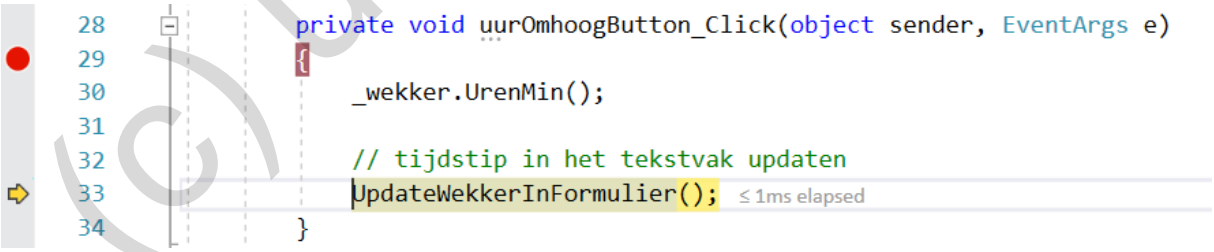
Dit venstertje vertelt ons dat op dit eigenste moment bij het businessobject `_wekker`, het veld `_minuut` op 0 en het veld `_uur` op 17 staat.

De debugger wacht op ons bevel om verder te doen.

F10: Step Over

Met de **F10**-toets kan je het programma **stap** per **stap** (**instructie** per **instructie**) verder laten lopen. Eén keer op F10 klikken, betekent dat de volgende instructie uitgevoerd wordt, maar het programma dan onmiddellijk weer gepauzeerd wordt.

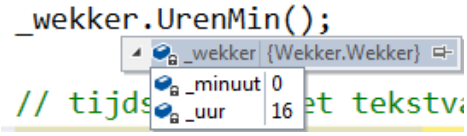
Heb je tweemaal op F10 geklikt, dan komt de gele pijl twee instructies lager te staan:



Opgelet: de instructie die aangewezen wordt door de gele pijl, is nog **net NIET** uitgevoerd! De uitvoering van het programma is dus onderbroken, **net vóór** de gemarkeerde instructie!

In bovenstaande screenshot is de instructie "`_wekker.UrenMin();`" dus al wel, maar de instructie "`UpdateWekkerInFormulier();`" nog net niet, gebeurd!

Omdat de methode `UrenMin()` het object `_wekker` laat wijzigen, zouden we dit bij de toestand van dit object moeten opmerken. En inderdaad, ga maar ergens in de code op het object `_wekker` staan. Het veld `_uur` blijkt aangepast:



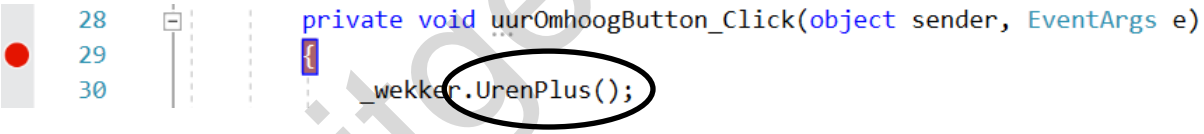
Klik je dan nog eens op F10, komt de pijl bij de sluitende accolade te staan. De instructie "`UpdateWekkerInFormulier();`" zit er dan ook op.

Met de volgende F10 is de methode helemaal afgewerkt. De Debug modus wordt hiermee beëindigd (of m.a.w. de debugger stopt ermee). In Visual Studio komen we weer terecht in ons wekkerformulier (waar het uur ook aangepast is).

De aanleiding om dat stukje code te inspecteren (via een onderbrekingspunt en de F10-toets) was het ongewenste gedrag van het '+'-knopje `uurOmhoogButton`. Het uur zou hiermee eentje verhoogd moeten worden, maar in de praktijk bleek het omgekeerde te gebeuren (er gaan een uur af).

Tijdens het debuggen kon je mooi meevolgen dat de code inderdaad de wekker van 17 naar 16 uur verdraaide.

We hadden hier natuurlijk zelf om gevraagd door de methode `UrenMin()` op te roepen. Hoe stom van ons, dit moest uiteraard `UrenPlus()` zijn. Verbeter de code:



Om dit verhaaltje helemaal af te sluiten, laat je ook het onderbrekingspunt verwijderen. Dit doe je door in de grijze balk nog eens in de betreffende rode bol te klikken.

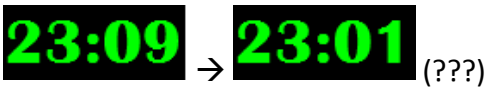
F11: Step Into

Toen we hierboven, met de debugger, de methode `uurOmhoogButton_Click` inspecteerden, lieten we met twee klikken op de F10-toets respectievelijk de instructies "`_wekker.UrenMin();`" en "`UpdateWekkerInFormulier();`" behandelen.

Maar `UrenMin()` (nu verbeterd naar `UrenPlus()`) is een methode van de klasse `Wekker` die op zich ook heel wat code bevat. `UpdateWekkerInFormulier()` is een hulpmethode in het formulier die ook uit verschillende instructies bestaat. Wat als de fout ergens binnenin deze methoden gemaakt werd?

Daarom moeten we zeker nog eens demonstreren hoe je met de **F11**-toets de debugger **een niveau dieper** in je code laat neuzen.

Om het nuttige aan het aangename te koppelen, proberen we terzelfdertijd het mysterie van de '+'-knop `minuutOmhoogButton` op te lossen, dat onverklaarbaar de wekker van 23:09 naar 23:01 laat transformeren (terwijl we hier 23:10 verwachten).



(???)

Om onze test op te zetten gaan we, zonder al onderbrekingspunten te gebruiken, de wekker op het formulier op 23:09 zetten. Voor wie *heel duidelijke* instructies nodig heeft:

- start het project, of dus het formulier `WekkerForm`, op;
- klik één keer op de '-'-knop links om de klok op 23 uur te zetten;
- klik negen keer op de '+'-knop rechts om de wekker op 23:09 te zetten.

We weten dat het foutloopt als we de volgende keer op het '+'-knopje `minuutOmhoog-Button` klikken. Dus de instructies die uitgevoerd worden bij de methode `minuutOmhoog-Button_Click` verdienen een grondig onderzoek!

Nu zetten we dus ons onderbrekingspunt bij de hoofding van deze methode:

```
44      private void minuutOmhoogButton_Click(object sender, EventArgs e)
45      {
46          _wekker.MinutenPlus();
47
48          // tijdstip in het tekstvak updaten
49          UpdateWekkerInFormulier();
50      }
```

Merk dus op dat je onderbrekingspunten kan plaatsen (en ook terug verwijderen), terwijl de toepassing aan het draaien is.

Klik je nu in het formulier op het betreffende '+'-knopje om de minuten te verhogen, komen we terug in de Debug modus terecht (daar is de gele pijl weer):

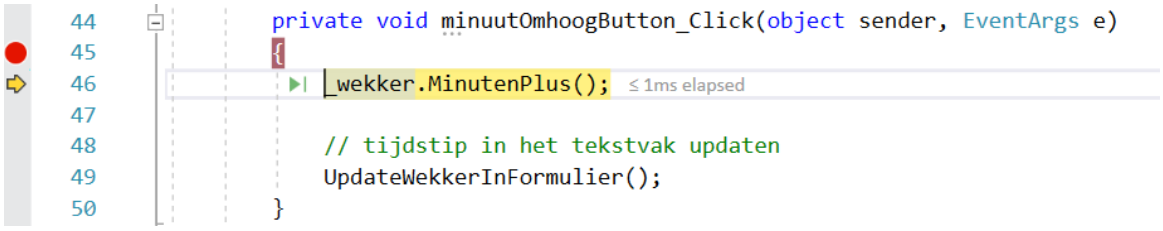
```
44      private void minuutOmhoogButton_Click(object sender, EventArgs e)
45      {
46          _wekker.MinutenPlus();
47
48          // tijdstip in het tekstvak updaten
49          UpdateWekkerInFormulier();
50      }
```

Met de F10-toets laat je de debugger door de instructies van een methode *razen*. Je zal onmiddellijk merken hoe je met de F11-toets de debugger veel kleinere stapjes laat maken.

Omdat het met kleine stapjes langer duurt tot je de eindbestemming bereikt, hebben we hieronder wel wat pagina's nodig om alles uit te schrijven. Goed kunnen debuggen is een heel belangrijke skill, dus we maken hier graag wat plaats voor in onze cursus.

De debugger is ondertussen ongeduldig aan het wachten op onze instructies. Laat ons er dus maar meteen aan beginnen.

❖ Klik je een eerste keer op F11 komt de gele pijl op de instructie `_wekker.Minuten-Plus()` te staan. Let op: dit betekent dat deze instructie dan nog net NIET uitgevoerd is.

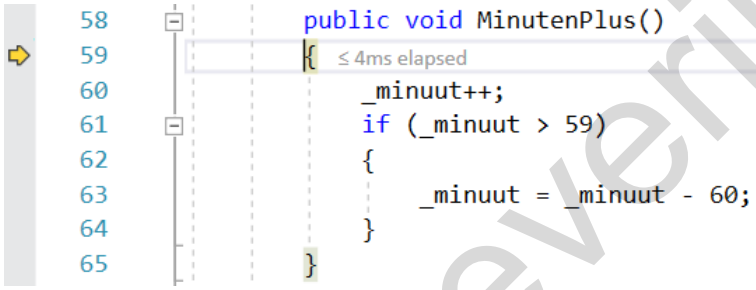


De F10 zou hier trouwens nog gewoon hetzelfde effect hebben.

❖ Als je nu nog eens F11 doet, wordt de betekenis van de **Step Into**-actie duidelijk.

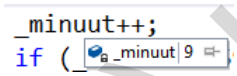
Omdat bij de aangewezen instructie een methode (`MinutenPlus()`) opgeroepen wordt, gaat de debugger nu de focus naar die methode verleggen. Zo kunnen we de code binnen die methode ook daar stap voor stap (instructie voor instructie) meevolgen.

We bevinden ons dus plots in de klasse `Wekker` bij de methode `MinutenPlus()`:

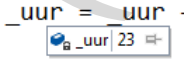


Je weet nog dat je tijdens het debuggen de variabelen, velden, objecten, … kan aanwijzen om de actuele waarde van die elementen op te vragen.

Het is dus niet moeilijk om te checken dat de wekker nu op 9 minuten staat:



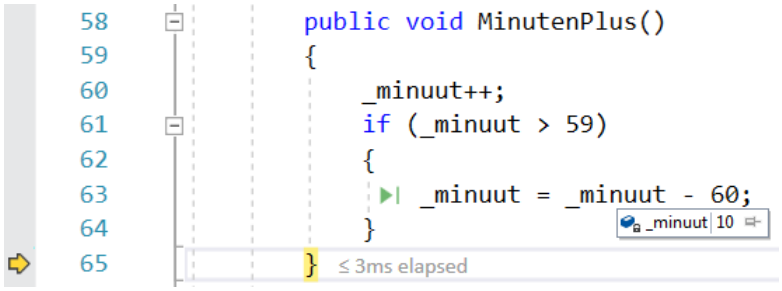
In de methode `MinutenPlus()` doen we niets met het veld `_uur`. Zonder problemen mag je ergens *in een andere methode* bij de klasse `Wekker` op het veld `_uur` gaan staat. Zoals verwacht wordt de waarde 23 getoond:



❖ De code in de methode `MinutenPlus()` kan je nu op z'n beurt lijntje per lijntje laten uitvoeren. Omdat in deze instructies geen andere methoden meer opgeroepen worden (we zitten al op het diepste niveau), maakt het niet uit of je hiervoor F10 of F11 gebruikt.

We weten dat er straks in het formulier iets raars gebeurt met de minuten, dus zijn we heel achterdochtig bij het doorlopen van deze methode.

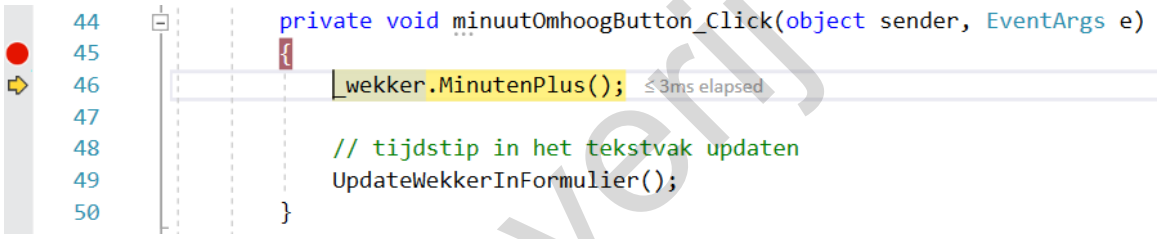
Breng de gele pijl (F10 of F11) naar de sluitende accolade:



Alles blijkt echter normaal te verlopen.

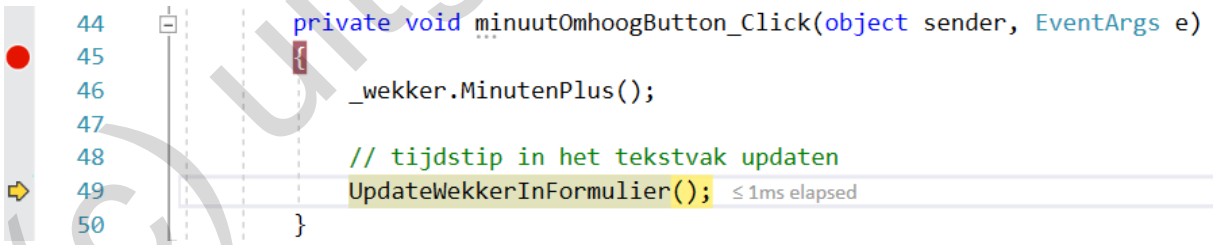
Je merkte dat we (zoals het hoort) niet in de if gesprongen hebben. De screenshot hierboven (waar we de debugger op de sluitende accolade staat) toont dat het veld `_minuut` (correct) op 10 staat.

- Omdat de methode `MinutenPlus()` erop zit, zal de volgende F11 (maar ook F10) ons terug een niveau naar boven liften:



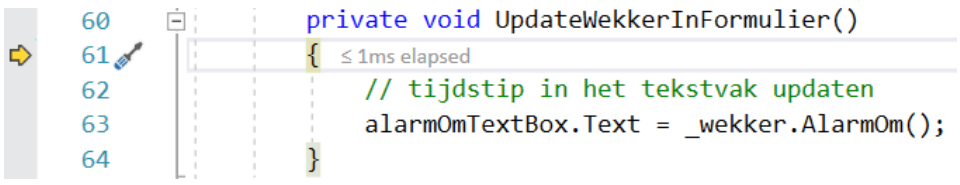
We zitten m.a.w. terug in de methode `minuutOmhoogButton_Click()`.

- Ga een stapje verder (F10 of F11), zodat de gele pijl bij de instructie "`UpdateWekkerInFormulier();`" komt te staan:

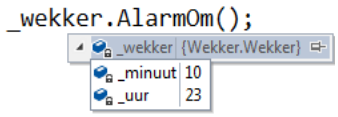


- De debugger wijst nu een instructie aan die een hulpmethode oproept. Omdat we willen weten wat er precies in deze methode gebeurt, hebben we opnieuw een F11 nodig om een Step Into-actie te forceren.

En ja, onze gele pijl verhuist naar de methode `UpdateWekkerInFormulier()`:



Ga er eerst eens op `_wekker` staan.



De wijzers van het `_wekker`-object wijzen 23:10 aan. Alles lijkt dus nog in orde.

❖ Breng de gele pijl (F10 of F11) naar de volgende instructie:

```
60      private void UpdateWekkerInFormulier()
61      {
62          // tijdstip in het tekstvak updaten
63          alarmOmTextBox.Text = _wekker.AlarmOm();
64      }
```

❖ Bij de aangewezen instructie wordt de methode `AlarmOm()` opgeroepen. We kunnen terug een Step Into doen. F11 brengt de debugger opnieuw een laag dieper in de code.

De gele pijl staat nu in de klasse `Wekker` bij de methode `AlarmOm()`:

```
76      public String AlarmOm()
77      {   ≤ 3ms elapsed
78          String minuutString;
79          String uurString;
80
81          minuutString = DisplayTweeCijfers(_minuut);
82          uurString = DisplayTweeCijfers(_uur);
83
84          return uurString + ":" + minuutString;
85      }
```

De methode `AlarmOm()` wordt gebruikt om voor onze wekker een **uu:mm-notatie** samen te stellen.

Deze methode plakt hiervoor het uur, een dubbele punt en de minuten aan elkaar. Er wordt voor gezorgd dat het uur en de minuten steeds met twee cijfers weergegeven worden.

❖ Breng de gele pijl (F10 of F11) naar de regel met nummer 81:

```
76      public String AlarmOm()
77      {
78          String minuutString;
79          String uurString;
80
81          minuutString = DisplayTweeCijfers(_minuut);
82          uurString = DisplayTweeCijfers(_uur);
83
84          return uurString + ":" + minuutString;
85      }
```

En opnieuw hebben we een instructie vast, waar er een *hulpmethode* opgeroepen wordt. Bij deze instructie wordt immers beroep gedaan op `DisplayTweeCijfers()`. Dit is de methode waarmee we indien nodig een extra nulletje voor het uur of de minuten zetten.

We staan dus klaar om na te gaan of er vóór de minuten (zie de parameter `_minuut` bij de methode `DisplayTweeCijfers()`) een extra nulletje moet komen.

Voordat we de volgende Step Into doen, nog even checken welke waarde de parameter `_minuut` heeft:

```
minuutString = DisplayTweeCijfers(_minuut);  ≤ 1ms elapsed
uurString = DisplayTweeCijfers(_uur);        _minuut 10
```

"10 minuten", alles is dus nog OK.

❖ Met de F11-toets stappen we de hulpmethode `DisplayTweeCijfers()` binnen:

```
87      private String DisplayTweeCijfers(int getal)
88      {  ≤ 1ms elapsed
89          String displayString;
90
91          if (getal > 10)
92          {
93              displayString = getal.ToString();
94          }
95          else
96          {
97              displayString = "0" + getal.ToString();
98          }
99
100         return displayString;
101     }
```

Logisch dat de parameter `getal` op de waarde 10 staat:

```
s(int getal)
          getal 10
```

❖ We gaan de uitvoering van het programma in de methode `DisplayTweeCijfers()` meevolgen. We zien geen instructies meer die andere (hulp)methoden oproepen. F10 of F11 zullen dus hetzelfde effect hebben.

We liegen een klein beetje. De `ToString()`, die enkele keren vermeld wordt, is natuurlijk ook een methode, maar deze wordt door F11 hier standaard genegeerd.

En even later staat de gele pijl bij volgende instructie:

```
87      private String DisplayTweeCijfers(int getal)
88      {
89          String displayString;
90
91          if (getal > 10)
92          {
93              displayString = getal.ToString();
94          }
95          else
96          {
97              displayString = "0" + getal.ToString();
98          }
99
100         return displayString;
101     }
```

Is dit niet bizar? Dit is de instructie om een voorloopnul voor het getal te zetten. Bij *10 minuten* is er toch GEEN voorloopnul nodig?

Ook als je deze instructie laat uitvoeren met F10 (of F11) en dan de waarde van de hulpvariabele `displayString` opvraagt, blijven we ons verbazen:

```
97          displayString = "0" + getal.ToString();
98      }   ≤ 2ms elapse  displayString "010"
99
100     return displayString;
```

De variabele `displayString` staat op "010". Dit onverwachte resultaat moet alle alarmbellen laten luiden!

We laten de debugger even rusten, of de gele pijl mag op regel 98 halt houden. We pikken straks de draad hier weer op, wanneer we het over enkele andere debugtechnieken hebben.

Maar toch eerst even stilstaan bij wat we net in de methode `DisplayTweeCijfers()` ervaren hebben!

We zijn tijdens een uitvoerige (meerlagige) debugsessie in een fase gekomen, waar het programma zich vreemd begint te gedragen. Meer nog, we kunnen zelfs het exacte moment aanwijzen, waar het lijkt mis te lopen.

Denk nog eens aan de *vijf minuten challenge* die we op pagina 63 lanceerden, waar we jullie naar een geniepige programmeerfout lieten zoeken in het `GekkeWekker`-project. Je had toen nog niet echt een clou waar de fout ergens verborgen zat.

Door de code slim te debuggen hebben we onze blik nu op een heel specifiek stukje code gericht. We hebben de methode `DisplayTweeCijfers()` in het vizier.

Nu is de *vijf minuten challenge* geen grote uitdaging meer. Hebben jullie ook de fout gevonden?

Om de uu:mm-notatie samen te stellen, moeten er voor de minuten (en voor het uur) een extra nul plaatsen als dit getal **kleiner is dan 10**. We mogen dus GEEN extra nulletje zetten als het getal **groter is dan 10 of *gelijk is aan 10***. Die laatste voorwaarde hebben we niet goed naar onze if() vertaald.

In de methode `DisplayTweeCijfers()` hebben we m.a.w. "`if (getal >= 10) {}`" nodig i.p.v. "`if (getal > 10) {}`".

Wacht echter nog even om deze fout te verbeteren. We gaan voor de fun (en om nog enkele nieuwe technieken te leren), wat verder debuggen.

Het watch-venster

Eerst brengen we in twee stapjes de debugger wat verder in het programma.

- Stuur met F10 de gele pijl naar de sluitende accolade van de methode `DisplayTwee-Cijfers()`:

```
97              displayString = "0" + getal.ToString();
98          }
99
100         return displayString;
101     }  ≤ 1ms elapsed
```

- Met de volgende F10 springen we terug een niveau naar boven. We zitten weer in de methode `AlarmOm()`.

  Ga daar met F10 ook verder naar de sluitende accolade:

```
76      public String AlarmOm()
77      {
78          String minuutString;
79          String uurString;
80
81          minuutString = DisplayTweeCijfers(_minuut);
82          uurString = DisplayTweeCijfers(_uur);
83
84          return uurString + ":" + minuutString;
85      }  ≤ 1ms elapsed
```

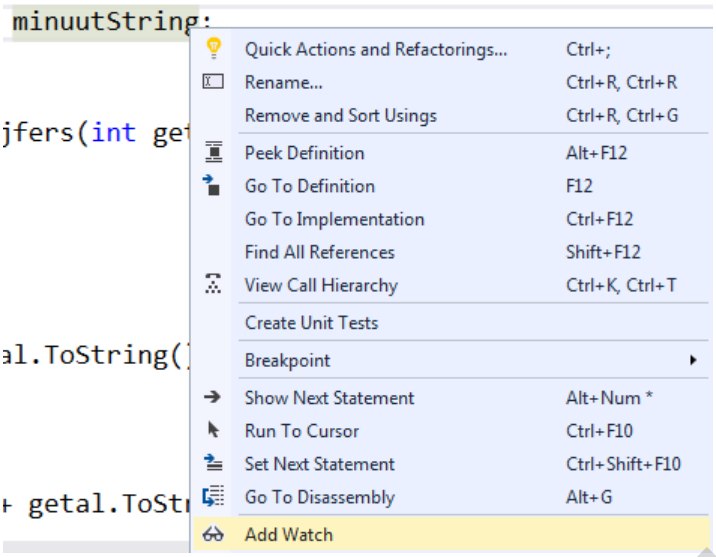
Tijdens het debuggen op een variabele, veld, object, … gaan staan, om zo de actuele waarde van het element te bekijken, is al een vaste reflex geworden. Wijs met de cursor bijvoorbeeld nog eens de variabele `minuutString` aan en in een infokadertje zie je de (foute) "010" weer verschijnen:

```
minuutString;
```
minuutString "010"

Om echter nog beter de evolutie van je variabelen, velden, objecten, … op te volgen zijn programmeurs vaak fan van het **Watch venster**.

We geven een kleine demonstratie:

❖ Klik rechts op de variabele `minuutString`. Het volgende snelmenu zal openen:

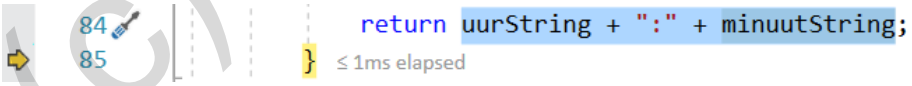


Kies je voor de optie **Add Watch**, dan wordt de variabele `minuutString` toegevoegd aan het Watch venster. Dit Watch venster zal hierna trouwens automatisch geopend worden onderaan in Visual Studio:

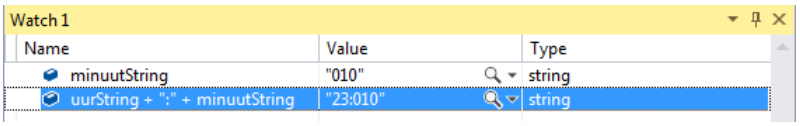
Het is handig dat je de actuele waarde van de variabele `minuutString` nu voortdurend in het zicht hebt!

❖ Een ander groot voordeel van het Watch venster is dat je er ook volledige expressies (die een waarde hebben) kan plaatsen.

Selecteer in je programmacode de expressie waar de uren en minuten aan elkaar geplakt worden (met er tussenin nog een ':'):



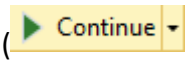
Klik je met de rechtermuisknop op deze geselecteerde expressie, kan je die ook naar het Watch venster overbrengen:



Ons lijkt deze waarde "23:010" helemaal een eyeopener dat er iets fout aan het lopen is in het programma.

F5 - Continue

Als je bij het debuggen van een groots project, vanaf je onderbrekingspunt, alle code instructie per instructie moet afwerken met de F10- of F11-toets, kan je soms wel eens heel lang bezig zijn.

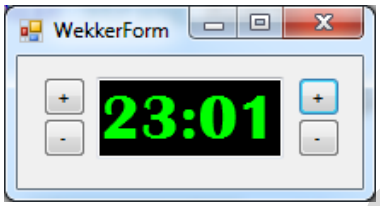
De knop **Continue** (

) in de werkbalk of de gelijkwaardige sneltoets **F5**, biedt in zo'n situatie vaak soelaas. Deze knop vertelt aan de debugger dat hij niet meer op onze commando's moet wachten en de rest van het programma mag afwerken.

Mocht er bij het verder verloop van het programma nog een onderbrekingspunt gepasseerd worden, zal de debugger daar wel opnieuw stoppen (en kan je weer stap per stap verder debuggen). Zijn er geen rode bollen waar het programma voorbij moet, zal verder gedaan worden tot het einde van de code.

- In onze test, stond de debugger nog vast op het einde van de methode `AlarmOm()`. Met Continue of F5 geven we de instructie om het programma verder te laten aflopen.

  Bij het uitvoeren van de code, wordt niet meer langs een onderbrekingspunt gepasseerd. Je zal merken dat Visual Studio daarom uit de Debug modus springt en dat je terug op je formulier terechtkomt.



Nu onze ruime debugsessie erop zit, snappen we het helemaal. In het tekstvak zien we wel "23:01" staan, maar eigenlijk is de inhoud er "23:010". Het laatste nulletje past er in het teksvak echter niet meer bij.

Het was dus niet onze wekker die een rekenfout maakte bij de som "9 minuten + 1 minuut", maar de logische fout in onze code gebeurde bij het samenstellen van de uu:mm-notatie.

We kunnen nu de code verbeteren. Typ er dus in de klasse `Wekker` bij de methode `DisplayTweeCijfers()` op regel 91 in de if-voorwaarde een '='-teken bij:

```
87      private String DisplayTweeCijfers(int getal)
88      {
89          String displayString;
90
91          if (getal >= 10)
92          {
93              displayString = getal.ToString();
```

En of af te sluiten nog een leuk weetje (of is het een broodje aap):

In de computerwereld wordt een fout vaak een *bug* genoemd. Betekent bug in het Engels niet iets als een insect?

De legende vertelt dat er ooit een mot ergens binnenin een computer sukkelde en daar voor technische problemen zorgde. De systeembeheerder maakte daar in zijn logboek melding van en plakte er de boosdoener bij (zie https://en.wikipedia.org/wiki/Software_bug#/media/File:H96566k.jpg) en noteerde "First actual case of bug being found."

En zo werd "de-*bug*-gen" voortaan de term om het proces aan te geven waar je fouten uit je programma's haalt. En de systeembeheerder (niet de mot) leefde nog lang en gelukkig …

Onthoud 2.9 Bij het debuggen van code zou je in Visual Studio volgende acties moeten kunnen uitvoeren:

- Door bij een bepaalde coderegel in de grijze linkermarge te klikken, zet je op die plaats in je code een **onderbrekingspunt** (**breakpoint**). Onderbrekingspunten worden er aangeven als een **rode bol**.

  Als bij het uitvoeren van de code aan een regel met zo'n onderbrekingspunt gepasseerd wordt, zal het programma op die plaats gepauzeerd worden. Het programma komt dan in de **Debug modus** terecht.

  De instructie waar het programma gepauzeerd staat, wordt geel gemarkeerd. In de grijze linkermarge staat op die plaats een **gele pijl**.

In de Debug modus kan je met volgende sneltoetsen de uitvoering van je code sturen:

- Met **F10** (**Step over**) laat je de geel gemarkeerde instructie uitvoeren.
- Als bij de geel gemarkeerde instructie een (hulp)methode opgeroepen wordt, zal je met **F11** (**Step Into**) de debugger binnen deze (hulp)methode brengen. Zo kan je ook deze (hulp)methode instructie per instructie doorlopen.
- Met **F5** (**Continue**) laat je de code (zonder onderbreking) verder uitvoeren. Het programma wordt dus gewoon doorlopen tot het einde, tenzij er een onderbrekingspunt gepasseerd wordt. Dan wordt de uitvoering op die plaats weer gepauzeerd.

In de Debug modus kan je op volgende manieren de actuele waarde van je variabelen, velden, objecten, … opvragen:

- Als je met de cursor op een variabele, veld, object, … gaat staan, krijg je de huidige waarde van dit element te zien. Als het om een object gaat, zal je na het openklappen van een menuutje de toestand van de velden van dat object kunnen bekijken.

- Door rechts op een variabele, veld, object, … te klikken en dan de optie **Add Watch** te selecteren, laat je dit element overbrengen naar het **Watch venster**. Zo blijft de waarde van dit element bij het debuggen steeds zichtbaar.

  Je kan ook een volledige expressie (die een waarde oplevert) naar het Watch venster overbrengen. Hiervoor selecteer je eerst deze expressie en dan kan je in het snelmenu ook de Add Watch optie nemen.

Oefening 2.6 We zouden jullie nu wat projecten kunnen voorschotelen waar wij enkele syntax- en logische fouten in verstopt hebben. Jullie moeten dan alle technieken die we in het hoofdstukje "2.8 Omdat missen menselijk is" besproken hebben, uit de kast halen om de fouten op te sporen en te verbeteren.

Er is echter niemand die *mooiere* programmeerfouten kan/zal maken dan *jijzelf*!

Laat ons dus afspreken dat deze "Oefening 2.6" een continue opdracht is, die pas eindigt als je de laatste bladzijde van de cursus omslaat. De opgave luidt:

- Bij build errors altijd de foutboodschap in het Error List venster nalezen. Vaak staat daar al duidelijk beschreven wat je fout gedaan hebt.
- Als een toepassing (onverklaarbaar) nog niet correct werkt, probeer dan eerst zelf eens om via de debugger naar de oorzaak te zoeken (vooraleer je vinger in de lucht gaat om de leerkracht om hulp te vragen).

3 Properties - part I

Nu we de overstap naar C# vlot gemaakt hebben en we ons de programmeerstijl die in de C#-community gehanteerd wordt, eigen gemaakt hebben, willen we het eens hebben over **Properties**.

Binnen de theorie van het objectgeoriënteerd programmeren nemen Properties bij sommige talen een belangrijke plaats in. In Java daarentegen heeft men de keuze gemaakt om Properties (nog?) niet te integreren. Vandaar dat dit in de cursus BlueJ ook niet aan bod kwam. Dat bewijst trouwens dat volwaardig objectgeoriënteerd programmeren zonder Properties natuurlijk wel kan.

In Visual Studio en C# zijn de Properties echter zo alomtegenwoordig dat we er in deze cursus twee volle hoofdstukken aan zullen besteden. In dit hoofdstuk leer je hoe je Properties kan implementeren in je *eigen businessklassen* en zie je op welke manier je diezelfde Properties moet aanspreken bij individuele objecten.

Alle besturingselementen (controls die we op formulieren kunnen slepen) beschikken over massa's Properties. We hebben er hiervan al enkele bekeken in het *Properties venster*. In hoofdstuk 4 komen we uitgebreid terug op dergelijke Properties.

In het Nederlands wordt Property vertaald als **eigenschap**. Wij zullen in de cursus beide benamingen (Property/eigenschap) door elkaar hanteren.

3.1 Properties in een klasse

Een belangrijke functie van Properties is dat ze de **accessoren en mutatoren** van een klasse kunnen **vervangen**. Waar een Java-programmeur een accessor en/of mutator gebruikte, zal een C#'er het steevast met een Property doen.

Je zal onmiddellijk merken dat een Property iets **compacter** is dan een accessor- en mutatormethode. Dit vooral omdat in één Property zowel de accessor als mutator verenigd worden.

Om onze uitleg te ondersteunen bekijken we *een laatste keer* een voorbeeldklasse met accessoren en mutatoren. Deze klasse vergelijken we dan met zijn gelijkwaardige variant waar de accessor- en mutatormethoden vervangen werden door de 'spiksplinternieuwe' Properties.

3.1.1 Een voorbeeldklasse met accessoren en mutatoren

We starten onze uitleg in het project `OldSchool`, meerbepaald in het bestand **Klas.cs** bij onze voorbeeldklasse `Klas`.

Het gaat om een eenvoudige klasse, waar we met enkele velden de gegevens bijhouden van een klas. Naast de accessoren/mutatoren om (enkele) velden op te vragen/in te stellen, hebben we GEEN extra functionaliteit (extra methoden) aan deze klasse toegevoegd.

```
public class Klas
{

    private String _afkorting;
    private int _jaar;
    private String _richting;
    private String _titularis;
    private int _aantalLeerlingen;

    public Klas(String afkorting, int jaar, String richting, String titularis,
                int aantalLeerlingen)
    {
        _afkorting = afkorting;
        _jaar = jaar;
        _richting = richting;
        _titularis = titularis;
        _aantalLeerlingen = aantalLeerlingen;
    }

    public String GetAfkorting()
    {
        return _afkorting;
    }

    public int GetJaar()
    {
        return _jaar;
    }

    public String GetRichting()
    {
        return _richting;
    }

    public String GetTitularis()
    {
        return _titularis;
    }

    public void SetTitularis(String titularis)
    {
        _titularis = titularis;
    }

    public int GetAantalLeerlingen()
    {
        return _aantalLeerlingen;
    }

    public void SetAantalLeerlingen(int aantalLeerlingen)
    {
        _aantalLeerlingen = aantalLeerlingen;
    }
}
```

Bij elke instantie (object) van de klasse `Klas`, zullen we dus vijf gegevens bijhouden. In de velden slaan we de afkorting (bijv. "5ST"), het jaar (bijv. 5), de richting (bijv. "Secretariaat Talen"), de titularis (bijv. "Dhr. Vandermeeren") en het aantal leerlingen (bijv. 11) op.

Bij de constructor laten we via parameters elk van deze vijf velden initialiseren.

Voor elk van de vijf velden is er een **accessor** voorzien. Enkel voor het veld `_titularis` en `_aantalLeerlingen` is er een **mutator** beschikbaar.

Dat betekent dus dat we aan een object van de klasse `Klas` de waarde van elk van de vijf velden kunnen **opvragen**, maar dat we *achteraf* enkel de waarde van het veld `_titularis` en `_aantalLeerlingen` kunnen **veranderen**.

Onze verantwoording om slechts die twee mutatoren te voorzien is dat in de loop van het schooljaar de afkorting, het jaar en de richting van een klas toch niet meer zal veranderen. Er kunnen in een klas natuurlijk plots wel leerlingen aansluiten (of afvallen). Als jouw klas-titularis het grote lot wint met de lotto of zijn/haar bitcoins ten gelde maakt - en stante pede op wereldreis vertrekt - zal de directeur een nieuwe klastitularis moeten aanstellen.

Het venster Immediate is een snelle manier om de klasse `Klas` aan het werk te zien. Ook onderstaande screenshot ziet er ondertussen vertrouwd uit:

```
Immediate Window
? Klas klas5st = new Klas("5ST", 5, "Secretariaat Talen", "Dhr. Vandermeeren", 11);
{OldSchool.Klas}
    _aantalLeerlingen: 11
    _afkorting: "5ST"
    _jaar: 5
    _richting: "Secretariaat Talen"
    _titularis: "Dhr. Vandermeeren"
? klas5st.GetTitularis()
"Dhr. Vandermeeren"
? klas5st.SetTitularis("Mevr. Desmet");
Expression has been evaluated and has no value
? klas5st.GetTitularis()
"Mevr. Desmet"
? klas5st
{OldSchool.Klas}
    _aantalLeerlingen: 11
    _afkorting: "5ST"
    _jaar: 5
    _richting: "Secretariaat Talen"
    _titularis: "Mevr. Desmet"
```

We verduidelijken graag nog eens apart de verschillende code-fragmenten.

```
Immediate Window
? Klas klas5st = new Klas("5ST", 5, "Secretariaat Talen", "Dhr. Vandermeeren", 11);
{OldSchool.Klas}
    _aantalLeerlingen: 11
    _afkorting: "5ST"
    _jaar: 5
    _richting: "Secretariaat Talen"
    _titularis: "Dhr. Vandermeeren"
```

We declareerden er eerst het object `klas5st` van het type `Klas`. Met de vijf parameters laten we het object `klas5st` onmiddellijk initialiseren. Het venster Immediate toont na een declaratie onmiddellijk de toestand van dit nieuwe object.

```
? klas5st.GetTitularis()
"Dhr. Vandermeeren"
? klas5st.SetTitularis("Mevr. Desmet");
Expression has been evaluated and has no value
? klas5st.GetTitularis()
"Mevr. Desmet"
```

Dat we met de accessor `GetTitularis()` de titularis kunnen opvragen en met de mutator `SetTitularis()` deze kunnen wijzigen, blijkt duidelijk uit bovenstaand fragment.

```
? klas5st
{OldSchool.Klas}
    _aantalLeerlingen: 11
    _afkorting: "5ST"
    _jaar: 5
    _richting: "Secretariaat Talen"
    _titularis: "Mevr. Desmet"
```

Als we tenslotte nog eens de toestand van het object `klas5st` opvragen, zie je dat de nieuwe titularis inderdaad opgeslagen werd in het veldje `_titularis`.

3.1.2 Een voorbeeldklasse met Properties

Om maar onmiddellijk met de deur in huis te vallen, ga in het project `NewSchool` ook op zoek naar de klasse `Klas` in het bestand **Klas.cs**.

Deze klasse heeft precies dezelfde functionaliteit als de gelijknamige voorbeeldklasse in het project `OldSchool`, maar de accessoren en mutatoren werden hier vervangen door gelijkwaardige Properties:

```
public class Klas
{
    private String _afkorting;
    private int _jaar;
    private String _richting;
    private String _titularis;
    private int _aantalLeerlingen;

    public Klas(String afkorting, int jaar, String richting, String titularis,
                int aantalLeerlingen)
    {
        _afkorting = afkorting;
        _jaar = jaar;
        _richting = richting;
        _titularis = titularis;
        _aantalLeerlingen = aantalLeerlingen;
    }

    public String Afkorting
    {
        get {return _afkorting;}
    }

    public int Jaar
    {
        get {return _jaar;}
    }

    public String Richting
    {
        get {return _richting;}
    }
```

```
    public String Titularis
    {
        get {return _titularis;}
        set {_titularis = value;}
    }

    public int AantalLeerlingen
    {
        get {return _aantalLeerlingen;}
        set {_aantalLeerlingen = value;}
    }
}
```

Bij de velden en de constructor hebben we niets nieuws te melden. We slaan nog steeds de afkorting, het jaar, de richting, de titularis en het aantal leerlingen van een klas op in vijf velden. Maak je via de constructor een nieuw object aan, zullen deze velden via vijf parameters geïnitialiseerd worden.

Wel helemaal nieuw zijn de Properties (eigenschappen). Het gaat hier concreet over de Properties `Afkorting`, `Jaar`, `Richting`, `Titularis` en `AantalLeerlingen`.

We nemen als eerste de Property `AantalLeerlingen` eens onder de loep.

De bedoeling van deze eigenschap `AantalLeerlingen` is om aan een `Klas`-object het aantal leerlingen te kunnen **opvragen** en om dit aantal leerlingen te kunnen **wijzigen**. Wij weten dat het aantal leerlingen bijgehouden wordt in het veld `_aantalLeerlingen`.

```
public int AantalLeerlingen
{
    get {return _aantalLeerlingen;}
    set {_aantalLeerlingen = value;}
}
```

Bij Properties is er dus ook sprake van een header:

```
public int AantalLeerlingen
{
}
```

- Na het sleutelwoord **public** plaatsen we het **type** van de Property. Hier `int` omdat deze eigenschap gekoppeld zal zijn aan het veld `_aantalLeerlingen` en dit veld is van het type `int`.
- Men heeft de gewoonte om een Property **dezelfde naam** te geven als het achterliggende veld, maar om hierbij een **PascalCase**-notatie te gebruiken en de underscore achterwege te laten. Onze eigenschap werkt voor het veld `_aantalLeerlingen`. Bijgevolg noemden we de Property `AantalLeerlingen`.
- Bij een Property zijn er **NOOIT parameters**. Vandaar dat men in de header van Properties **GEEN ronde haakjes** plaatst.

Bij de code van een Property kan een **get**- en een **set**-statement voorkomen.

```
get {return _aantalLeerlingen;}
```

- De **get**-regel vertelt dat als je aan een `Klas`-object de Property `AantalLeerlingen` **opvraagt**, dat dan het veld `_aantalLeerlingen` geretourneerd wordt.

```
set {_aantalLeerlingen = value;}
```

- De **set**-regel vertelt dat als je bij een `Klas`-object de eigenschap `AantalLeerlingen` **wijzigt**, de nieuwe waarde (symbolisch aangegeven door het sleutelwoord **value**) wordt weggeschreven in het veld `_aantalLeerlingen`.

Samenvattend kunnen we zeggen dat we via deze code onze Property `AantalLeerlingen` lieten koppelen aan het veld `_aantalLeerlingen`. Wat we straks allemaal doen met de eigenschap `AantalLeerlingen` bij een `Klas`-object zal in de achtergrond verwerkt worden in het veld `_aantalLeerlingen`.

Onthoud 3.1 **Properties** (eigenschappen) kunnen gebruikt worden om bij een object een bepaald veld op te vragen / te wijzigen. We hebben dan in feite een één op één relatie tussen de eigenschap en het veld.

Sommige Properties in de klasse `Klas` hebben wel een get-statement, terwijl het set-statement ontbreekt. Dit is zo voor de eigenschappen `Afkorting`, `Jaar` en `Richting`.

```
public String Afkorting
{
    get {return _afkorting;}
}
```

Via de Property `Afkorting` kan je dus wel de afkorting opvragen aan een `Klas`-object (je krijgt dan het veld `_afkorting` terug), maar is er geen mogelijkheid om deze afkorting te laten wijzigen.

Vergelijk dit met de situatie waar je voor een bepaald veld wel een accessor, maar geen mutator in de klasse plaatste. Zo'n eigenschap noemen we een **Read only Property**.

Onthoud 3.2 Een **Read only Property** is een eigenschap waarvoor je enkel een get-statement definieert en er zodoende geen set-statement voorkomt. Zo'n eigenschap kan je wel opvragen aan een object, maar de eigenschap kan je niet laten wijzigen.

Met de komst van Properties moesten we onze naming conventions wat uitbreiden. Onderstaande afspraak volgen wij in deze cursus strikt op!

> **Onthoud 3.3** We maken de afspraak om een Property (eigenschap) dezelfde naam te geven als het 'gekoppelde' veld, maar om hiervoor de **PascalCase**-stijl te gebruiken; de underscore laten we hierbij vallen.
>
> Bijvoorbeeld: de Property `AantalLeerlingen` en de Property `Titularis` voor de velden `_aantalLeerlingen` en `_titularis`.

In het grotere plaatje van objectgeoriënteerd programmeren:

- De velden blijven **verborgen** (private) bij objecten.
- Properties zijn de **zichtbare** (public) kenmerken van objecten.

  De programmeur houdt door wel/geen Properties te voorzien zelf in de hand welke informatie hij/zij publiek maakt bij de objecten van een klasse.

  Door wel/geen set-statement op te nemen bij een Property bepaalt de programmeur of een kenmerk van een object wel/niet gewijzigd kan worden.

3.2 Properties in het venster Immediate

Je weet ondertussen hoe je in een klassendefinitie een **Property** (eigenschap) moet opnemen en hoe je met een get- en set-statement de eigenschap aan een veld moet koppelen.

Dit is echter nog maar de helft van het verhaal. In deel twee van onze uitleg kijken we hoe we die Properties (eigenschappen) nu op onze objecten moeten toepassen.

Laten we ons hiervoor maar weer wenden tot het klassieke venster Immediate. We zetten hier een gelijkaardige test op als op pagina 91. Je kan veel leren door beide screenshots met elkaar te vergelijken.

```
Immediate Window
? Klas klas5ha = new Klas("5HA", 5, "Handel", "Mevr. De Bel", 19);
{NewSchool.Klas}
    _aantalLeerlingen: 19
    _afkorting: "5HA"
    _jaar: 5
    _richting: "Handel"
    _titularis: "Mevr. De Bel"
    AantalLeerlingen: 19
    Afkorting: "5HA"
    Jaar: 5
    Richting: "Handel"
    Titularis: "Mevr. De Bel"
? klas5ha.Titularis
"Mevr. De Bel"
? klas5ha.Titularis = "Mevr. Bostyn";
"Mevr. Bostyn"
? klas5ha.Titularis
"Mevr. Bostyn"
```

Vervolg:

```
? klas5ha
{NewSchool.Klas}
    _aantalLeerlingen: 19
    _afkorting: "5HA"
    _jaar: 5
    _richting: "Handel"
    _titularis: "Mevr. Bostyn"
    AantalLeerlingen: 19
    Afkorting: "5HA"
    Jaar: 5
    Richting: "Handel"
    Titularis: "Mevr. Bostyn"
```

We startten hier opnieuw met de declaratie en initialisatie van een `Klas`-object. Dit keer het object `klas5ha`:

```
Immediate Window
? Klas klas5ha = new Klas("5HA", 5, "Handel", "Mevr. De Bel", 19);
```

Het venster Immediate toont na deze declaratie onmiddellijk de toestand van dit nieuwe object `klas5ha`. Je krijgt nu niet enkel meer een overzicht van de velden, want ook alle Properties worden hierbij opgelijst:

```
{NewSchool.Klas}
    _aantalLeerlingen: 19
    _afkorting: "5HA"
    _jaar: 5
    _richting: "Handel"
    _titularis: "Mevr. De Bel"
    AantalLeerlingen: 19
    Afkorting: "5HA"
    Jaar: 5
    Richting: "Handel"
    Titularis: "Mevr. De Bel"
```

Omdat alle Properties één op één gekoppeld zijn aan een veld, krijg je uiteraard steeds dezelfde waarde te zien bij de Property en bij het overeenkomstige veld. Zo hebben bijvoorbeeld zowel het veld `_titularis` als de Property `Titularis` beiden de waarde "Mevr. De Bel".

Aan een object een bepaalde eigenschap opvragen, doe je als volgt:

```
? klas5ha.Titularis
"Mevr. De Bel"
```

We gebruiken dus de gekende punt-notatie (**object.Property**) om bij het object `klas5ha` de eigenschap `Titularis` aan te spreken!

Als we de eigenschap `Titularis` **opvragen**, wordt op de achtergrond het **get**-statement bij onze eigenschap `Titularis` geactiveerd (`"return _titularis;"`). Resultaat: het veld `_titularis` wordt geretourneerd.

Heb je het verschilletje gemerkt met de accessor-manier (zie pagina 91)? Bij een Property worden er **GEEN ronde haakjes** meer vermeld:

- met accessor: `klas5st.GetTitularis()`
- met Property: `klas5ha.Titularis`

Onthoud 3.4 Met de expressie **object.Property** vraag je aan het object de waarde op van de **Property**.

Die Property `Titularis` gebruiken we niet enkel om de titularis van een klas op te vragen, maar we kunnen via deze eigenschap ook de titularis van een klas **wijzigen**:

```
? klas5ha.Titularis = "Mevr. Bostyn";
```

Je weet nog dat je het symbool '**=**' (de toekenningsoperator) moet interpreteren als: "**wordt**" of "**krijgt de waarde**". Met dit trucje blijkt bovenstaande instructie in het venster Immediate intuïtief heel gemakkelijk te begrijpen. Je kan deze programmaregel immers lezen als:

De *titularis* van *klas5ha* ***wordt/krijgt de waarde*** *"Mevr. Bostyn".*

We gebruiken in deze context dus ook de gekende punt-notatie (**object.Property**). Echter door deze expressie nu **links van het '='-teken** te zetten, vragen we niet meer de waarde op van de Property, maar laten we de Property **wijzigen**.

Omdat we in deze context de eigenschap `Titularis` **wijzigen**, wordt het **set**-statement bij onze eigenschap `Titularis` geactiveerd (`"_titularis = value;"`). Resultaat: het veld `_titularis` krijgt een nieuwe waarde.

Deze instructie (met een Property) ziet er nu toch merkelijk anders uit dan zoals we het deden met een mutator (zie pagina 91).

- met mutator: `klas5st.SetTitularis("Mevr. Desmet");`
- met Property: `klas5ha.Titularis = "Mevr. Bostyn";`

Onthoud 3.5 Met de instructie "**`object.Property = ...;`**" geef je de **Property** van het object een nieuwe waarde.

Een **Read only Property** kan je wel opvragen, maar niet wijzigen. Ook dit kunnen we perfect demonstreren in het venster Immediate. De eigenschap `Afkorting` is een voorbeeld van zo'n read only eigenschap bij de klasse `Klas`.

We werken verder met ons object `klas5ha`:

```
Immediate Window
? klas5ha.Afkorting
"5HA"
? klas5ha.Afkorting = "5HAN";
Property or indexer 'Properties.Klas.Afkorting' cannot be assigned to -- it is read only
```

De afkorting opvragen (met de expressie `klas5ha.Afkorting`) is geen probleem. We krijgen in het venster Immediate mooi het antwoord "5HA" gepresenteerd.

Als we echter proberen om de afkorting te wijzigen (met `klas5ha.Afkorting = "5HAN";`), wordt er wel hard geprotesteerd. We lezen in de foutboodschap bijna letterlijk dat dit niet lukt wegens "een Property die read only is".

Opmerking:

- In theorie is het zelfs mogelijk dat je een Property wel een set-, maar geen get-statement geeft. Je krijgt dan het vreemde effect dat je die Property NIET kan opvragen, maar dat het wel lukt om deze te wijzigen bij een object van die klasse.

Oefening 3.1 We willen in een nieuwe businessklasse `Pacman` eens testen of je 'op papier' de juiste syntax kent om Properties te definiëren.

De header, de declaratie van de velden en de constructor van een klasse `Pacman` krijg je hieronder al onmiddellijk gegeven.

Wij vragen jullie om aan deze klassendefinitie de **Properties** toe te voegen. De commentaarregels vertellen telkens welke Property we precies nodig hebben:

```
public class Pacman
{

    private String _naam;          // de naam voor het happertje
    private int _aantalLevens;     // over hoeveel leventjes beschikt het happertje
                                   //   nog
    private bool _isOnkwetsbaar;   // true betekent dat het happertje onkwetsbaar is
                                   //   en bijgevolg spoken kan eten

    public Pacman(String naam, int aantalLevens)
    {
        _naam = naam;
        _aantalLevens = aantalLevens;
        _isOnkwetsbaar = false;
    }

    // definieer hier de Property Naam waarmee je het veld _naam kan opvragen en
    //   schrijven
```

```
    // definieer hier de Read only Property AantalLevens voor het veld
    //   _aantalLevens




    // definieer een Read only Property voor het veld _isOnkwetsbaar




}
```

Oefening 3.2 Zoals uit onderstaande test in het venster Immediate blijkt, beschikt de klasse `Smartphone` over de Properties `Telnummer` en `Pincode`.

Immediate Window

```
? Smartphone myPhone = new Smartphone("0494 94 94 49", "1234");
? myPhone.Telnummer = "0474 74 74 44"
Property or indexer 'Smartphone.Smartphone.Telnummer' cannot be assigned to -- it is read only
? myPhone.Telnummer
"0494 94 94 49"
? myPhone.Pincode = "9876";
? myPhone.Pincode
"9876"
```

Merk op dat de instructie `? myPhone.Telnummer = "0474 74 74 44"` hierboven een foutboodschap opleverde.

Kunnen jullie op basis van deze informatie de klassendefinitie van de klasse `Smartphone` aanvullen met deze Properties.

```
class Smartphone
{

    private String _telnummer;
    private String _pincode;

    public Smartphone(String telnummer, String pincode)
    {
        _telnummer = telnummer;
        _pincode = pincode;
    }

    // definieer hier de Property Telnummer
```

```
    // definieer hier de Property Pincode




}
```

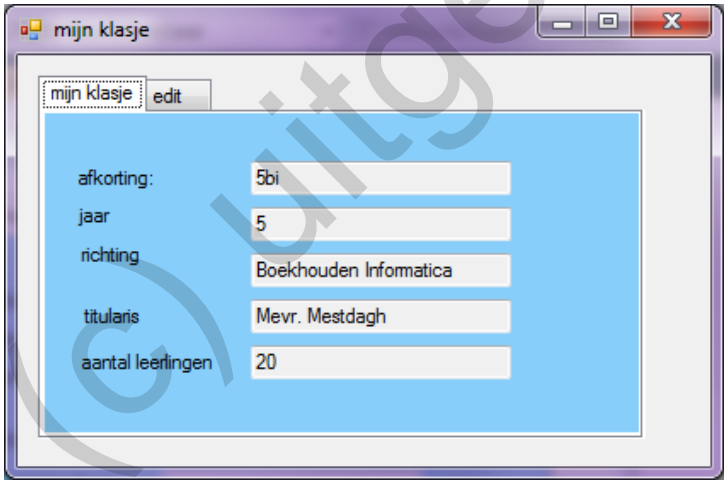
3.3 Properties in de Presentation Layer

Het venster Immediate is handig om een snelle test op te zetten voor een klasse, maar in deze cursus draait het sinds hoofdstuk 2 rond **formulieren**! We moeten dus zeker eens nagaan hoe we onze Properties (die we implementeerden in een klasse in de Business Layer) in een formulier (of dus in de Presentation Layer) moeten integreren.

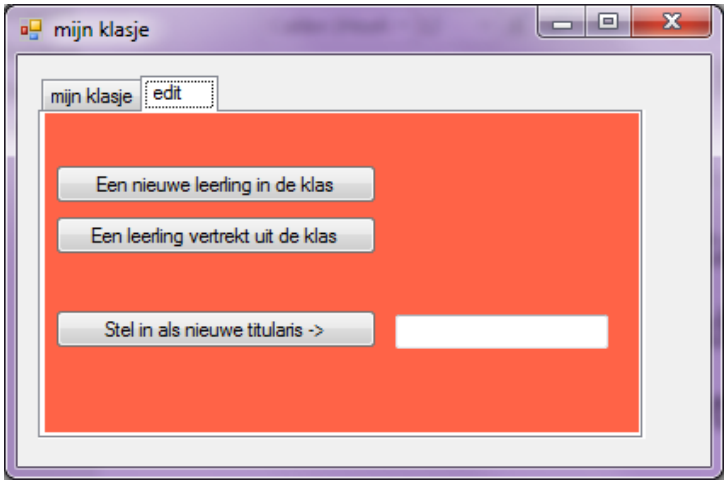
We stellen jullie hiervoor in het project `NewSchool` het formulier `KlasForm` voor.

Gewoon omdat het kan, hebben we de tekstvakken en knopjes in dit formulier verdeeld over twee verschillende tabbladen. We gebruikten hiervoor een **tabbesturingselement**. In de Toolbox vind je die trouwens terug onder de naam `TabControl` bij de categorie **Containers**.

- Bij het starten van het project krijg je in het formulier een tabblad te zien met de gegevens van een klas:



❖ Via het tabblad 'edit' kunnen de klasgegevens aangepast worden. We hebben er de mogelijkheid om het aantal leerlingen te verhogen of te verlagen en er kan een nieuwe titularis toegekend worden:



Speel zeker eens dit formuliertje af en vergewis je dat de wijzigingen die je in het 'edit'-tabblad maakt inderdaad in het 'mijn klasje'-tabblad overgenomen worden. Het formulier al eens live aan het werk gezien hebben, zal je helpen om hieronder de achterliggende programmacode beter te snappen!

Oefening 3.3 In het formulier `KlasForm` zal het je niet lukken om in het tabblad 'mijn klasje' de inhoud van de tekstvakken manueel aan te passen. Die tekstvakken staan dus op één of andere manier vergrendeld.

Vind je welke instelling we hiervoor in het Properties venster bij deze tekstvakken gemaakt hebben?

We tonen hieronder de code achter ons formulier of m.a.w. de code van de gelijknamige klasse `KlasForm`.

```
public partial class KlasForm : Form
{

    private Klas _mijnKlas;  // veld om klasgegevens op te slaan

    public KlasForm()
    {
        InitializeComponent();

        // het veld _mijnKlas initialiseren
        _mijnKlas = new Klas("5bi", 5, "Boekhouden Informatica",
                             "Mevr. Mestdagh", 20);

        // methode om klasgegevens in de tekstvakken te tonen
        ToonKlasgegevens();
    }
```

```
    private void aantalLeerlingPlusButton_Click(object sender, EventArgs e)
    {
        // aantal leerlingen in het _mijnKlas-veld met eentje verhogen
        _mijnKlas.AantalLeerlingen = _mijnKlas.AantalLeerlingen + 1;

        // methode om klasgegevens in tekstvakken te vernieuwen
        ToonKlasgegevens();
    }

    private void aantalLeerlingMinButton_Click(object sender, EventArgs e)
    {
        // aantal leerlingen in het _mijnKlas-veld met eentje verminderen
        _mijnKlas.AantalLeerlingen = _mijnKlas.AantalLeerlingen - 1;

        // methode om klasgegevens in tekstvakken te vernieuwen
        ToonKlasgegevens();
    }

    private void wijzigTitularisButton_Click(object sender, EventArgs e)
    {
        // titularis in _mijnKlas-veld bijwerken op basis van de inhoud
        //   van het tekstvak nieuweTitularisTextBox
        _mijnKlas.Titularis = nieuweTitularisTextBox.Text;

        // tekstvak nieuweTitularisTextBox weer leeg maken
        nieuweTitularisTextBox.Text = "";

        // methode om klasgegevens in tekstvakken te vernieuwen
        ToonKlasgegevens();
    }

    private void ToonKlasgegevens()
    {
        // laat de gegevens van de klas weergeven in de tekstvakken
        afkortingTextBox.Text = _mijnKlas.Afkorting;
        jaarTextBox.Text = _mijnKlas.Jaar.ToString();
        richtingTextBox.Text = _mijnKlas.Richting;
        titularisTextBox.Text = _mijnKlas.Titularis;
        aantalLeerlingenTextBox.Text = _mijnKlas.AantalLeerlingen.ToString();
    }
}
```

Alle code in dit formulier is opgebouwd rond het veld `_mijnKlas`.

Het veld `_mijnKlas` houdt immers alle gegevens bij van de klas die we in het formulier weergeven. `_mijnKlas` is uiteraard een object van het type `Klas` en doet dus dienst als businessobject in het formulier. Via dit object `_mijnKlas` kunnen we binnenin de Presentation Layer beroep doen op de functionaliteit die we in de Business Layer uitgewerkt hebben.

In de constructor van `KlasForm` initialiseren we ons veld `_mijnKlas`. Pas er gerust de parameters aan, zodat het echt 'jouw klasje' wordt.

```
_mijnKlas = new Klas("5bi", 5, "Boekhouden Informatica", "Mevr. Mestdagh", 20);
```

Hieronder concentreren we ons op alle facetten van het gebruik van de Properties van dit object `_mijnKlas`.

3.3.1 Een Property opvragen

De hulpmethode `void ToonKlasgegevens()` is er om de info van onze klas (opgeslagen in het object `_mijnKlas`) weer te geven in de vijf tekstvakken op het formulier:

```
private void ToonKlasgegevens()
{
    // laat de gegevens van de klas weergeven in de tekstvakken
    afkortingTextBox.Text = _mijnKlas.Afkorting;
    jaarTextBox.Text = _mijnKlas.Jaar.ToString();
    richtingTextBox.Text = _mijnKlas.Richting;
    titularisTextBox.Text = _mijnKlas.Titularis;
    aantalLeerlingenTextBox.Text = _mijnKlas.AantalLeerlingen.ToString();
}
```

Deze code is een perfecte illustratie van hoe je via eigenschappen gegevens kan **opvragen** aan een businessobject.

Eén van de instructies hierbij is:

```
afkortingTextBox.Text = _mijnKlas.Afkorting;
```

Met de Property `Afkorting` vraag je aan het `_mijnKlas`-object op wat de afkorting is van deze klas.

Je schrijft deze bekomen afkorting weg in het tekstvak `afkortingTextBox`, want zoals je ongetwijfeld nog weet, verwijs je met de expressie `afkortingTextBox.Text` naar **de inhoud** van het tekstvak.

In de methode `ToonKlasgegevens()` botsten we ook op een conversieprobleempje:

```
jaarTextBox.Text = _mijnKlas.Jaar.ToString();
```

Eventjes het probleem analyseren:

- De expressie `_mijnKlas.Jaar` levert het jaar van de klas op, of dus een getalletje (een `int`). In de klasse `Klas` hebben wij inderdaad bij de header van de Property `Jaar` deze eigenschap op het type `int` ingesteld:

```
public int Jaar
{
    get {return _jaar;}
}
```

- De inhoud van een tekstvak daarentegen, dat is altijd van het type `String`.

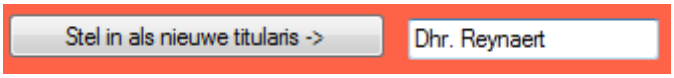
Visual Studio laat door het verschil bij deze gegevenstypen dan ook niet zomaar toe dat we het *getalletje* in het tekstvak zetten.

Als je met de methode `ToString()` echter eerst het getal omzet naar een tekst, voert Visual Studio, zo mak als een lammetje, de instructie wel uit.

3.3.2 Een Property wijzigen

Als de methode `ToonKlasGegevens()` dient om de gegevens van de klas te **tonen**, dan kunnen we met de methode `wijzigTitularisButton_Click()` de titularis van een klas **wijzigen**.

Door de naam `wijzigTitularisButton_Click()` weten jullie trouwens al dat deze methode opgestart zal worden als op het knopje `wijzigTitularisButton` geklikt wordt. De naam van de nieuwe titularis tik je hiervoor eerst in het bijhorende tekstvak in:



De code:

```
private void wijzigTitularisButton_Click(object sender, EventArgs e)
{
    // titularis in _mijnKlas-veld bijwerken op basis van de inhoud
    //   van het tekstvak nieuweTitularisTextBox
    _mijnKlas.Titularis = nieuweTitularisTextBox.Text;

    // methode om klasgegevens in tekstvakken te vernieuwen
    ToonKlasgegevens();

    // tekstvak nieuweTitularisTextBox weer leeg maken
    nieuweTitularisTextBox.Text = "";
}
```

En zo hebben we meteen een voorbeeldinstructie te pakken hoe je **een Property wijzigt**:

```
_mijnKlas.Titularis = nieuweTitularisTextBox.Text;
```

- Doordat we de Property `Titularis` van het `_mijnKlas`-object nu links van een "="-teken plaatsen, betekent dit dat we deze eigenschap een waarde gaan geven.
- De expressie rechts van het "="-teken kennen jullie ondertussen. Met de expressie `nieuweTitularisTextBox.Text` verwijzen we naar de inhoud van dat tekstvak.

Of dus in een zinnetje: "*De titularis van de klas **wordt/krijgt de waarde van** de inhoud van het tekstvak*". Of net iets vlotter geformuleerd: "*De nieuwe titularis wordt diegene die ingevuld staat in het tekstvak*".

We hebben daarmee ons businessobject `_mijnKlas` dan wel bijgewerkt, maar het tekstvak in het 'mijn klasje'-tabblad, waar we de titularis van de klas tonen, weet dit nog niet. We moeten het tekstvak daar nog updaten (anders blijven we er de 'oude' titularis zien).

Dit doen we het gemakkelijkst door de hulpmethode `ToonKlasGegevens` nog eens op te roepen.

```
// methode om klasgegevens in tekstvakken te vernieuwen
ToonKlasgegevens();
```

Opmerking:

- Akkoord, de methode `ToonKlasGegevens` zal alle vijf tekstvakken uit het eerste tabblad aanpassen, terwijl in deze situatie eigenlijk enkel de titularis moest bijgewerkt worden. Hier kiezen we echter voor ons eigen programmeergemak en roepen gewoon de hulpmethode `ToonKlasGegegevens()` op.

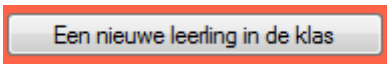
Het tekstvak `nieuweTitularisTextBox` waarin we de nieuwe titularis ingetikt hadden, laten we weer leeg maken:

```
// tekstvak nieuweTitularisTextBox weer leeg maken
nieuweTitularisTextBox.Text = "";
```

3.3.3 Een Property opvragen én schrijven in één en dezelfde instructie

En tenslotte nog eens kijken naar de methoden om het aantal leerlingen te verhogen/verlagen, want daar doet het fenomeen zich voor dat we in één enkele instructie eenzelfde Property zowel laten lezen als schrijven.

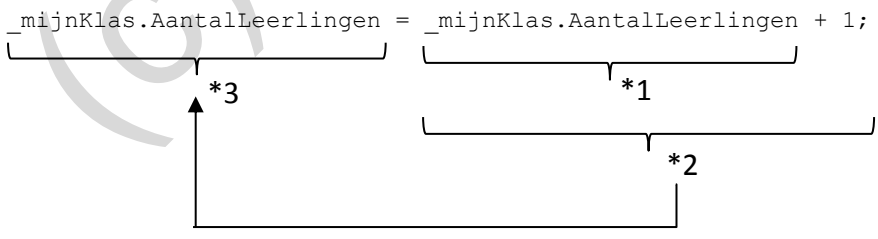
Via het knopje 'een nieuwe leerling in de klas' moeten we het aantal leerlingen in onze klas met één laten verhogen.



De methode die getriggerd wordt bij het klikken op deze knop is:

```
private void aantalLeerlingPlusButton_Click(object sender, EventArgs e)
{
    // aantal leerlingen in het _mijnKlas-veld met eentje verhogen
    _mijnKlas.AantalLeerlingen = _mijnKlas.AantalLeerlingen + 1;

    // methode om klasgegevens in tekstvakken te vernieuwen
    ToonKlasgegevens();
}
```

Om onderstaande cruciale instructie te ontleden, geven we met behulp van accolades aan in welke volgorde welk onderdeeltje van deze instructie geëvalueerd wordt:



*1: We **vragen** aan het object `_mijnKlas` het aantal leerlingen **op**.

*2: We tellen er ééntje bij.

*3: We **wijzigen** het aantal leerlingen van het object `_mijnKlas` naar de nieuwe berekende waarde.

M.a.w. omdat de Property `AantalLeerlingen` eens rechts en eens links van het "="-teken voorkomt, laten we in deze instructie deze eigenschap zowel opvragen als wijzigen.

Uiteraard zijn onderstaande kortere varianten ook bij Properties een geldig alternatief:

```
_mijnKlas.AantalLeerlingen += 1;
_mijnKlas.AantalLeerlingen++;
```

Ook in deze methode moesten we zorgen dat nadat we het aantal leerlingen van het object `_mijnKlas` gewijzigd hadden, we het betreffende tekstvakje in het tabblad 'mijn klasje' lieten bijwerken.

Vandaar dat we nog eens onze methode `ToonKlasgegevens()` oproepen:

```
// methode om klasgegevens in tekstvakken te vernieuwen
ToonKlasgegevens();
```

3.4 Een ander voorbeeldje

Omdat ouders het soms wel eens beu raken om 's avonds en in het weekend hun kroost het land rond te moeten rijden voor allerhande sport-, muziek-, jeugdbeweging-, CoderDojo- en schoolactiviteiten, wil de school een *kleine* duit in het zakje doen. Als de leerlingen een groepswerk willen voorbereiden, dan mag dat voortaan 's avonds op school. Dit zou voor de ouders toch al enkele taxibeurten kunnen schelen.

De school zal hiervoor tijdens de avondstudie in een apart lokaal toezicht voorzien. De afspraak is wel dat leerlingen *in de loop van de dag* op het leerlingensecretariaat hun plaatsje gaan reserveren. Het leerlingensecretariaat behoudt zo het overzicht en zorgt natuurlijk dat ze niet méér leerlingen toelaten dan er stoelen in het lokaal staan.

Op vraag van het leerlingensecretariaat willen wij wel een *programmaatje* maken om hen een beetje te helpen met hun administratie.

3.4.1 De Business Layer

De essentie van de toepassing die we willen uitwerken, is dat het leerlingensecretariaat gedurende de dag moet kunnen bijhouden hoeveel leerlingen er inschrijven voor de vrije studie die na de lessen in een gegeven lokaal zal plaatsvinden. Ze willen hiermee vooral voorkomen dat er 's avonds te veel leerlingen in het lokaal aanschuiven! Omdat de vrije studie voor groepswerk gebruikt mag worden, kunnen de inschrijvingen ook in groep gebeuren.

Deze probleemstelling willen we in een businessklasse `VrijeStudie` vatten. Denken jullie even mee …

Oefening 3.4 Doen jullie een voorstel van welke **velden** (met welk gegevenstype) je in de klasse `VrijeStudie` zou opnemen.

Oefening 3.5 In de klasse `VrijeStudie` zal je een **constructor** moeten definiëren, waar je de velden zal initialiseren. Welke van je velden laat je hierbij instellen via een parameter; welke velden geef je een vaste startwaarde?

Oefening 3.6 In dit hoofdstuk Properties, laten we jullie natuurlijk ook eens noteren welke **Properties** je in de klasse `VrijeStudie` zou voorzien. Welke van deze Properties maak je **Read only** en waarom?

Oefening 3.7 Doe ook een voorstel welke extra **methoden** je nog aan de klasse `VrijeStudie` zou toevoegen.

Wij hebben uiteraard zelf eens de denkoefening gemaakt hoe we de klasse `VrijeStudie` zouden kunnen invullen. Hieronder vind je ons voorstel.

In het bronproject `LeerlingenSecretariaat` (zie **_hoofdstuk_2/_voorbeelden**) zou je dus deze versie van de klasse `VrijeStudie` kunnen invoeren (in **VrijeStudie.cs**).

```
public class VrijeStudie
{
    private String _lokaal;  // lokaal waar vrije studie doorgaat
    private int _capaciteit;  // hoeveel leerlingen kunnen in het lokaal
    private int _gereserveerdePlaatsen; // hoeveel plaatsen werden al gereserveerd

    public VrijeStudie(String lokaal, int capaciteit)
    {
        _lokaal = lokaal;
        _capaciteit = capaciteit;
        _gereserveerdePlaatsen = 0;
    }

    public String Lokaal
    {
        get { return _lokaal; }
    }

    public int Capaciteit
    {
        get { return _capaciteit; }
    }

    public int GereserveerdePlaatsen
    {
        get { return _gereserveerdePlaatsen; }
        set { _gereserveerdePlaatsen = value; }
    }

    public bool IsErNogPlaats(int aantal)
    {
        bool isErNogPlaats = false;

        if (_gereserveerdePlaatsen + aantal <= _capaciteit)
        {
            isErNogPlaats = true;
        }

        return isErNogPlaats;
    }
}
```


Opmerking:

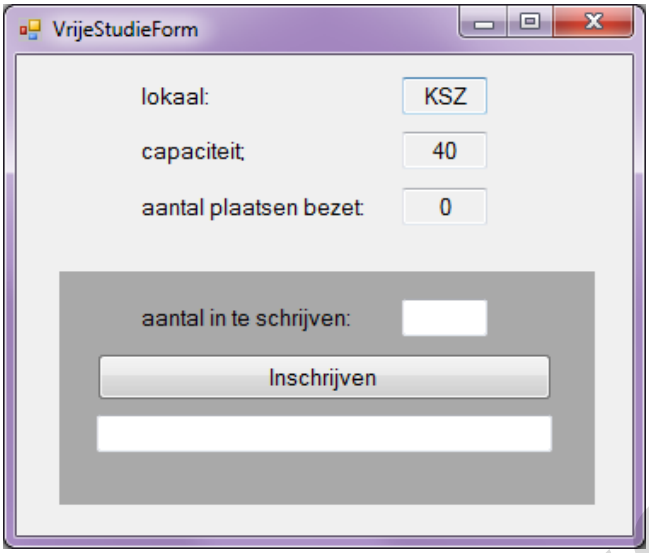
- Een methode `void Inschrijven(int)` waarmee we een gegeven aantal leerlingen (=parameter) laten inschrijven, stond ook heel lang op onze shortlist.

  Omdat we – zonder deze methode - straks in de Presentation Layer iets meer beroep moeten doen op de Properties (en dit hoofdstuk nu eenmaal over deze Properties handelt), hebben we deze methode `Inschrijven()` *om pedagogische redenen*, niet aan de klasse `VrijeStudie` toegevoegd.

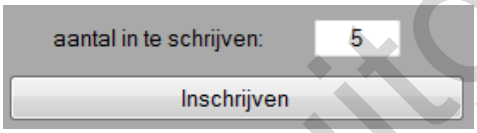
3.4.2 De Presentation Layer

De businessklasse `VrijeStudie` is een belangrijke stap in de ontwikkeling van onze toepassing, maar zolang we er geen GUI bijdoen, kan het leerlingensecretariaat er niet veel mee aanvangen!

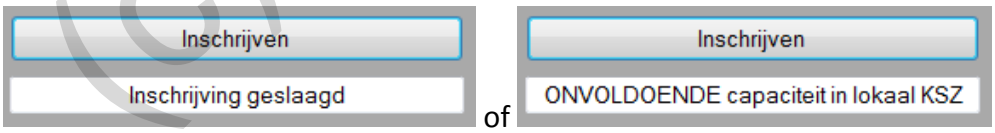
Volgend formuliertje `VrijeStudieForm` daarentegen spreekt zo voor zich dat ze in het leerlingensecretariaat, bijna zonder één woord extra uitleg, onmiddellijk aan de slag kunnen:



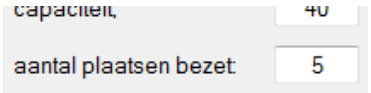
Als een leerling / een groepje leerlingen zich aan het leerlingensecretariaat aanmeldt, zal men in het vakje 'aantal in te schrijven:' invullen hoeveel leerlingen er willen inschrijven voor de vrije studie:



Klikken op het knopje 'Inschrijven' zal controleren of het opgegeven aantal leerlingen (hier 5) er die avond in de vrije studie nog bij kan. Er zijn dan twee uitkomsten mogelijk, die in het tekstvak helemaal onderaan respectievelijk tot volgende boodschap leiden:



Uiteraard wordt het vak 'aantal plaatsen bezet:' in het formulier altijd up-to-date gehouden:



In onderstaande oefeningen vragen we weer NIET om programma-instructies uit te schrijven, maar om in woorden, of als dit je beter ligt met een schema … uit te leggen welke acties nodig zijn om het formulier `VrijeStudieForm` correct aan te sturen.

Maar eerst nog eens de situatie overschouwen:

Het formulier `VrijeStudieForm` is de Presentation Layer die we bovenop de business-klasse `VrijeStudie` zullen implementeren. Daarom zullen we in de formulierklasse `VrijeStudieForm` een businessobject van de klasse `VrijeStudie` nodig hebben.

In `VrijeStudieForm` declareren we hiervoor een veld van de klasse `VrijeStudie`. Laat ons ervan uitgaan dat we dit veld `_vrijeStudie` zullen noemen.

Van het leerlingensecretariaat kregen we trouwens nog door dat de vrije studie altijd zal plaatsvinden in het lokaal "KSZ" en dat er daar tot 40 leerlingen mogen gaan zitten.

Oefening 3.8 Welke acties laat je in de constructor van de formulierklasse `VrijeStudie-Form` uitvoeren? Houd er rekening mee dat je in deze constructor bepaalt wat er te zien zal zijn op het moment dat je het formulier opstart.

Oefening 3.9 Wanneer in het formulier `VrijeStudieForm` op het knopje 'Inschrijven' geklikt wordt, zal er één en ander uitgevoerd moeten worden. We zullen dus nood hebben aan een methode die reageert op de gebeurtenis `Click` van deze knop.

Kunnen jullie, op basis van de beschrijving van de werking van het formulier hierboven, uittekenen welke zaken er in deze methode zullen moeten gebeuren.

Hieronder een voorstel hoe wij in het project `LeerlingenSecretariaat` (zie **_hoofdstuk_2/_voorbeelden**) de klasse `VrijeStudieForm` zouden uitwerken.

```
public partial class VrijeStudieForm : Form
{
    private VrijeStudie _vrijeStudie;

    public VrijeStudieForm()
    {
        InitializeComponent();

        // veld _vrijeStudie initialiseren (vrije studie gaat door in lokaal KSZ,
        //   er zijn daar 40 plaatsen)
        _vrijeStudie = new VrijeStudie("KSZ", 40);

        // nodige informatie aan opvragen_vrijeStudie en weergeven in tekstvakken
        lokaalTextBox.Text = _vrijeStudie.Lokaal;
        capaciteitTextBox.Text = _vrijeStudie.Capaciteit.ToString();
        bezetTextBox.Text = _vrijeStudie.GereserveerdePlaatsen.ToString();
    }

    private void inschrijvenButton_Click(object sender, EventArgs e)
    {
        // nagaan hoeveel leerlingen willen inschrijven
        int aantal = Convert.ToInt32(aantalInschrijvenTextBox.Text);

        // object _vrijeStudie controleert of er nog plaats is voor deze leerlingen
        if (_vrijeStudie.IsErNogPlaats(aantal) == true)
        {
            // aantal bezette plaatsen bij _vrijeStudie bijwerken
            _vrijeStudie.GereserveerdePlaatsen += aantal;

            // nodige tekstvakken updaten
            boodschapTextBox.Text = "Inschrijving geslaagd";
            bezetTextBox.Text = _vrijeStudie.GereserveerdePlaatsen.ToString();
        }
        else
        {
            // boodschap tonen in tekstvak
            boodschapTextBox.Text = "ONVOLDOENDE capaciteit in lokaal " +
                                    _vrijeStudie.Lokaal;
        }

        // tekstvak met aantal in te schrijven leerlingen leegmaken
        aantalInschrijvenTextBox.Text = "";
    }
}
```

Overal waar we in bovenstaande code gebruik maakten van een Property bij het object `_vrijeStudie` hebben we dit in het vet geplaatst.

Omdat Properties vanaf nu een vast ingrediënt zullen vormen in ons kookboek (een beetje zoals het klontje boter bij Jeroen Meus), zal je steeds maar weer met dergelijke instructies geconfronteerd worden. Een greep uit dit formulier:

- Instructies waar we via een eigenschap een bepaald kenmerk aan een object opvragen, om dit dan in een tekstvak te printen:

```
lokaalTextBox.Text = _vrijeStudie.Lokaal;
```

- Soms zal er hierbij een bijkomende conversie nodig zijn:

```
capaciteitTextBox.Text = _vrijeStudie.Capaciteit.ToString();
```

- Of we verwerken de eigenschap ergens in één of andere berekening (hier worden teksten aan elkaar geplakt):

```
boodschapTextBox.Text = "ONVOLDOENDE capaciteit in lokaal " +
                        _vrijeStudie.Lokaal;
```

- Via eigenschappen zullen we ook de toestand van objecten laten wijzigen:

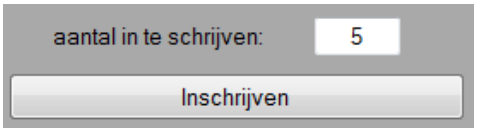
```
_vrijeStudie.GereserveerdePlaatsen += aantal;
```

Conversie van tekst naar getal

Als je – tussen alle Properties door – de code bij de klasse `VrijeStudieForm` aandachtig bestudeerd hebt, dan zal je waarschijnlijk even de wenkbrauwen gefronst hebben bij volgende bijzondere instructie:

```
int aantal = Convert.ToInt32(aantalInschrijvenTextBox.Text);
```

Deze instructie wordt gebruikt in de methode bij het klikken op het knopje `inschrijven-Button`. Nog even situeren:



In deze situatie moeten we, bij het klikken op de knop 'Inschrijven', nagaan of er nog *5* plaatsen vrij zijn in de vrije studie. Het is dus noodzakelijk dat we in onze code uitlezen welk aantal de gebruiker opgegeven heeft in dit tekstvak.

Visual Studio geeft ons echter een rode kaart als we op de volgende manier de inhoud van het tekstvak `aantalInschrijvenTextBox` in de `int`-variabele `aantal` proberen te zetten:

```
int aantal = aantalInschrijvenTextBox.Text;
```

Waarom lukt dit niet:

- Rechts van het '='-teken duiden we met de expressie `aantalInschrijvenText-Box.Text` **de inhoud** van het tekstvak aan. Wat in een tekstvak staat, wordt altijd geïnterpreteerd als van het type `String`.
- Links van het '='-teken hebben we een `int`-variabele.

En voor Visual Studio is het heel simpel: een `String` (een tekst) past NIET in een `int`-variabele!

Om dit probleem op te heffen zouden we de `String` met de inhoud van het tekstvak, expliciet moeten laten converteren naar een `int`. De `ToString()` die we vroeger gebruikten, biedt natuurlijk geen soelaas, want die doet net het omgekeerde (van getal naar tekst).

De expressie waarmee je van `String` naar `int` overgaat, ziet er zo uit:

`Convert.ToInt32(`*`String`*`).`

Tussen de haakjes geef je de `String` op die je wilt omzetten naar een `int`.

Omdat wij de inhoud van het tekstvak `aantalInschrijvingenTextBox` naar een `int` moeten krijgen, wordt de instructie dus:

```
int aantal = Convert.ToInt32(aantalInschrijvenTextBox.Text);
```

Daar er nu zowel links als rechts van het '='-teken een `int`-staat, krijgen we hier wel groen licht!

Onthoud 3.6 Conversie van `String` naar `int`.

Met de expressie `Convert.ToInt32(`*`String`*`)` laat je de `String`-parameter omzetten naar het type `int`.

Bijvoorbeeld:

```
String getalAlsTekst = "200";
int getal;

getal = Convert.ToInt32(getalAlsTekst);
```

Resultaat: de variabele `getal` zal de waarde 200 bevatten.

Een ander numeriek gegevenstype is `double`. In dezelfde categorie geven we dan ook maar onmiddellijk de conversietechniek om een `String` om te zetten naar `double`:

Onthoud 3.7 Conversie van `String` naar `double`.

Met de methode `Convert.ToDouble(`*`String`*`)` laat je de `String`-parameter omzetten naar het type `double`.

Bijvoorbeeld:

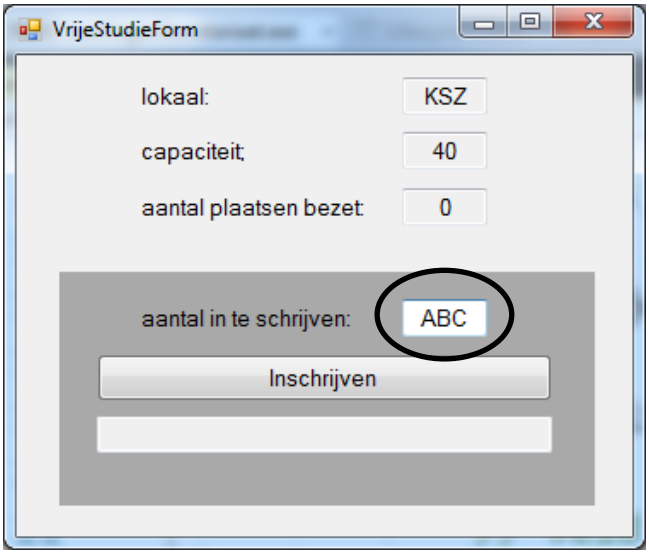
```
String bedragAlsTekst = "10,50";
double bedrag;

bedrag = Convert.ToDouble(bedragAlsTekst);
```

Resultaat: de variabele `bedrag` zal de numerieke waarde 10,5 bevatten.

We zitten nog met een laatste bedenking over onze LeerlingenSecretariaat-toepassing.

Uiteindelijk zijn wij (en de personeelsleden op het leerlingensecretariaat) tevreden over het formulier `VrijeStudieForm`. Het registreren van de vrije plaatsen in de avondstudie lijkt probleemloos te verlopen. Toch kan je met een beetje slechte wil, het formulier in een oogwenk laten crashen.

Als je in het tekstvak met het aantal leerlingen dat een plaatsje wil reserveren, iets anders dan een getal intikt en dan op de 'Inschrijven'-knop klikt, heb je het meteen zitten!



In zo'n situatie wordt in de Code weergave aangegeven bij welke instructie het fout liep:

```
32          // nagaan hoeveel leerlingen willen inschrijven
33          int aantal = Convert.ToInt32(aantalInschrijvenTextBox.Text);
34
35          // obj                                    of er nog plaats is voor deze leerlingen
36          if ( vl                                    al) == true)
```

Exception Unhandled
System.FormatException: 'De indeling van de invoertekenreeks is onjuist.'

Het is niet zo moeilijk om te bedenken dat het inderdaad niet lukt om de tekst "ABC" (de inhoud van het tekstvak), met `Convert.ToInt32()` om te zetten naar een `int`-getal.

Je zou nu natuurlijk kunnen argumenteren dat de gebruiker maar niet zo *dom* moet zijn om in het tekstvak iets als "ABC" in te tikken en je zo verder niets meer van dit potentiële probleem aantrekken. Een software-ontwikkelaar met een beetje beroepsernst daarentegen zal zo'n fouten willen voorkomen.

Op één of andere manier zullen we dus een **invoercontrole** moeten doen op de input die in de toepassing gebeurt. In dit concrete geval mogen we in het tekstvak `aantalInschrijvenTextBox` enkel getallen toelaten. We tackelen dit thema nu nog niet, maar we komen hier later zeker nog op terug!

3.5 Even oefenen

In het 1e hoofdstuk van de cursus hebben we ons enkel geconcentreerd op code in de Business Layer. Na hoofdstuk 2 kunnen jullie, in de vorm van een Windows Form, een Presentation Layer bouwen bovenop de businesscode.

Nu we in dit 3e hoofdstuk ook de stap naar Properties gezet hebben, mogen we ons wel eens aan wat grotere oefeningen wagen. De bedoeling is om een toepassing uit te werken, waar je beide Layers voor je rekening neemt.

Oefening 3.10 - OpDeWeegschaal. Januari en september zijn de traditionele maanden waar men er werk van maakt om zijn/haar overtollige vakantiekilo's kwijt te raken. Het is een publiek geheim dat fitnesszaken in die periodes de meeste abonnementen verkopen.

Een gezond BMI bereiken, is vaak de doelstelling die men voor ogen heeft als men gaat diëten.

Om vlug je BMI te berekenen wil je een eigen programmaatje maken. Natuurlijk doe je dit volgens de principes van het tweelagenmodel!

Business Layer

In het project `OpDeWeegschaal` vinden jullie de businessklasse `WeightWatcher` terug. Om het BMI te berekenen heb je de grootte (in meter) en het gewicht (in kilo) van een persoon nodig. Dan begrijp je onmiddellijk waarom we volgende velden in deze klasse gedeclareerd hebben:

```
private int _gewicht;
private double _grootte;
```

Aan jullie om deze klasse verder af te werken:

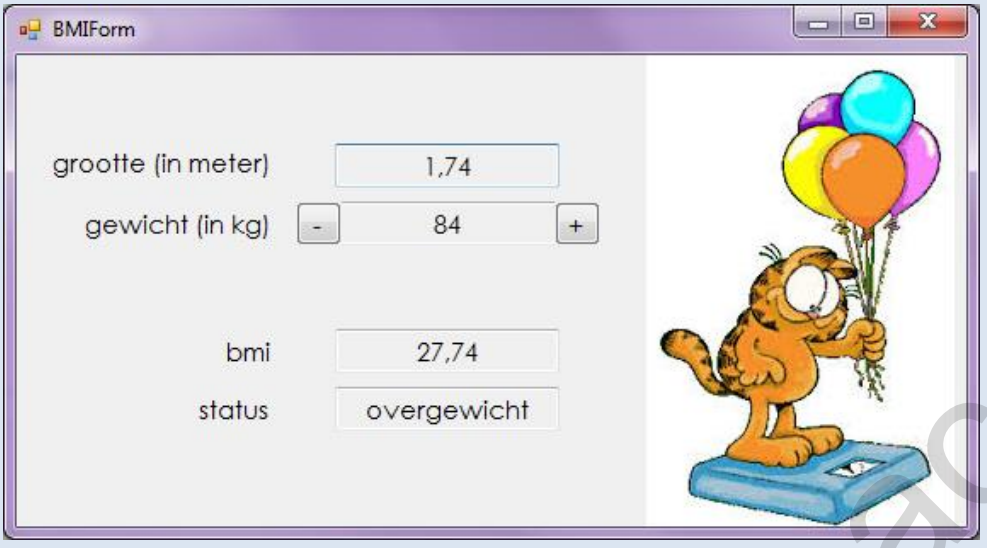
- Bij de constructor van de klasse `WeightWatcher` laat je beide velden instellen via de parameters `int gewicht` en `double grootte`.
- Voor het veld `_gewicht` voorzie je een Property `int Gewicht`. Met deze eigenschap kunnen we het veld `_gewicht` opvragen, maar ook wijzigen.
- Voor het veld `_grootte` voorzie je een Read only Property `double Grootte`.
- De formule achter het BMI is: "gewicht in kg" / ("grootte in meter" * "grootte in meter"). Met de methode `double GeefBMI()` in de klasse `WeightWatcher` laat je dit BMI retourneren.
- Tenslotte doen we er nog een methode `String GeefStatus()` bij. De retourwaarde van deze methode is de tekst:

  --> "ondergewicht" als het bmi in [0,20[valt;
  --> "ok" als het bmi in [20,25] valt;
  --> "overgewicht" als het BMI in]25,30] valt;
  --> "obees" als het BMI groter is dan 30

Het lijkt ons slim om in het venster Immediate onmiddellijk de klasse `WeightWatcher` aan een strenge controle te onderwerpen. Hoe vroeger je fouten opspoort hoe beter!

Presentation Layer

De presentatielaag, of m.a.w. het formuliertje `BMIForm`, zou er als volgt kunnen uitzien:



Wat willen we hiermee bereiken:

Je ziet in dit formulier onmiddellijk je grootte, gewicht en bmi-gegevens. Met het '-' en '+' knopje kan je je gewicht laten variëren en krijg je onmiddellijk feedback wat dit met je bmi doet.

In de weergave Design

Alle labeltjes, tekstvakken en knoppen op het formulier slepen, is een taak die we aan jullie overlaten. De tekstvakken en knoppen MOET je hierbij onmiddellijk een passende naam geven. "Je weet wat we hiervoor afgesproken hebben!" kijft de leraar streng.

De vier tekstvakken zullen straks door onze programmacode ingevuld worden. Het is niet de bedoeling dat de gebruiker hier zelf iets intikt. Zet daarom in het Properties venster de vier tekstvakken op read only.

In de weergave Code

Om het formulier de juiste functionaliteit te geven:

- Het formulier toont de BMI-gegevens van een persoon. Het BMI laten uitrekenen hebben we al eens geprogrammeerd in de klasse `WeightWatcher`.

  1 + 1 = 2, of zorg dat je in de klasse `BMIForm` een `WeightWatcher`-veld declareert. Dit veld houdt de gegevens bij van de persoon die we in het formulier tonen. Dit veld is dus ons businessobject in het formulier.

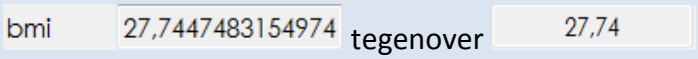
- In de constructor van `BMIForm` zal je het `WeightWatcher`-veld moeten initialiseren. Gebruik hiervoor je persoonlijk gewicht en grootte.

In de constructor zorg je ook dat alle tekstvakken op het formulier al opgevuld worden met de juiste startgegevens.

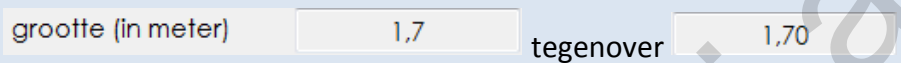
❖ Voor beide knopjes op het formulier zal je een methode moeten toevoegen die reageert op de gebeurtenis `Click`. Deze methoden laten de persoon in het formulier respectievelijk een kilo afvallen of bijkomen.

Opmerking:

- Als je in het tekstvak met het BMI jouw uitvoer vergelijkt met onze screenshot hierboven, zal je waarschijnlijk een verschil opgevallen zijn:



Stel dat je 1,70 meet, dan ziet jouw tekstvak met de grootte er misschien ook wat vreemd uit:



De rode draad is dat wij er blijkbaar in geslaagd zijn om deze getallen (altijd) met twee cijfers na de komma weer te geven.

We gokken dat jullie bij de instructies waarmee je de tekstvakken met de grootte en het BMI laat opvullen, een `ToString()` gebruikten. Of iets als:

```
grootteTextBox.Text = _weightWatcher.Grootte.ToString();
bmiTextBox.Text = _weightWatcher.GeefBmi().ToString();
```

Wij hebben bij die `ToString()`-methode de parameter **"0.00"** vermeld.

```
grootteTextBox.Text = _weightWatcher.Grootte.ToString("0.00");
bmiTextBox.Text = _weightWatcher.GeefBmi().ToString("0.00");
```

Met die extra parameter bij de `ToString()`-methode, kan je dus blijkbaar forceren dat bij het omzetten van een getal naar het type `String` er een notatie met twee decimalen gebruikt wordt.

Onthoud 3.8 Als je een numerieke waarde laat omzetten naar een `String` met behulp van **de methode `ToString()`**, dan kan je via een **parameter** aangeven om de retourwaarde in een **bepaalde notatie** te zetten.

Enkele voorbeelden in het venster Immediate:

```
Immediate Window
? (0.7).ToString("0.00")
"0,70"
? (0.7).ToString("C")
"€ 0,70"
? (0.7).ToString("P")
"70,00%"
? (0.7).ToString("P0")
"70%"
```

Conclusie:

- De parameter **"0.00"**, laat het getal omzetten naar tekst waarbij een notatie met **2 cijfers na de komma** gebruikt wordt;
- de parameter **"C"**, laat het getal omzetten naar een tekst met **Valuta-notatie**;
- de parameter **"P"**, laat het getal omzetten naar een tekst met een **procent-notatie** (met standaard twee decimalen); hierbij doet men standaard het getal maal 100;
- de parameter **"P0"**, laat het getal omzetten naar een tekst met een **procent-notatie** zonder decimalen.

Op https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings en https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-numeric-format-strings krijg je een uitgebreid overzicht van alle mogelijkheden met deze parameter.

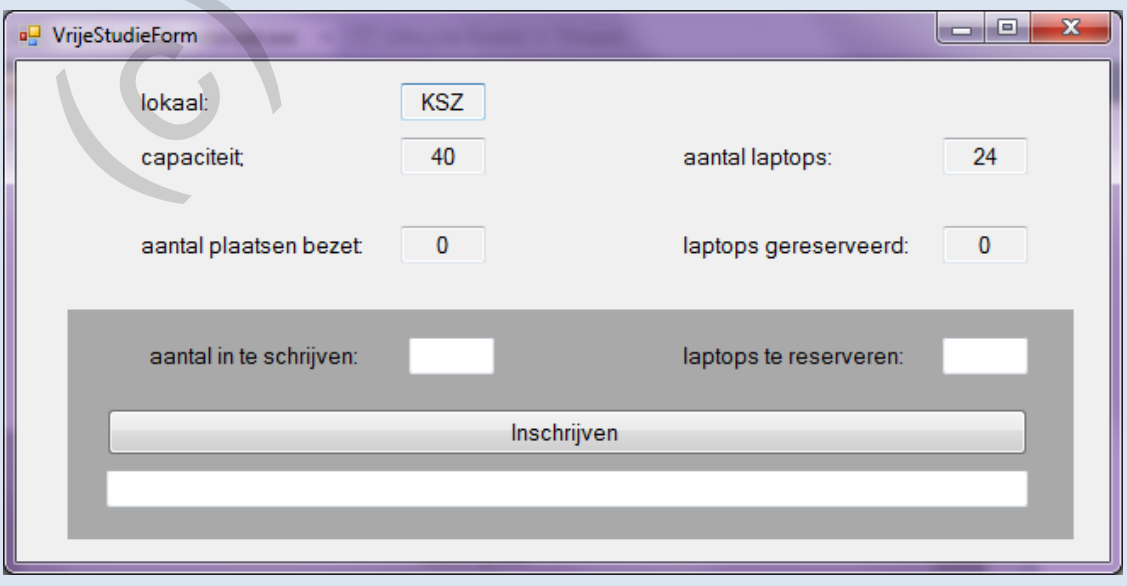
Oefening 3.11 - LeerlingenSecretariaat. In het project `LeerlingenSecretariaat` (zie **_hoofdstuk_3/_oefeningen**) vind je onze uitgewerkte oplossing voor de "Vrije studie"-case die we vanaf pagina 106 in de cursus uitgebreid besproken hebben. Lees je nog eens in.

Heel kort samengevat: Via onze toepassing kan je in de loop van de dag (groepen) leerlingen laten inschrijven voor een vrije studie die 's avonds plaatsvindt en krijg je onmiddellijk een foutboodschap te zien zodra blijkt dat het lokaal te vol raakt.

We moeten zeggen dat men op het leerlingensecretariaat in de wolken was over ons formuliertje. Dan volgde er een "maar", en een "maar" dat betekent dat de programmeur weer aan de slag moet …

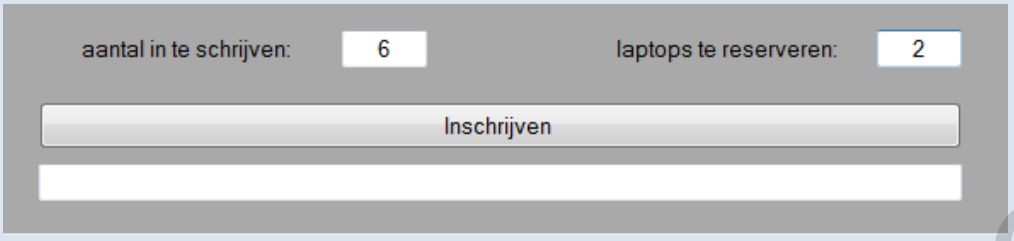
Blijkt nu dat leerlingen in de vrije studie ook een laptop kunnen reserveren en het aantal laptops is natuurlijk beperkt. Op het leerlingensecretariaat willen ze meteen de leerlingen briefen als er geen laptops meer voor hen zullen zijn.

We willen naar volgend formuliertje toewerken:

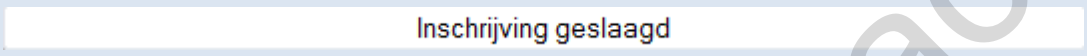


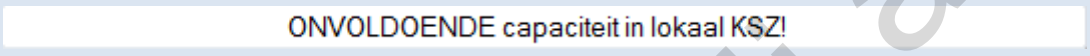
Als er leerlingen zich komen inschrijven in het leerlingensecretariaat, zal men in het donkergrijze kadertje voortaan ook opgeven hoeveel laptops deze leerlingen willen gebruiken.

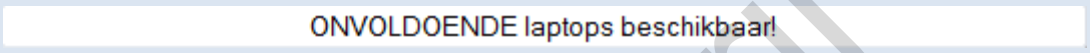
Bijvoorbeeld een groep van 6 leerlingen wil naar de vrije studie en zou hierbij graag op 2 laptops werken:



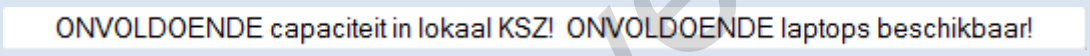
Na het klikken op de knop 'Inschrijven' zijn nu volgende boodschappen mogelijk:







of



Enkel als er voldoende zitplaatsen én voldoende laptops zijn, laten we de inschrijving voltooien.

Om deze uitbreiding te implementeren, valt er zowel in de **Business** als in de **Presentation Layer** werk te doen!

Business Layer

In het Class-diagram van de klasse `VrijeStudie` hebben we hieronder de nodige uitbreidingen in het vet geplaatst.

VrijeStudie
- String _lokaal - int _capaciteit - int _gereserveerdePlaatsen **- int _laptops** **- int _gereserveerdeLaptops**
+ VrijeStudie(String,int,**int**)
+ Read only Property String Lokaal + Read only Property int Capaciteit + Property int GereserveerdePlaatsen **+ Read only Property int Laptops** **+ Property int GereserveerdeLaptops** + bool IsErNogPlaats(int) **+ bool ZijnErNogLaptops(int)**

De nieuwigheden:

- In een veld `int _laptops` houd je bij hoeveel laptops er in het studielokaal beschikbaar zijn. Het veld `int _gereserveerdeLaptops` vertelt hoeveel van die laptops reeds gereserveerd zijn door leerlingen.
- Aan de constructor hebben we een derde parameter toegevoegd. Bij het aanmaken van een nieuw `VrijeStudie`-object zullen we voortaan via die parameter opgeven hoeveel laptops er in het studielokaal zijn.
- Met `int Laptops` en `int GereserveerdeLaptops` voorzie je voor de twee extra velden ook een aparte eigenschap.
- Bij de methode `bool ZijnErNogLaptops(int)` geef je als parameter mee hoeveel laptops er moeten gereserveerd worden. Deze methode retourneert **true** als dit aantal laptops nog beschikbaar is; anders is de retourwaarde **false**.

Presentation Layer

In `VrijeStudieForm` zal je eerst de extra tekstvakken moeten plaatsen. Daarna kan je aan de slag in de programmacode:

- Bij de constructor van `VrijeStudieForm` gebeurt de initialisatie van het veld `_vrijeStudie` (dit is het businessobject in het formulier). Bij deze initialisatie moet je nu ook opgeven hoeveel laptops er in het studielokaal beschikbaar zijn.

  Stel dat in ons studielokaal KSZ een laptopkast staat met 24 laptops.
- De screenshot op pagina 118 toont hoe het formulier eruitziet als het initieel geopend wordt. Dit vraagt van jou nog een beetje extra code in de constructor.
- De foutboodschappen die je op pagina 119 ziet, zouden voldoende moeten zijn om de aanpassingen te programmeren in de methode die reageert op de gebeurtenis `Click` van de 'Inschrijven'-knop.

4 Return of the Properties

Als wij – net na hoofdstuk 3 – een brainstorm houden over wat **Properties** (**eigenschappen**) zijn, komen waarschijnlijk volgende antwoorden naar boven:

- "De Properties in C# vervangen de accessoren en mutatoren uit Java";
- "Een Property in C# is een beknoptere notatie van een accessor en/of mutator in Java";
- "Bij de definitie van een Property kan je een get{} en/of een set{} statement voorzien";
- "Via een Property kan men aan een object de waarde van een veld opvragen/schrijven".

Allemaal mooie antwoorden die bewijzen dat de cursusnota's tot hier goed doorgenomen werden. Maar dan komt die ene dwarse leerling met volgend antwoord:

- "In Visual Studio heb je een **Properties venster;** in dat venster zie je alle Properties die horen bij het formulierelement dat je in de weergave Design geselecteerd hebt".

Oei, dit gaat hier *schijnbaar* over iets helemaal anders; over een *'ander' soort Properties*?

De insteek van dit hoofdstuk is om het te hebben over die Properties uit het Properties venster. Eens je beseft dat je die Properties ook via je programmacode kan aanspreken en manipuleren, zal er een hele nieuwe wereld open gaan!

We zijn echter pas in ons opzet geslaagd, als je straks ook inziet dat enerzijds "de Properties die we in een klasse zelf definiëren met een get{}- en set{}-statement" en anderzijds "de Properties uit het Properties venster" echt wel *dezelfde Properties* zijn.

In onze uiteenzetting dat er in de theorie van objectgeoriënteerd programmeren maar *één soort* Properties is, zal de **Object Browser** van onschatbare waarde zijn.

4.1 Kennismaken met het project `Showroom`

We gebruiken het nieuwe project `Showroom` als bronmateriaal in dit hoofdstuk. Fantaseer jezelf even in de showroom van een Peugeot-garage.

Als je het project start, wordt een formulier geopend waar klanten kunnen nagaan hoeveel een nieuwe Partner Tepee zal kosten, afhankelijk van de opties die ze al dan niet verkiezen.

Om de opties die de klant neemt te visualiseren, laten we naast de geselecteerde opties een klein passend icoontje verschijnen. Een optie aan- of uitvinken betekent dus niet enkel dat

de totaalprijs van de wagen verandert, maar er is ook een dynamiek op het formulier dat er dan een icoontje zichtbaar wordt of net zal verdwijnen.

Tweelagengewijs zal je in dit project een Business Layer (de klasse `PartnerTepee`) en een Presentation Layer (de klasse `PartnerTepeeForm`) kunnen onderscheiden.

4.1.1 De Business Layer

Alhoewel we ons in dit project voornamelijk op de Presentation Layer zullen richten, moeten we het toch even hebben over de Business Layer. De core van een toepassing steekt immers altijd in de businesslaag.

We printen hieronder **een deel** van de code van de businessklasse `PartnerTepee`:

```
public class PartnerTepee
{
    private bool _autoradio;
    private bool _airco;

    public PartnerTepee()
    {
        _autoradio = false;
        _airco = false;
    }

    public bool Autoradio
    {
        get { return _autoradio; }
        set { _autoradio = value; }
    }

    public bool Airco
    {
        get { return _airco; }
        set { _airco = value; }
    }
```

```
    public double GeefPrijs()
    {
        //basisprijs zonder opties is 13904
        double prijs = 13904;

        if (_autoradio == true)
        {
            prijs += 450;
        }

        if (_airco == true)
        {
            prijs += 1280;
        }

        return prijs;
    }
}
```

Met deze klasse `PartnerTepee` modelleert men in de Peugeot-garage dus hun wagens van het merk Partner Tepee.

In de `bool`-velden `_autoradio` en `_airco` houdt men bij of een individuele wagen al dan niet over deze opties beschikt. De Properties `bool Autoradio` en `bool Airco` spreken hierbij voor zich.

Verder beschikt de klasse over een methode `double GeefPrijs()` die op basis van de gekozen opties (= de waarde van de `bool`-velden) de prijs van de wagen retourneert.

Oefening 4.1 In het venster Immediate declareren en initialiseren we volgend object `wagen`:

```
Immediate Window
? PartnerTepee wagen = new PartnerTepee();
```

Wat is nu de toestand van dit object `wagen`?

Een klant neemt er graag de opties "autoradio" en "airconditioning" bij en wil weten hoeveel de wagen dán kost. Welke instructies tikt de garagist in het venster Immediate om het juiste antwoord aan de klant te kunnen geven?

In het project `Showroom` hebben we trouwens nog drie extra opties (of aldus drie extra `bool`-velden en Properties) opgenomen in de klasse `PartnerTepee`. Als een klant voor de goedkoopste Partner Tepee versie wil gaan, zal hij dus tegen vijf opties neen moeten zeggen.

4.1.2 De Presentation Layer

Het formulier `PartnerTepeeForm` vormt de presentatielaag in het project `Showroom`.

Een klant die een nieuwe Partner Tepee met de opties 'autoradio' en 'airconditioning' bestelt, zal met dit formulier heel gemakkelijk de juiste prijs kunnen opvragen. Het is gewoon een kwestie om de juiste selectievakjes in het formulier aan te vinken.

We printen hieronder **een deel** van de code van de presentatieklasse `PartnerTepeeForm`:

```
public partial class PartnerTepeeForm : Form
{
    private PartnerTepee _wagen;

    public PartnerTepeeForm()
    {
        InitializeComponent();

        _wagen = new PartnerTepee();

        prijsTextBox.Text = _wagen.GeefPrijs().ToString("C");
    }

    private void autoradioCheckBox_CheckedChanged(object sender, EventArgs e)
    {

        if (autoradioCheckBox.Checked == true)
        {
            autoradioPictureBox.Visible = true;
            _wagen.Autoradio = true;
        }
        else
        {
            autoradioPictureBox.Visible = false;
            _wagen.Autoradio = false;
        }

        prijsTextBox.Text = _wagen.GeefPrijs().ToString("C");
    }
```

Zonder al in detail op deze code in te gaan, enkele inleidende vraagjes om na te gaan of je de grote lijnen in deze formulierklasse herkent:

Oefening 4.2 Welk object (geef naam en type) doet dienst als **businessobject** binnen de klasse `PartnerTepeeForm`?

Dit object zal in het formulier dus bijhouden welke opties al dan niet ingeschakeld zijn. Aan dit object zullen we ook kunnen vragen om de bijhorende totaalprijs te berekenen.

Oefening 4.3 Je vindt in bovenstaande codefragment de volgende header terug:

```
private void autoradioCheckBox_CheckedChanged(object sender, EventArgs e) {}
```

Deze header doet heel sterk vermoeden dat deze methode zal reageren op een **gebeurtenis** (**Event**).

Formuleer - met je eigen woorden – bij welke gebeurtenis in het formulier deze methode zal geactiveerd worden:

Als we je gevraagd hadden om in het stukje geprinte code de Properties aan te duiden, dan zou je waarschijnlijk naar de Property `Autoradio` van het businessobject `_wagen` gewezen hebben?

Je ziet hieronder de instructies waarmee we de 'Autoradio'-optie via de gelijknamige Property aan- en uitzetten voor het object `_wagen`:

```
if (autoradioCheckBox.Checked == true)
{
    autoradioPictureBox.Visible = true;
    _wagen.Autoradio = true;
}
else
{
    autoradioPictureBox.Visible = false;
    _wagen.Autoradio = false;
}

prijsTextBox.Text = _wagen.GeefPrijs().ToString("C");
```

Spoileralert! Straks zal blijken dat er in bovenstaand kadertje nog drie andere Properties verscholen zitten!

4.2 Objecteigenschappen

Vóór we straks terugkeren naar de programmacode bij het formulier `PartnerTepeeForm` willen we eerst het begrip **objecteigenschap** definiëren:

Onthoud 4.1 In de Design weergave van een formulier gebruiken we het Properties venster om de elementen op het formulier (en het formulier zelf) in te stellen. Zo'n afzonderlijk in te stellen kenmerk uit het Properties venster duiden we aan als een **objecteigenschap**.

Bijvoorbeeld: Wil je de afbeelding van de wagen op het formulier kleiner maken, dan zal je de objecteigenschappen `Width` en `Height` moeten aanpassen in het Properties venster. Wil je het formulier `ParnerTepeeForm` een andere achtergrondkleur geven, dan heb je met de objecteigenschap `BackColor` van doen.

Toch nog even volgende twee weergaven van het Properties venster onderscheiden:

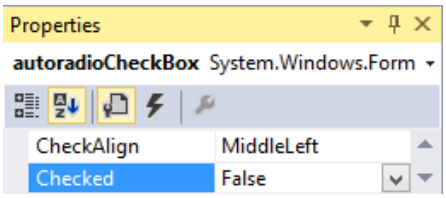
- Via het knopje () vraag je de *objecteigenschappen* van het geselecteerde element in het Properties venster op.
- Met het knopje () daarentegen kom je op de lijst van *gebeurtenissen (Events)* terecht waarvoor je aan het geselecteerde element methoden kan koppelen.

4.2.1 Objecteigenschappen in het Properties venster

Bij het formulier `PartnerTepeeForm` gaan we er enkele objecteigenschappen uitlichten:

De objecteigenschap `Checked` van een `CheckBox`

Als je in de **weergave Design** het selectievakje `autoradioCheckBox` selecteert, vind je in het Properties venster de objecteigenschap `Checked` terug:

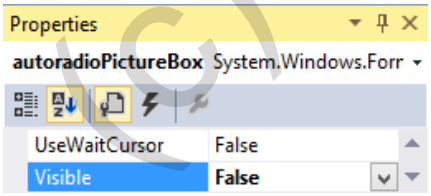


Deze objecteigenschap `Checked` geeft aan of het selectievakje `autoradioCheckBox` al dan niet aangevinkt is. Omdat we deze objecteigenschap de waarde **true** of **false** kunnen geven, raad je dat dit een objecteigenschap van het type `bool` is.

De objecteigenschap `Visible` van een `PictureBox`

De afbeeldingen op ons formulier zijn `PictureBox`-elementen. Vanuit de **Toolbox** hebben we dus vijf zo'n `PictureBox`-elementen naar de linkerzijde van het formulier gesleept.

Als je de bovenste afbeelding `autoradioPictureBox` selecteert, vind je in het Properties venster de objecteigenschap `Visible` terug:



Deze objecteigenschap `Visible` geeft aan of de afbeelding `autoradioPictureBox` al dan niet zichtbaar is. Omdat we deze objecteigenschap de waarde **true** of **false** kunnen geven, bedenk je dat dit een objecteigenschap van het type `bool` is.

Opgelet:

- De objecteigenschap `Visible` wordt pas toegepast in de **run time** van het formulier. In de Design weergave blijft de `PictureBox` steeds zichtbaar.

De objecteigenschap `Visible` staat bij de vijf `PictureBox`-elementen links op het formulier op **false**. Bij de vijf `CheckBox`-elementen staat de objecteigenschap `Checked` op **false**.

Bij het starten van het formulier zullen de vijf selectievakjes daarom NIET aangevinkt zijn en worden de vijf afbeeldingen links initieel NIET weergegeven in het formulier.

Oefening 4.4 Als we het project `Showroom` starten, krijgen we nu initieel de Partner Tepee zonder opties te zien. Een autoverkoper in hart en nieren gaat hier niet mee akkoord! "Splits de klant maar onmiddellijk alle opties in zijn maag" zal het hele gild in koor roepen.

We passen graag het formulier aan:

- Doe **in het Properties venster** de nodige aanpassingen zodat bij het openen van het formulier de vijf selectievakjes aangevinkt zijn en de vijf afbeeldingen daarnaast onmiddellijk zichtbaar staan. Hiervoor hoef je nog NIETS te programmeren.
- Om in het formulier initieel ook de correcte prijs (inclusief alle opties) te zien, zullen we wel even naar de **weergave Code** van het formulier moeten.

  Bij de constructor van `PartnerTepeeForm` lieten we eerst en vooral het businessobject `PartnerTepee _wagen` initialiseren. Bij een nieuw `ParnterTepee`-object staan alle opties echter nog op false.

  Vooraleer we in de constructor van `PartnerTepeeForm` aan dit object `_wagen` de prijs opvragen om deze in het tekstvak `prijsTextBox` te printen, zal je `_wagen` dus moeten updaten zodat alle opties er wel *ingeschakeld* zijn!

4.2.2 Objecteigenschappen in programmacode

Na een *heel lange* aanloop zijn we eindelijk bij ons Eureka-moment beland: Het blijkt dat de objecteigenschappen ook perfect bereikbaar zijn vanuit onze programmacode!!!

We nemen er opnieuw de methode `autoradioCheckBox_CheckedChanged()` bij. Onze objecteigenschappen `Checked` en `Visible` die we er hierboven al even uitgelicht hebben in het Properties venster, vind je hieronder in het vet terug.

```
private void autoradioCheckBox_CheckedChanged(object sender, EventArgs e)
{

    if (autoradioCheckBox.Checked == true)
    {
        autoradioPictureBox.Visible = true;
        _wagen.Autoradio = true;
    }
    else
    {
        autoradioPictureBox.Visible = false;
        _wagen.Autoradio = false;
    }

    prijsTextBox.Text = _wagen.Prijs().ToString("C");
}
```

Besef dat deze methode `autoradioCheckBox_CheckedChanged` telkens geactiveerd wordt als **het selectievakje** `autoradioCheckBox` op het formulier **aan- of uitgevinkt** wordt. Het formulier werkt dus echt zo:

- Je vinkt het selectievakje `autoradioCheckBox` aan → de methode `autoradio-CheckBox_CheckedChanged` wordt opgeroepen.
- Je doet daarna het vinkje weg in `autoradioCheckBox` → diezelfde methode `auto-radioCheckBox_CheckedChanged` wordt opnieuw opgestart.

We moeten in deze methode daarom allereerst achterhalen of de gebruiker het selectievakje i.v.m. de autoradio-optie nu wel aan- of net uitgevinkt heeft?

We doen dit door de objecteigenschap `Checked` van het selectievakje `autoradioCheck-Box` na te kijken:

```
if (autoradioCheckBox.Checked == true)
```

We **raadplegen** met deze if() dus de waarde van de objecteigenschap `Checked` van dit selectievakje. Daar deze objecteigenschap `Checked` bijhoudt of een selectievakje aan- of uitgevinkt staat, hebben we zo dus de staat van het selectievakje beet.

Onthoud 4.2 Met de expressie **element.objecteigenschap** vraag je aan een bepaald element de waarde op van een bepaalde **objecteigenschap**.

Afhankelijk of de gebruiker het selectievakje in- of uitgeschakeld heeft, moeten we de afbeelding met het autoradio-icoontje tonen of verbergen. We regelen dit door de objecteigenschap `Visible` van deze afbeelding te manipuleren.

Om de afbeelding zichtbaar te maken, gebruiken we deze instructie:

```
autoradioPictureBox.Visible = true;
```

De omgekeerde bewerking (de afbeelding terug verbergen) kan dan uiteraard met:

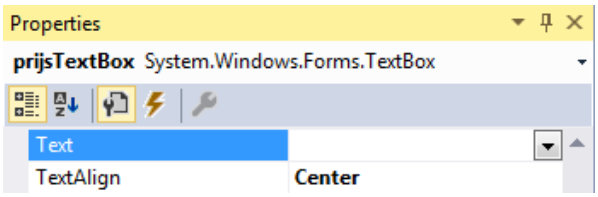
```
autoradioPictureBox.Visible = false;
```

We **wijzigen** met deze instructies dus zonder probleem de waarde van de objecteigenschap `Visible` van deze afbeelding.

Onthoud 4.3 Met de instructie "**`element.objecteigenschap = ...;`**" geef je een **objecteigenschap** van een element een nieuwe waarde.

Nu kunnen we de `Text`, waarmee we al drie hoofdstukken lang **de inhoud** van een tekstvak aanspreken, ook in het juiste plaatje zetten. `Text` is namelijk ook een objecteigenschap en dit voor `TextBox`-elementen.

Als je in de weergave Design op `PartnerTepeeForm` het tekstvak `prijsTextBox` selecteert, vind je in het Properties venster inderdaad deze objecteigenschap `Text` terug.



Deze objecteigenschap `Text` konden jullie al in je code gebruiken. Hieronder de instructie waarmee we de totaalprijs van de wagen in het tekstvak printen:

```
prijsTextBox.Text = _wagen.Prijs().ToString("C");
```

Merk op dat we in `PartnerTepeeForm` ook voor de vier andere selectievakjes een gelijk–aardige methode voorzien hebben die bij de **Event** `CheckedChanged` het overeenkomstig icoontje laat verschijnen/verbergen en die de prijs in `prijsTextBox` laat vernieuwen.

Conclusie

Dat je in het Properties venster objecteigenschappen van elementen op het formulier (en van het formulier zelf) kan instellen en dat deze instellingen initieel van toepassing zijn als je het formulier opent, wist je eigenlijk wel al.

Dat je diezelfde objecteigenschappen ook via programmacode kan lezen/manipuleren is daarentegen een nieuwe mijlpaal in jullie 'Visual Studio'-curriculum!

Formulierelementen laten verschijnen of verbergen, van kleur laten schrikken, van positie laten switchen, laten inkrimpen of uitrekken, … als je hier in het Properties venster maar een objecteigenschap voor vindt, dan kan je het ook programmeren.

4.2.3 Objecteigenschappen in de Object Browser

Dat we de objecteigenschappen uit het Properties venster ook via programmacode kunnen opvragen/wijzigen is een blikopener van formaat. Ga op een koude herfstdag eens in het Properties venster rondstruinen en er zullen haast vanzelf nieuwe programmeerideeën naar boven komen drijven …

Toch willen we enkele bedenkingen maken bij het Properties venster:

- Als je bij een objecteigenschap in het Properties venster ziet dat de twee mogelijke waarden **true** en **false** zijn, dan beredeneer je onmiddellijk dat het om een object-eigenschap van het type `bool` gaat.

  In andere situaties is het niet altijd duidelijk wat het type van een objecteigenschap is, terwijl dit toch wel belangrijke informatie is als je met zo'n objecteigenschap in je programmacode aan de slag wil gaan.

❖ Het blijkt ook dat niet altijd *alle* beschikbare eigenschappen in het Properties venster opgenomen worden!

Zo weten wij – en nu jullie ook - dat elk besturingselement dat je op een formulier plaatst over een objecteigenschap `Left` en `Top` beschikt, die respectievelijk de afstand t.o.v. de linker- en de bovenrand van het formulier voorstelt (of dus m.a.w. met `Left` en `Top` wordt de locatie van het element op het formulier vastgelegd).

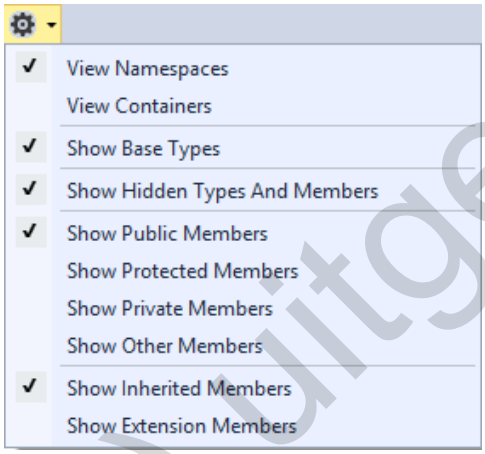
Wel, deze objecteigenschappen `Left` en `Top` staan toch NIET in het Properties venster.

Alhoewel het Properties venster een schat aan snelle informatie biedt over welke object–eigenschappen beschikbaar zijn, heeft dit venster dus toch *soms* zijn tekortkomingen. Daarom doen de kenners het liever met de **Object Browser**, waar je wel *altijd* feilloos op mag vertrouwen.

De Object Browser laat je openen in Visual Studio via **View** | **Object Browser**.

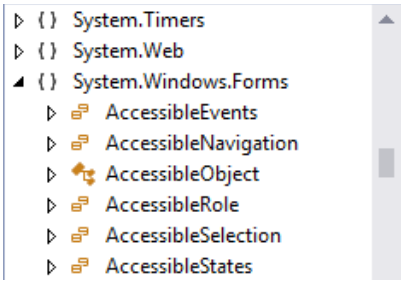
Opmerking:

○ Via de karrenwiel-knop () kan je vastleggen hoe de Object Browser georganiseerd wordt. Je neemt op jouw computer best onderstaande instellingen over:

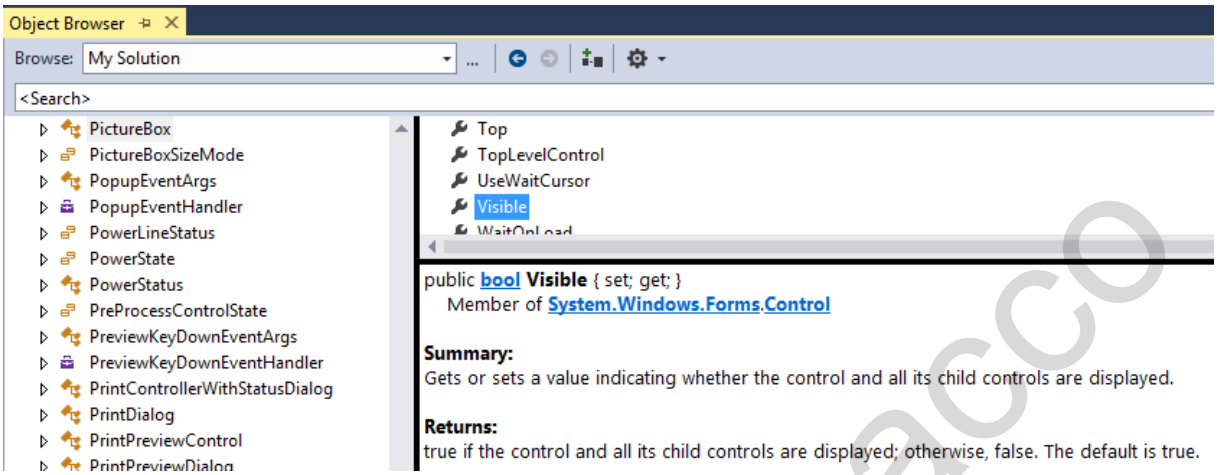


Met volgend trucje onthoud je makkelijk hoe wij de Object Browser gebruiken: Bij elke *groep* vink je enkel de eerste optie aan!

In het linker venster van de Object Browser zie je alle **namespaces** die in Visual Studio gekend zijn. Wij klappen de namespace `System.Windows.Forms` open. Deze namespace verzamelt de klassen die te maken hebben met Windows formulieren.



Weet je nog dat we in `PartnerTepeeForm` de afbeeldingen al dan niet lieten weergeven via de objecteigenschap `Visible`? Als je nu in het linker venster het element `PictureBox` selecteert, dan zal je in het venster aan de rechterkant deze objecteigenschap `Visible` terugvinden.



In het venster rechts onderaan krijg je dan iets meer informatie over deze objecteigenschap `Visible`:

- Bovenaan in dit venster staat een 'soort' header:

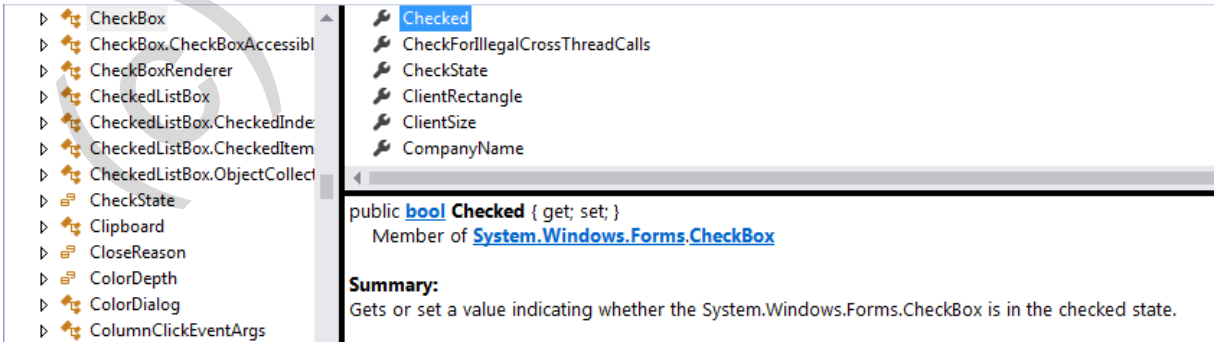
  ```
  public bool Visible { set; get;}
  ```

  Dit geeft duidelijk aan dat de objecteigenschap `Visible` van het type `bool` is. De `get` en `set` betekent dat je deze objecteigenschap kan opvragen en wijzigen.

- Bij **Summary** volgt nog een beknopte beschrijving over deze objecteigenschap.

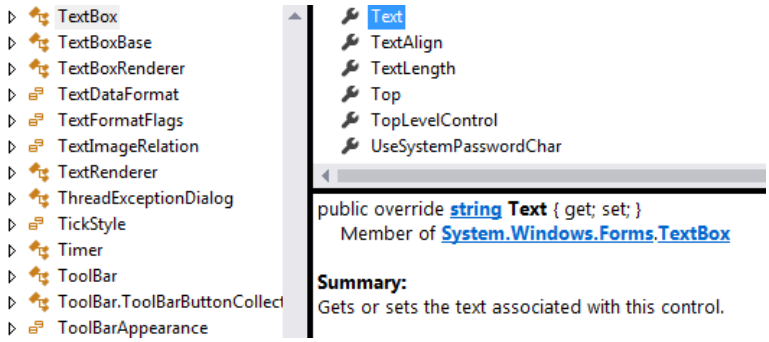
Ook die andere objecteigenschappen die we toepasten in `PartnerTepeeForm` zal je in de Object Browser naar boven kunnen halen:

De objecteigenschap `Checked` bij het element `CheckBox`



Zoals verwacht wordt hier bevestigd dat de objecteigenschap `Checked` van het type `bool` is.

De objecteigenschap `Text` bij het type `TextBox`



Hier krijgen we het bewijs dat de objecteigenschap `Text` - of de inhoud van een tekstvak - van het type `String` is.

Om in de Object Browser het overzicht op te vragen van de objecteigenschappen die beschikbaar zijn bij een bepaald element, kan je dit stappenplan volgen:

1. Selecteer in het venster links het type van het element. Voor onze toepassingen zal dit voornamelijk binnen de namespace `System.Windows.Forms` zijn.
2. De objecteigenschappen worden in het venster rechts bovenaan aangeduid met het icoontje: .
3. In het venster rechts onderaan krijg je wat extra informatie over de geselecteerde objecteigenschap.

Oefening 4.5 Op onze formulieren hebben we al vaak knoppen gezet. Het type van zulke knoppen is `Button`. We laten jullie in de Object Browser eens snuisteren naar de object–eigenschappen bij dit type `Button` (zie namespace `System.Windows.Forms`).

De objecteigenschap `Width`

Met deze objecteigenschap kunnen we de breedte van een knop instellen.

- Wat is het type van deze objecteigenschap `Width`?
- Wat is de gebruikte eenheid bij de objecteigenschap `Width`?

De objecteigenschap `Bottom`

- Leg in eigen woorden uit wat de **Summary** bij de objecteigenschap `Bottom` betekent.

- Hoe zie je dat `Bottom` een **Read only** objecteigenschap is?

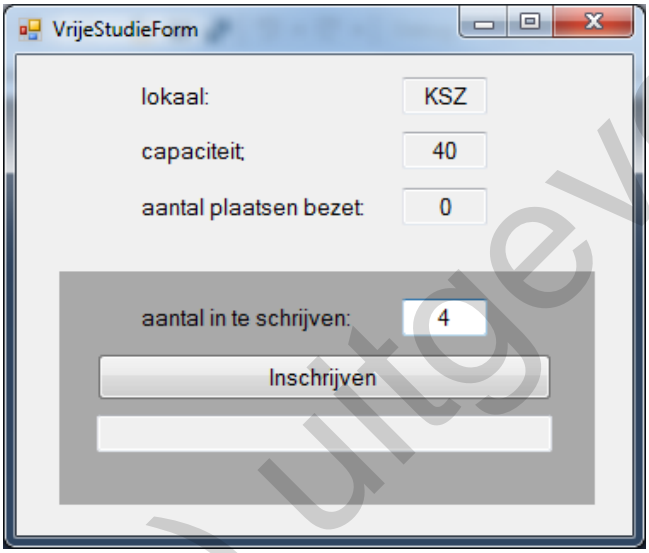
Op het einde van dit hoofdstuk komen we nog eens uitgebreid terug op de Object Browser om jullie zo nog een **veel ruimere** kijk te geven op de informatie die je hier kan terugvinden. We laten er de Object Browser helemaal inpassen in onze theorie over objectgeoriënteerd programmeren.

4.3 Invoercontrole

Bij onze toepassingen wordt vaak verwacht dat de gebruiker op een formulier het één of andere invoert.

Neem bijvoorbeeld de LeerlingenSecretariaat-toepassing die we vanaf pagina 106 opgebouwd hebben (zie de bronbestanden bij **_hoofdstuk_3/_voorbeelden**). De personeelsleden van het leerlingensecretariaat zullen daar in een tekstvak op `VrijeStudieForm` intikken hoeveel leerlingen een plaatsje willen reserveren voor de vrije studie.

Hieronder geeft men bijvoorbeeld op dat een groepje van 4 leerlingen naar de vrije studie wil komen (om een groepswerk voor te bereiden, of zoiets):



Wij verwachten dat men in het betreffende tekstvak een aantal (een *getal*) opgeeft. In een tekstvak kan je echter *van alles* typen! Het spijtige is dat een gebruiker dit formulier makkelijk kan saboteren door in dit tekstvak geen getal, maar een *blabla*-tekstje, in te tikken. Het formulier zal in deze situatie onmiddellijk crashen na het klikken op de 'Inschrijven'-knop.

Er is dus nood om op één of andere manier na te gaan of er in het tekstvak wel een geldige waarde ingetikt werd. Algemener geformuleerd: "De invoer die in onze programma's gebeurt, dient gecontroleerd te worden."!

Onthoud 4.4 De maatregelen die je neemt om te zorgen dat er ten allen tijde een geldige invoer gebeurt in je formulieren, benoemen we met de term **invoercontrole**.

Op het eerste gezicht denk je misschien dat deze invoercontrole extra programmacode veronderstelt. Visual Studio helpt ons echter al enorm op weg door - naast de `TextBox` - heel wat andere formulierelementen te voorzien. Deze formulierelementen kunnen verhinderen dat de gebruiker iets fout invoert.

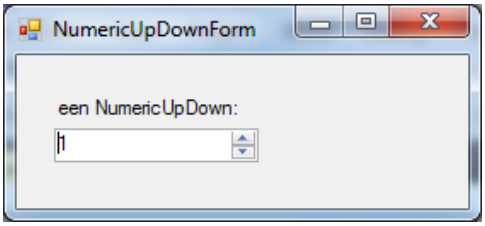
Verder in de cursus maken we bijvoorbeeld eens gebruik van een `ComboBox` (= een keuze-lijst). Dit verplicht de gebruiker om één van de voorgedefinieerde (geldige) opties te kiezen.

Wij willen er nu graag eens het `NumericUpDown`-element uit lichten:

In zo'n element kan je enkel getallen, binnen een bepaalde range, invoeren. Als invoer-controle zal dit vaak al voldoende zijn.

4.3.1 Invoercontrole met een `NumericUpDown`-element

In het project `TestNumericUpDown` hebben we het formulier `TestNumericUpDownForm` beschikbaar, waarop we vanuit de ToolBox een `NumericUpDown`-element sleepten.



Start het project even op. Je zal merken dat je via de pijltjes dit element van 1 tot 10 kan laten variëren. Zonder de pijltjes te gebruiken, kan je in dit vak ook gewoon zelf een getal in het bereik van 1 tot 10 intikken.

Als je in een toepassing verlangt dat een gebruiker een getal van 1 tot 10 invoert, dan heb je dit dus al helemaal geregeld met deze specifieke `NumericUpDown`. Inzake invoercontrole hoeft dan verder niets meer te gebeuren.

In onderstaande vragenlijst overlopen we enkele belangrijke objecteigenschappen van dit `NumericUpDown`-element.

Oefening 4.6 Op het formulier `TestNumericUpDownForm` staat een `NumericUpDown`-element met de naam `vakNumericUpDown`.

In het Properties venster én in de Object Browser (namespace: `System.Windows.Forms`) zou je de antwoorden moeten vinden op onderstaande vragen.

- Bij een tekstvak bepaalt de objecteigenschap `Text` wat de inhoud ervan is (wist je al lang). Maar welke objecteigenschap heeft diezelfde betekenis bij een `NumericUpDown`-element, of anders geformuleerd met welke objecteigenschap kan je zo bepalen welk

getalletje er initieel weergegeven wordt in een `NumericUpDown`? Wat is het type van deze objecteigenschap?

- Je kan bij een `NumericUpDown`-element een ondergrens instellen. Met het pijltje omlaag kan je niet onder deze waarde gaan. Welke objecteigenschap regelt dit? Wat is het type van deze objecteigenschap?

- Je kan bij een `NumericUpDown`-element een bovengrens instellen. Met het pijltje omhoog kan je niet boven deze waarde komen. Welke objecteigenschap regelt dit? Wat is het type van deze objecteigenschap?

- De 'sprong' die je met de pijltjes maakt, kan je ook via een objecteigenschap vastleggen. Standaard ga je het getal 1 verhogen of verlagen, maar je kan hier evengoed 'per 2' of 'per 10' van maken. Welke objecteigenschap regelt dit? Wat is het type van deze eigenschap?

- Stel dat we willen dat de gebruiker wél de pijltjes mag gebruiken om het getal aan te passen, maar niet zelf rechtstreeks een waarde kan intikken in het `NumericUpDown`-element. Welke objecteigenschap moet je hiervoor instellen? Wat is het type van deze eigenschap?

Als samenvattende oefening mag je in het Properties venster bovenstaande Properties nog eens instellen, zodat je van `vakNumericUpDown` een element maakt waarmee je een getal tussen 0 en 20 kan selecteren. Je mag enkel via de pijltjes de inhoud kunnen wijzigen (je kan dus zelf niets intikken in het vak), waarbij je telkens een sprong per 2 maakt. De startwaarde moet 10 zijn.

Bij de antwoorden hierboven is een aantal keer het type `decimal` opgedoken. Omdat het de eerste keer is dat je met dit type te maken krijgt, moeten we hier toch even bij stilstaan.

Het gegevenstype `decimal`

`decimal` is een primitief gegevenstype waarmee je getallen met cijfers na de komma kan voorstellen. Dit gegevenstype is dus te vergelijken met het type `double`. Toch zijn er enkele essentiële verschillen!

Een probleem met het gegevenstype `double` is dat dit type gevoelig is aan de vreemdste afrondingseffecten.

Kijk maar eens naar onderstaand voorbeeldje in het venster Immediate:

```
Immediate Window
? double d1 = 1.4;
1.4
? double d2 = d1 * 3;
4.1999999999999993
```

Je had natuurlijk verwacht dat de variabele `d2` hierboven geïnitialiseerd zou zijn op 1.4 * 3 of dus 4.2, maar om één of andere reden trad er een verschilletje op van 0,0000000000000007 (eat your heart out, mister Contador).

Zonder verder in detail te treden … de manier waarop `double`–waarden intern in de com–puter opgeslagen worden (als *floating point* getallen) zorgt dat dit ongewenste effect zich soms voordoet! Het gegevenstype `decimal` is ongevoelig voor dit euvel!

Het type `decimal` is voornamelijk in het leven geroepen om te werken met *geldbedragen* omdat je daar die afrondingseffecten ten allen tijde wil vermijden. Het presenteert uiteraard niet zo goed om op een kasticket de prijs € 4,1999999999999993 te printen.

Een nadeel is dan weer dat een `decimal`-waarde, intern in de computer, meer geheugen-ruimte in beslag neemt dan een `double`-waarde, maar daar maakt men zich bij het ontwik-kelen van software tegenwoordig minder zorgen om.

Dat Visual Studio erg strikt is bij het werken met verschillende gegevenstypen heeft wel een grote impact als je met het gegevenstype `decimal` aan de slag wil. In onderstaand testje in het venster Immediate loopt het al onmiddellijk fout:

```
Immediate Window
? decimal d1 = 1.4;
Literal of type double cannot be implicitly converted to type 'decimal'; use an 'M' suffix to create a literal of this type
```

Gewoon een `decimal`-variabele declareren en deze daarbij initialiseren op de waarde 1.4 lukt dus al NIET!

De oorzaak is dat een letterlijk kommagetal (hier 1.4) standaard aan het type `double` gekoppeld wordt. Een `double`-waarde opslaan in een `decimal`-variabele, dat wil Visual Studie dus NIET doen in C#!

De foutboodschap in het venster Immediate hierboven geeft ons de tip dat de suffix 'M' van een getal een `decimal` maakt:

```
Immediate Window
? decimal d1 = 1.4M;
1.4
```

Dit 'M'-trucje kan zeker van pas komen, maar wij willen dit conversieprobleem hier toch nog even in een breder kader plaatsen. Eerder in deze cursus kwamen bij de conversie tussen gegevenstypen al volgende onderwerpen aan bod:

- **Getallen converteren naar tekst:** 'Onthoud 1.1' (zie pagina 20) met de `ToString()`.

- **Tekst converteren naar getallen:** 'Onthoud 3.6' en 'Onthoud 3.7' (zie pagina 113) met `Convert.ToInt32()` en `Convert.ToDouble()`.

Dan komt er hier een derde sectie bij, namelijk: **converteren tussen numerieke gegevenstypen**.

Onthoud 4.5 Conversies tussen numerieke gegevenstypen

Om numerieke waarden naar een ander numeriek gegevenstype om te zetten, kan je de **conversie-notatie met ronde haakjes** gebruiken. Je vermeldt dan gewoon vóór de waarde (of de expressie) die je wilt omzetten het **doelgegevenstype** tussen ronde haakjes.

Een voorbeeldje:

```
double d1;
decimal d2;
d1 = 1.4;
d2 = (decimal) d1
```

In de laatste instructie laten we de `double`-variabele `d1` - via de ronde haken notatie - converteren naar het type `decimal`.

Hieronder nog enkele extra voorbeelden in het venster Immediate met conversies tussen de numerieke gegevenstypen `int`, `double` en `decimal`.

- Een letterlijke kommawaarde omzetten naar een `decimal`:

```
Immediate Window
? decimal d = (decimal) 1.4;
1.4
```

Merk op dat letterlijke kommagetallen (hier 1.4) altijd aanzien worden als `double`. Met de vermelding "`(decimal)`" lieten wij deze `double` omzetten naar een `decimal`.

"`(decimal) 1.4`" is dus een geldig alternatief voor de notatie "`1.4M`" die hierboven al even aan bod kwam.

- Een kommawaarde omzetten naar een `int`:

```
Immediate Window
? double bmi1 = 83 / (1.76 * 1.76);
26.794938016528928
? int bmi2 = 83 / (1.76 * 1.76);
Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
? int bmi3 = (int) (83 / (1.76 * 1.76));
26
```

De foutboodschap na de declaratie en initialisatie van de variabele `bmi2` toont dat je geen kommagetal kan laten opslaan in een `int`-variabele.

Met de extra vermelding "`(int)`" lieten we diezelfde berekening daarentegen omzetten naar een geheel getal (en succesvol opslaan in de variabele `bmi3`). Alle cijfers na de

komma worden bij deze conversie weggewerkt. Merk dus op dat met "`(int)`" het kommagetal altijd naar beneden afgerond wordt.

❖ Numerieke gegevenstypen en delingen:

Twee getallen delen door elkaar kan - afhankelijk van het type van deze getallen - verschillende resultaten opleveren.

Bijvoorbeeld in het venster Immediate:

```
Immediate Window
? 10 / 3
3
? 10 / 3.0
3.3333333333333335
? 10 / (double) 3
3.3333333333333335
? 10 / (decimal) 3
3.3333333333333333333333333333
```

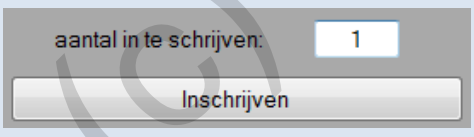
Bovenstaande vier delingen leverden respectievelijk een resultaat op van het type: `int`, `double`, `double` en `decimal`.

Voor de volledigheid vermelden we nog even dat je met `ToString()` een `decimal`-waarde kan laten omzetten naar een tekst.

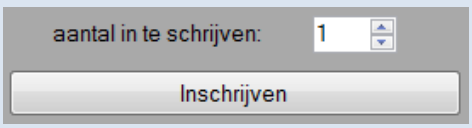
De omgekeerde bewerking, of een `String` converteren naar het type `decimal`, kan met `Convert.ToDecimal()` (cfr. `Convert.ToInt32()` en `Convert.ToDouble()`).

Oefening 4.7 LeerlingenSecretariaat – In het project `LeerlingenSecretariaat` (**_hoofdstuk_4/_oefeningen**) vind je ons werkende formulier `VrijeStudieForm` terug waarmee men op het leerlingensecretariaat de inschrijvingen voor de vrije studie kan opvolgen (zie hoofdstuk 3.4 vanaf pagina 106).

In het tekstvak `aantalInschrijvenTextBox` geeft de secretariaatsmedewerker op hoeveel leerlingen zich willen inschrijven:



Ondertussen hebben we uit de doeken gedaan dat het veel verstandiger is om dit aantal te laten invoeren via een `NumericUpDown`-element:



We vatten de koe bij de horens:

❖ Verwijder het tekstvak `aantalInschrijvenTextBox` op het formulier en sleep er vanuit de ToolBox een `NumericUpDown`-element in de plaats. Noem dit element `aantalInschrijvenNumericUpDown`.

In het Properties venster stel je via de betreffende objecteigenschappen in dat dit element enkel gehele getallen van 1 tot 40 mag bevatten. Zorg dat de `NumericUpDown` initieel (bij het openen van het formulier) op 1 staat.

We zullen ook nog een beetje moeten sleutelen aan de programmacode (bij de methode die reageert op de gebeurtenis `Click` van de 'Inschrijven'-knop):

❖ Dit is de *oude* instructie waarmee we de inhoud opvroegen aan het *oude* tekstvak:

```
// nagaan hoeveel leerlingen willen inschrijven
int aantal = Convert.ToInt32(aantalInschrijvenTextBox.Text);
```

We hebben een *nieuwe* instructie nodig waarmee we de variabele `int aantal` laten initialiseren op de waarde in ons *nieuwe* `NumericUpDown`-element.

Tip:

🔔 Als je de inhoud van een `NumericUpDown`-element opvraagt, krijg je een resultaat van het type `decimal`. Om deze waarde op te slaan in een `int`-variabele zal ook hier een cast noodzakelijk zijn.

Omdat het deze keer over een cast van `decimal` naar `int` gaat (dus een conversie tussen numerieke gegevenstypen), zou je de kortere ronde haken-notatie kunnen gebruiken.

❖ Achteraan bij de methode vind je de *oude* instructie om, na het verwerken van de inschrijving, het *oude* tekstvak weer leeg te maken:

```
// tekstvak met aantal in te schrijven leerlingen leegmaken
aantalInschrijvenTextBox.Text = "";
```

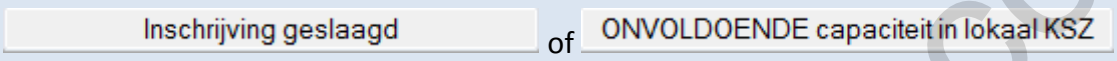
Dit zouden we hier kunnen vervangen door een *nieuwe* instructie waarmee we het *nieuwe* `NumericUpDown`-element opnieuw instellen op zijn beginwaarde 1.

4.4 Even oefenen

Omdat er ondertussen in hoofdstuk 4 al weer veel nieuwe leerstof de revue gepasseerd is, leggen we jullie graag nog enkele (grotere) programmeeroefeningen voor.

Oefening 4.8 LeerlingenSecretariaat – We keren nog eens terug naar het project `LeerlingenSecretariaat` (**_hoofdstuk_4/_oefeningen**). We werken hier verder met de toepassing die we in hoofdstuk 3.4 (zie vanaf pagina 106) opgebouwd hebben.

Een secretariaatsmedewerker laat via het formulier `VrijeStudieForm` een gegeven aantal leerlingen inschrijven voor een vrije studie na de schooluren. Afhankelijk of er al dan niet nog voldoende plaatsen beschikbaar zijn, presenteert het formulier ons in het tekstvak `boodschapTextBox` één van onderstaande meldingen:



We willen deze boodschappen nog wat extra benadrukken!

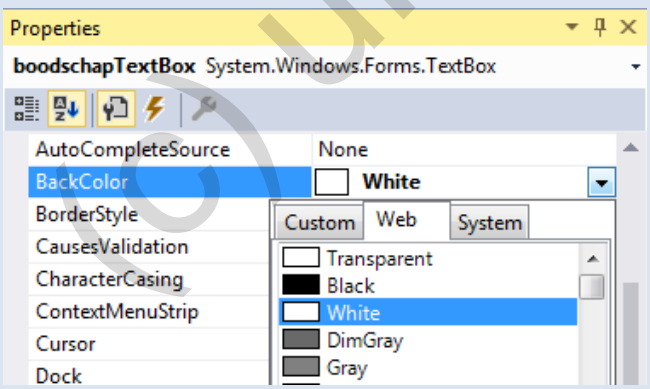
Kleurtjes doen het in deze context altijd goed. Als de inschrijving geslaagd is, zouden we het tekstvak *groen* willen kleuren; een *rood* tekstvak bij een fail, lijkt ons ook niet onlogisch.

Je kan je er met slechts twee extra lijntjes programmacode vanaf maken. Ga hiervoor aan slag met de onderstaande achtergrondinformatie.

Je neemt sowieso best alle uitleg hieronder eens grondig door. Zo ben je voldoende gewapend om in de toekomst alle *kleuropdrachten* met succes aan te pakken!

<u>In het Properties venster</u>

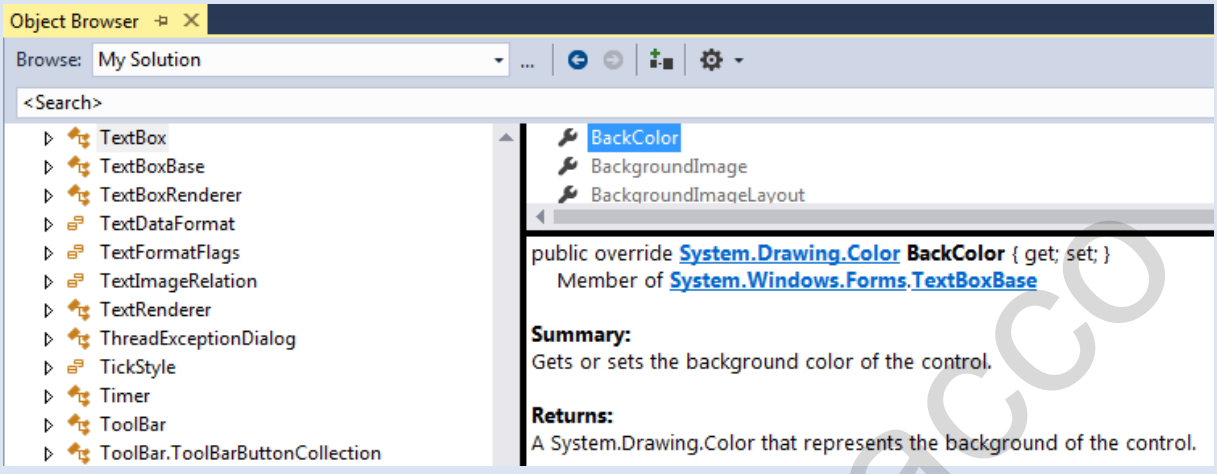
De objecteigenschap in het Properties venster waarmee je de kleur van een tekstvak (`TextBox`) instelt, is `BackColor`.



O.a. via het tabblad **Web** bij het opengeklapte menu, krijg je een heel ruime lijst met vooraf gedefinieerde kleurnamen.

In de Object Browser

Uiteraard kan je in de Object Browser bij het type `TextBox` (namespace `System.Windows.Forms`) diezelfde objecteigenschap `BackColor` terugvinden:



De header, in het venstertje rechts onderaan, vertelt dat je de objecteigenschap `BackColor` bij een tekstvak kan **opvragen** en **wijzigen** (want `get` en `set`). Je leest er ook dat deze objecteigenschap `BackColor` van het type `System.Drawing.Color` is. `System.Drawing.Color` bestaat eigenlijk uit twee componenten:

- `System.Drawing` is de **namespace**;
- `Color` is het eigenlijke **type**.

In je programmacode

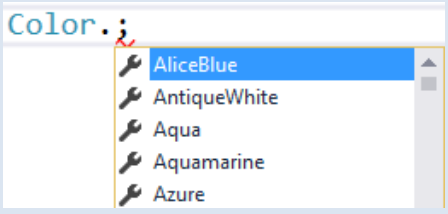
True en false zijn de enige `bool`-waarden; 1, 5 en 99 zijn voorbeelden van `int`-waarden; "hallo wereld" is een `String`; 'x' is een `char`; … Maar hoe moet je nu in je programmacode letterlijke `Color`-waarden voorstellen? Hoe geef je m.a.w. het kleur 'rood' en 'groen' aan?

Simpel, met de expressie `Color.`*`Kleurnaam`* stel je het betreffende kleur voor!

De 'kleurnamen' die je hier mag gebruiken, zijn dezelfde als deze die in het Properties venster opgelijst worden bij een objecteigenschap van dit type. Even speuren in zo'n lijstje en je ontdekt bijvoorbeeld dat onderstaande drie `Color`-waarden rood-varianten zijn:

- `Color.Red;`
- `Color.Tomato;`
- `Color.Maroon.`

Als je in je code ergens "`Color.`" intikt, kan je trouwens zó uit het lijstje een kleur selecteren:



Oefening 4.9 **Wielertenue -** Bij een reclame-actie van een sportkrant kunnen de abonnees een fietstenue bestellen. Men kan hierbij kiezen om enkel een fietstruitje te kopen of opteren om er ook een wielerbroek bij te nemen.

Een opbergzakje in het truitje voor de gsm of een duurder zeem in de broek zijn nog twee upgrades, die het tenue iets duurder maken. Bestel je een grotere hoeveelheid, dan mag je rekenen op een korting.

Een programmaatje om snel de prijs te berekenen zou dus heel gewenst zijn.

Het eindresultaat zou er als volgt kunnen uitzien:

De Business Layer

Met de klasse `BestellingWielertenue` zullen we de bestellingen van wieler–truien/wielertenues beschrijven. De velden en de constructor hebben we reeds in deze klasse geplaatst.

Wat doen jullie nog:

1. Properties

   Voorzie voor elk van de vier velden een Property in de klasse. Voeg telkens een get- en set-statement toe.

2. Methode `decimal GeefPrijs()`

   Deze methode berekent de prijs van de bestelling. Laat ons afspreken dat we voor geldbedragen voortaan het gegevenstype `decimal` gebruiken, al zal je dan steeds weer rekening moeten houden met heel wat mogelijke 'conversies'.

   Hoe bereken je de prijs:

   De wielertrui kost € 45,00. Neem je de volledige tenue (je neemt er dus de broek bij), dan kom je aan € 80,00.

Wil je de wielertrui met een waterdicht zakje (waar een gsm in past) dan kost je dat € 3,50 extra. Bij de broek kan je voor een schamele € 5,50 een verbeterd zeem nemen (superzeem genoemd).

Bij een bestelling houden we ook het aantal stuks bij. Wie 5 of meer stuks ineens bestelt, krijgt een korting van 10%.

De Presentation Layer

De afbeeldingen (`PictureBox`) hebben wij al op het formulier `BestellingForm` gezet. Opgelet: het gaat hier over **twee** afbeeldingen. Omdat beide afbeeldingen 'boven' elkaar staan, zie je enkel `wielertenuePictureBox`. Daarachter staat de afbeelding `wielertruiPictureBox` (afbeelding met enkel een wielertruitje).

De NumericUpdown (`aantalNumericUpDown`), de drie selectievakjes (`waterdicht-ZakjeCheckBox`, `broekCheckBox` en `superzeemCheckBox`) en het tekstvak (`prijsTextBox`) zetten jullie er zelf bij.

Bij `prijsTextBox` stel je de objecteigenschap `ReadOnly` in op **true**. Hierdoor kan de gebruiker straks zelf niets meer in het tekstvak met de prijs intikken.

1. Om de Presentation en Business Layer aan elkaar te koppelen, laten jullie binnen de klasse `BestellingForm` (presentatielaag) een object van de klasse `Bestelling-Wielertenue` (= businesslaag) declareren. Noem dit veld `_bestelling`.

   In de constructor laat je dit object `_bestelling` initialiseren op een bestelling van één wielertruitje (geen wielerbroek en bijgevolg ook geen superzeem), zonder waterdicht zakje.

2. **Niet via programmacode** - maar door de juiste objecteigenschappen in te stellen in het Properties venster - zorg je dan ook dat het formulier er initieel als volgt uitziet:

   We tonen (enkel) de afbeelding met het truitje; het selectievakje met het superzeem laat je verbergen! Dit laatste selectievakje is - wegens geen broek - toch niet van toepassing.

   De weergave van het formulier komt zo helemaal overeen met het businessobject `_bestelling` zoals we dit lieten initialiseren bij de constructor!

3. Bij het openen van het formulier willen we in het tekstvak onderaan wel al onmiddellijk de juiste prijs tonen. Door het correcte lijntje code toe te voegen aan de constructor is dit zo geregeld.

Elke keer als we in het formulier iets wijzigen aan de bestelling, zal de prijs vernieuwd moeten worden. Concreet zullen we dit doen door hierbij telkens ons businessobject `_bestelling` bij te werken, om daarna aan `_bestelling` de nieuwe prijs te vragen (en te vernieuwen in het tekstvak).

Er zal hiervoor in dit formulier bij heel wat diverse gebeurtenissen code moeten geactiveerd worden. We sommen ze allemaal op.

4. Methode die reageert op de gebeurtenis `ValueChanged` van `aantalNumericUpDown`

   `ValueChanged` is de gebeurtenis die geactiveerd wordt telkens als je in een `NumericUpDown`-element een andere waarde selecteert.

   Als de methode bij deze gebeurtenis opgeroepen wordt, dan weet je dat de gebruiker het 'aantal stuks' bijgewerkt heeft.

   In deze methode laat je bijgevolg het 'aantal stuks' bijwerken bij ons businessobject `_bestelling` en laten we in het tekstvak `prijsTextBox` de nieuwe prijs verschijnen.

5. Methode die reageert op de gebeurtenis `CheckedChanged` van `waterdichtZakjeCheckBox`

   Afhankelijk of het vinkje in- of uitgeschakeld werd, laat je bij het businessobject `_bestelling` al dan niet het waterdicht zakje opnemen.

   Laat daarna in het tekstvak `prijsTextBox` de nieuwe prijs tonen.

6. Methode die reageert op de gebeurtenis `CheckedChanged` van `broekCheckBox`

   Je snapt dat je ook hier bij het aan- of afvinken van de wielerbroek het businessobject `_bestelling` moet bijwerken en de nieuwe prijs moet tonen.

   Bij dit selectievakje hebben we echter wat extra 'presentatie'-werk:

   - Afhankelijk of de klant al dan niet de wielerbroek mee bestelt, laten we de `PictureBox` met de volledige wielertenue of de `PictureBox` met enkel het truitje tonen.
   - Als je de wielerbroek neemt, moet je de gebruiker de mogelijkheid geven om voor het superzeem te kiezen. Het selectievakje `superzeemCheckBox` moet dan dus zichtbaar zijn! Als de gebruiker geen wielerbroek neemt, heeft het selectievakje `superzeemCheckBox` geen betekenis en hebben we dit selectievakje liever weg.

   Een mogelijke bug in je formulier is dat een klant een superzeem aangerekend kan krijgen, terwijl hij/zij GEEN broek neemt!

7. Methode die reageert op de gebeurtenis `CheckedChanged` van `superzeemCheckBox`

   Zorg tenslotte dat ook bij het aan- of afvinken van `superzeemCheckBox` de prijs correct ge-updatet wordt.

Ten slotte nog even checken of jullie er ook aan gedacht hebben om de prijs van de bestelling in het tekstvak met een Valuta-notatie weer te geven:

4.5 De Object Browser nader bekeken

De Object Browser kwam in dit hoofdstuk al even aan bod. Als we extra informatie over één of andere objecteigenschap nodig hebben, bewijst dit venster zeker zijn nut. Maar hiermee alleen doen we de Object Browser nog veel tekort.

Daarom dat we hier per se nog een ode wilden brengen aan de Object Browser.

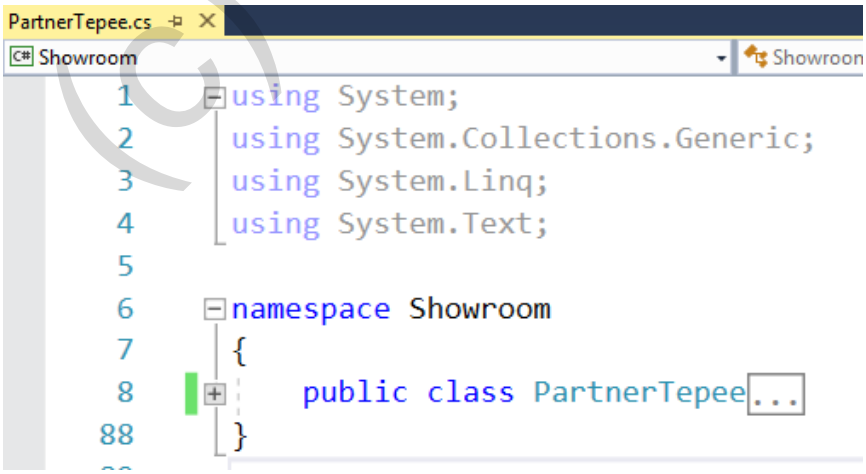
Echt álles wat je tot nu toe geleerd hebt over objectgeoriënteerd programmeren, te rekenen vanaf jullie eerste les BlueJ, laat sporen na in de Object Browser.

4.5.1 Properties in de Object Browser

Het linker venster van de Object Browser bevat de namespaces die in Visual Studio gekend zijn.

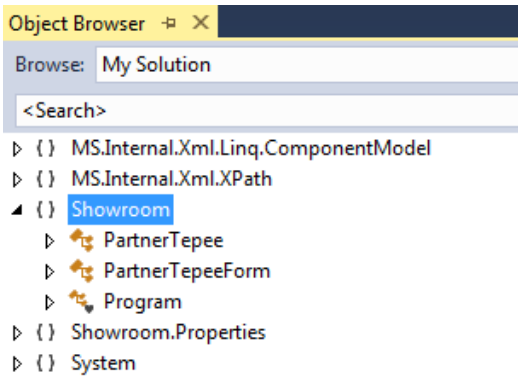
Zo weet je ondertussen dat in de namespace `System.Windows.Forms` alles verzameld wordt wat te maken heeft met Windows Formulieren. Denk hierbij voornamelijk aan de vele 'dingen' die je op deze formulieren kan slepen.

Herinner je je nog dat de klassen die we zelf programmeren ook binnen een namespace staan? Kijk in het bronproject `Showroom` maar eens naar de code van de businessklasse `PartnerTepee`:



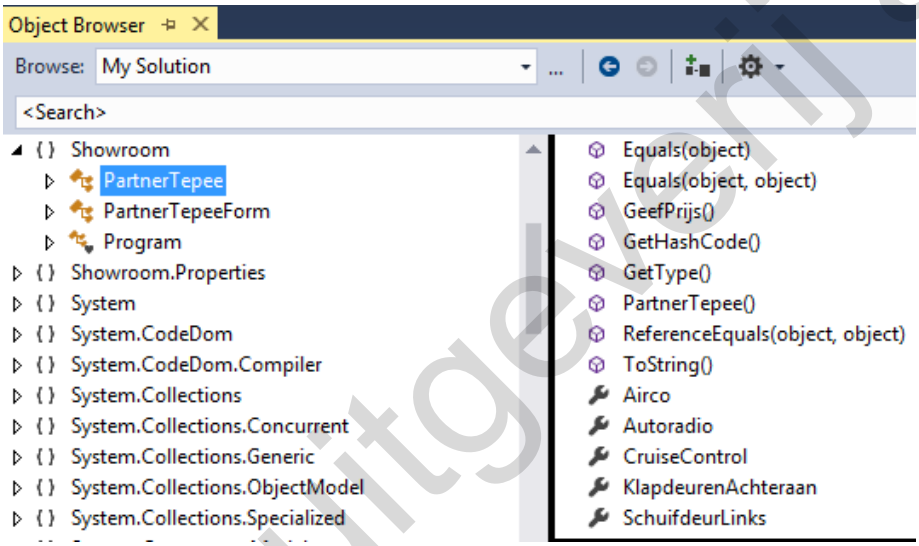
De klasse `PartnerTepee` hoort dus bij de namespace `Showroom`.

Als je de Object Browser opent in het project `Showroom` zal je links daarom ook die namespace `Showroom` terugvinden! Opengeklapt zie je onder de namespace `Showroom` inderdaad onze businessklasse `PartnerTepee` staan.



Hetzelfde geldt voor de presentatieklasse `PartnerTepeeForm`.

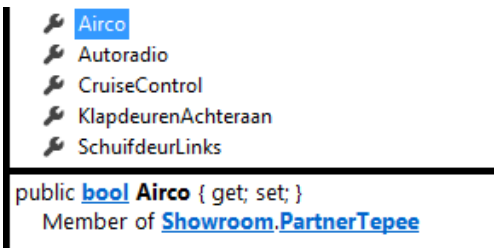
Het wordt nog veel interessanter voor de klasse `PartnerTepee`:



Nog niet alles wat in het venster rechts prijkt, kunnen we duiden, maar je zou wel de constructor `PartnerTepee()`, de methode `GeefPrijs()` en onze vijf Properties `Airco`, `Autoradio`, `CruiseControl`, … moeten herkennen.

Fijn dat we in de Object Browser dus ook onze eigen klasse kunnen inspecteren!

Zelfs het venster rechts onderaan in de Object Browser werkt mee. Je leest er bijvoorbeeld dat de Property `Airco` (selecteer deze Property) over een get- en set-component beschikt en dat het type van deze Property `bool` is:



We herkennen in deze korte beschrijving in de Object Browser inderdaad mooi de eigenschap `Airco` zoals wij die programmeerden in de klasse `PartnerTepee`:

```
public bool Airco
{
    get { return _airco; }
    set { _airco = value; }
}
```

De **Properties** van de klasse `PartnerTepee` worden in de Object Browser allen aangeduid met het symbool .

In het eerste deel van dit hoofdstuk hadden we het over **objecteigenschappen**. Denk aan `Visible` bij een `PictureBox`, `Checked` bij een `CheckBox`, `BackColor` bij een `TextBox`, … Ook deze objecteigenschappen werden gelabeld met het -icoontje.

Dit laat ons toe om volgende gewichtige conclusie te trekken:

De begrippen **objecteigenschap** en **Property** zijn gewoon **synoniemen** van elkaar! Net zoals `Airco` een Property is bij de klasse `PartnerTepee`, mag je ontegensprekelijk stellen dat `Visible` een Property is bij de klasse `PictureBox`.

Het verschil is gewoon dat `PartnerTepee` een klasse is die wij zelf programmeerden (wij definieerden zelf de Property `Airco`), terwijl `PictureBox`, met o.a. de Property `Visible`, een klasse is die door Visual Studio aangeleverd wordt. Een ander verschil is dat je geen mogelijkheid hebt om de Properties uit je eigen klassen op te roepen in het Properties venster.

In de Object Browser daarentegen hebben `Airco` en `Visible` in hun respectievelijke klasse **precies dezelfde** status!

Onthoud 4.6 Selecteer je in de **Object Browser** in het linker venster een klasse, worden rechts alle **Properties** bij deze klasse opgesomd. De Properties hebben een -icoontje.

Omdat `Airco` en `Visible` beiden Properties van het type `bool` zijn, gebruik je deze in je code ook op eenzelfde manier. Dat blijkt duidelijk uit onderstaand 'oud' codefragment bij de klasse `PartnerTepeeForm`:

```
private void autoradioCheckBox_CheckedChanged(object sender, EventArgs e)
{

    if (autoradioCheckBox.Checked == true)
    {
        autoradioPictureBox.Visible = true;
        _wagen.Autoradio = true;
    }
    else
    {
        autoradioPictureBox.Visible = false;
        _wagen.Autoradio = false;
    }

    prijsTextBox.Text = _wagen.Prijs().ToString("C");
}
```

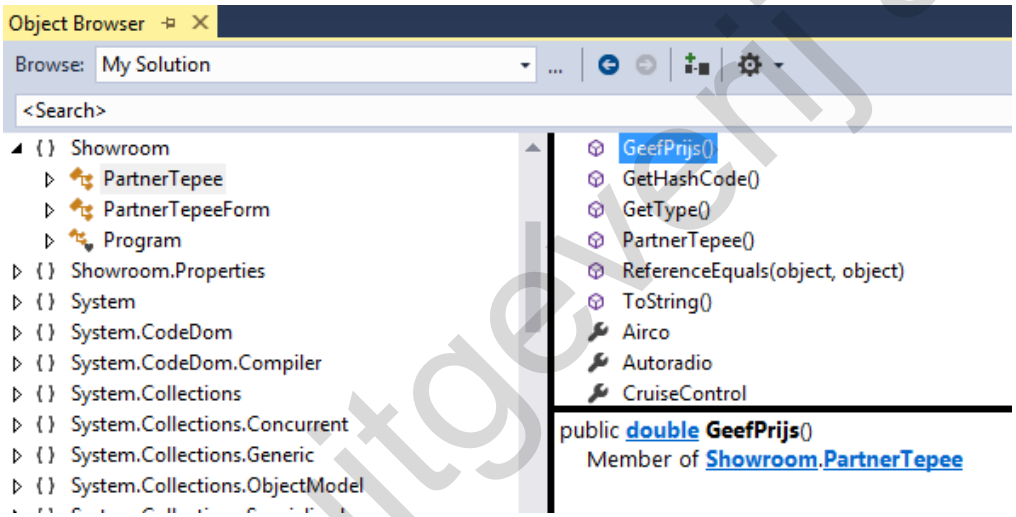
Je leest hieronder dan ook twee keer dezelfde beschrijving voor de instructies in het vet:

- Met de instructie "`autoradioPictureBox.Visible = true`" zetten we de Property `Visible` van het object `autoradioPictureBox` op true. `autoradioPictureBox` is hierbij een object van de klasse `PictureBox`.
- Met de instructie "`_wagen.Autoradio = true`" zetten we de Property `Autoradio` van het (business)object `_wagen` op true. `_wagen` is hierbij een object van de klasse `PartnerTepee`.

In bovenstaand codefragment zie je trouwens ook nog de objecteigenschappen (dus Properties) `Checked` en `Text` aan het werk.

4.5.2 Methoden in de Object Browser

Bij onze klasse `PartnerTepee` konden we in de Object Browser ook de **methode** `GeefPrijs()` aanduiden:



In het hulpvenster rechts onderaan staan ook deze keer geen leugens. Sla er de code van de methode `GeefPrijs()` in **PartnerTepee.cs** maar op na:

- deze methode neemt geen parameters:

```
public double GeefPrijs() ;
```

- deze methode heeft een retourwaarde van het type `double`:

```
public double GeefPrijs()
```

(al zouden we deze methode nu eerder een `decimal` laten retourneren).

De methode `GeefPrijs()` wordt in de Object Browser gelabeld met het symbool . We veralgemenen dit als volgt:

> **Onthoud 4.7** Als je in de **Object Browser** in het linker venster een klasse selecteert, worden rechts alle **methoden** bij deze klasse opgesomd. De methoden herken je aan het -icoontje.

Opmerking:

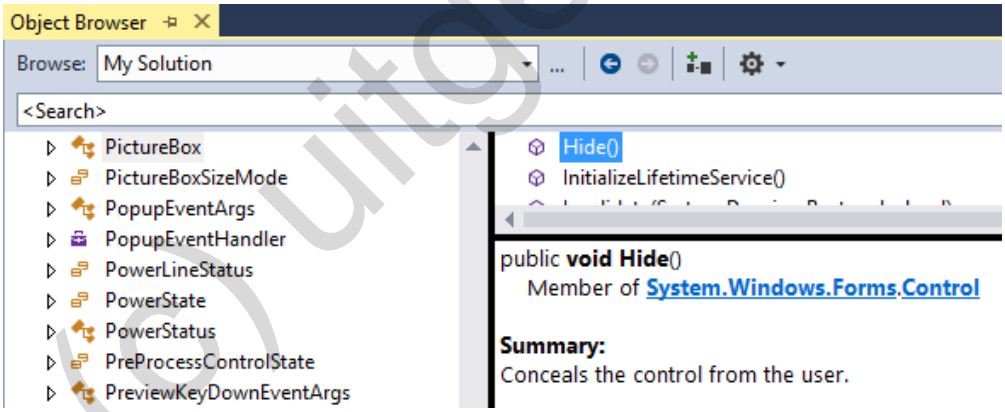
- Waar in de Object Browser dan plots al die andere methoden bij de klasse `Partner-Tepee` vandaan komen, zoals: `Equals(object)`, `GetHashCode()`, `GetType()`, … blijft nog even een raadsel! Wij hebben die methoden alvast niet zelf geprogrammeerd.

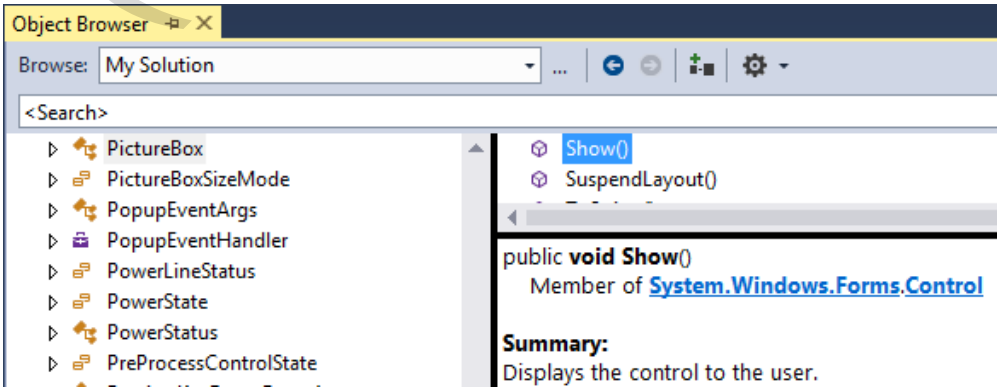
  Verder in de cursus zullen we ook dit mysterie ontrafelen.

Nu we weten hoe we in de Object Browser **methoden** kunnen herkennen (via het -icoon), vragen we ons af of we bij de klassen uit de namespace `System.Windows.Forms` mis–schien ook methoden kunnen terugvinden!

Als een `PictureBox`, een `CheckBox`, een `TextBox`, … over methoden beschikt, dan kunnen we die ook in onze programmacode aanwenden. Dit zou ons scala aan mogelijke programmeerinstructies alleen maar groter maken.

Even de klasse `PictureBox` (namespace `System.Windows.Forms`) onder de loep nemen in de Object Browser levert ons bijvoorbeeld volgende twee methoden op:





Het gaat hier twee keer over:

- een methode die geen parameters neemt:
  public **void Hide**() en public **void Show**()
- een methode die geen retourwaarde heeft (want retourtype is void):
  public **void Hide**() en public **void Show**()

Nadat we "to conceal" even door de Google-translate lieten draaien, konden we voor de twee 'Summary'-helptekstjes (venster rechts onderaan) volgende vertaling maken:

- public **void Hide**() → Verbergt het besturingselement voor de gebruiker.
- public **void Show**() → Toont het besturingselement aan de gebruiker.

In onze `Showroom`-toepassing lieten we enkele `PictureBox`-elementen tonen/verbergen door de objecteigenschap (=Property) `Visible` te manipuleren. Het blijkt nu dat we het–zelfde kunnen regelen via de methoden `Hide()` en `Show()`! De Object Browser vertelt ons alvast dat we deze methoden mogen toepassen op objecten van het type `PictureBox`.

We keren even terug naar het stukje code dat reageert als de gebruiker in `PartnerTepee-Form` het selectievakje `autoradioCheckBox` aan- of uitvinkt (en waarmee we o.a. de bijhorende afbeelding laten tonen/verbergen).

We vervangen er de doorstreepte instructies als volgt:

```
private void autoradioCheckBox_CheckedChanged(object sender, EventArgs e)
{
    if (autoradioCheckBox.Checked == true)
    {
        ~~autoradioPictureBox.Visible = true;~~
        autoradioPictureBox.Show();
        _wagen.Autoradio = true;
    }
    else
    {
        ~~autoradioPictureBox.Visible = false;~~
        autoradioPictureBox.Hide();
        _wagen.Autoradio = false;
    }

    prijsTextBox.Text = _wagen.Prijs().ToString("C");
}
```

Test het maar uit, de afbeelding `autoradioPictureBox` zal nog steeds pas zichtbaar gemaakt worden op het moment dat men het selectievakje `autoradioCheckBox` aanvinkt! De methoden `Show()` en `Hide()` bij het type `PictureBox` zijn m.a.w. een perfect alternatief voor de Property `Visible`.

De overvloed aan methoden die je overal in de Object Browser terugvindt, maakt na de ingrijpende introductie van de objecteigenschappen onze kijk op de OO-programmeerwereld in Visual Studio nog eens explosief breder!

We maken hierbij nog twee bemerkingen:

- De objecteigenschap `Visible` hadden we eerst al in het **Properties venster** geïdentificeerd vooraleer we deze Property nader gingen bekijken in de Object Browser.

  Methoden worden echter per definitie NOOIT in het Properties venster vermeld.

  Wil je - om met een nieuw formulierelement te leren werken – de mogelijke methoden verkennen, dan ben je dus sowieso aangewezen op de Object Browser!

- Omdat de methode `Show()` (alsook `Hide()`) geen parameters neemt en er geen retourwaarde is, zag de instructie waarmee we deze methode opriepen, er wel heel eenvoudig uit: "`autoradioPictureBox.Show();`".

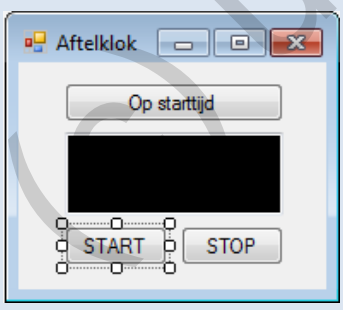
  We zullen in de Object Browser zeker nog botsen op methoden mét (meerdere) parameters die wel in een retourwaarde resulteren. Het zal dus niet altijd zó simpel zijn.

Oefening 4.10 Aftelklok - In hoofdstuk 2 lieten we jullie bij 'Oefening 2.4' (zie pagina 60) een aftelklok programmeren. Bij de brongegevens (zie map **_hoofdstuk_4/_oefeningen**) vind je in het project `Aftelklok` een modeloplossing bij die opgave.

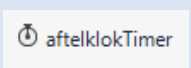
Start je het project op, dan verschijnt een formulier met een klokje op 1:30 dat **meteen** begint af te tellen. Met de knop 'Op starttijd' zetten we de klok opnieuw op 1:30 die daarna **onmiddellijk** weer gaat aftellen.

Dat we geen mogelijkheid hebben om het aftellen te stoppen (en dan opnieuw te starten), is toch een minpunt van onze toepassing. Omdat we al lang niet meer de 'bleutjes' zijn, die we nog waren in hoofdstuk 2, pakken we het probleem hier aan.

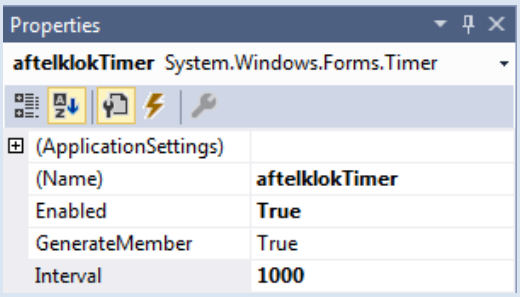
❖ In de weergave Design plaats je hiervoor een START- en een STOP-button op het formulier:



Weet je nog dat het aftellen aangestuurd wordt door een `Timer`? In de weergave Design zie je onder het formulier (in het componentenvak) die `Timer` met de naam `aftelklokTimer` staan:



In het Properties venster hadden we één en ander in te stellen bij dit `Timer`-object:



De Property `Interval` bepaalt hoe snel `aftelklokTimer` tikt. De 1000 staat hier voor "per 1000 milliseconden" of dus "per seconde".

De Property `Enabled` bepaalt of `aftelklokTimer` al dan niet ingeschakeld is. Door deze objecteigenschap op true in te stellen, begint de `Timer` bij het openen van het formulier onmiddellijk te tikken.

In dit hoofdstuk leerde je dat de Properties uit het Properties venster ook in de Object Browser te vinden zijn. Even checken …

❖ Zoek in de Object Browser bij het type `Timer` (namespace `System.Windows.Forms`) de Property `Enabled` op.

- Is `Enabled` een Read only Propery? Hoe zie je dit?

- Welke beschrijving wordt bij de Summary van de Property `Enabled` gegeven?

❖ Zet in het Properties venster de Property `Enabled` van de `Timer` `aftelklokTimer` op false.

Bij het openen van het formulier zal `aftelklokTimer` nu niet meer ingeschakeld zijn. Gevolg: de klok loopt nu standaard niet meer. Controleer maar.

Je beschikt nu over alle informatie om de nieuwe START- en STOP-button de nodige functionaliteit te geven. De clou is natuurlijk dat we het aftellen van het klokje aan- en uitzetten door de Property `Enabled` van onze `Timer` `aftelklokTimer` te manipuleren.

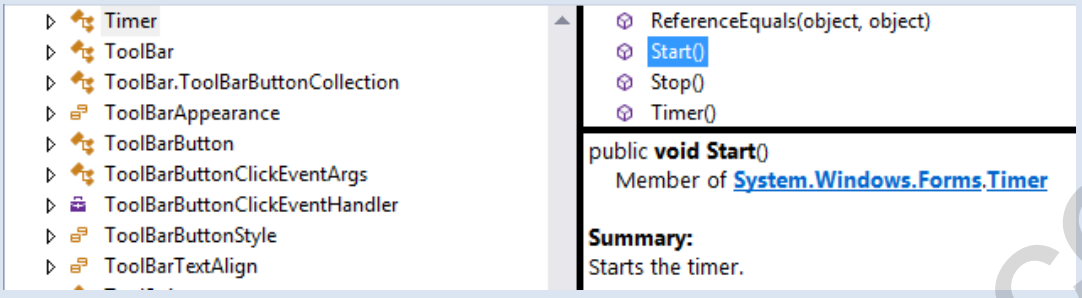
❖ Klikken op de START-knop zou de klok moeten laten lopen. Schrijf bij het klikken op deze knop code om de `Timer` `aftelklokTimer` via de Property `Enabled` in te schakelen.

❖ Klikken op de STOP-knop zou de aftelklok moeten stilzetten. Laat bij het klikken op deze knop bijgevolg de `Timer` `aftelklokTimer` weer uitschakelen.

❖ Tenslotte zorg je nog dat met het knopje 'Op Starttijd', nadat de tijd weer op 1:30 gezet werd, het aftellen ook onderbroken wordt (tot iemand weer op START klikt).

Als het formulier `AftelklokForm` getest en goed bevonden is, willen we jullie ook nog even op een alternatieve oplossingsmethode wijzen.

In de Object Browser bij de klasse `Timer` (namespace `System.Windows.Forms`), zien we dat deze klasse ook de methoden `void Start()` en `void Stop()` bevat.



De Summary bij deze methoden maakt veel duidelijk.

❖ Pas in `AftelklokForm` de code aan, zodat we geen beroep meer doen op `Enabled` om onze timer aan te sturen, maar dat we dit nu regelen met de geduide methoden.

(*Noot van de auteur*) Jullie krijgen het principe van het werken met de Object Browser stilletjesaan onder de knie. Voor *mij persoonlijk* gingen door die Object Browser plots veel lampen branden, vielen de puzzelstukken in elkaar, werd plots het groter plaatje zichtbaar …

De theorie van objectgeoriënteerd programmeren kan je in enkele vuistregels vatten. We spekken dit met concrete voorbeeldjes uit onze Showroom-toepassing en verwijzen volop naar de Object Browser:

❖ In het linker venster van de Object Browser vind je de **klassen** terug. Dit kunnen klassen zijn die jij zelf geprogrammeerd hebt (`PartnerTepee`). Vele klassen worden door Visual Studio aangeleverd. Zo hadden wij in ons project `Showroom` te maken met de klassen `PictureBox`, `CheckBox` en `TextBox`.

❖ Van een klasse kan je **instanties** (=**objecten**) maken. Dit kunnen objecten zijn die je zelf declareert en initialiseert. `_wagen` is zo'n instantie van de klasse `PartnerTepee`. De afzonderlijke elementen die je op een formulier sleept, zijn even goed objecten. Noem maar op: `autoradioPictureBox` is een instantie van de klasse `PictureBox`, `prijsTextBox` is een instantie van de klasse `TextBox`.

❖ In het rechter venster worden de **Properties** en **methoden** bij een klasse opgelijst. Deze Properties en methoden kan je toepassen op de instanties van die klasse. We maken hierbij weer geen enkel onderscheid tussen de objecten van je eigen klassen en de ob–jecten die op het formulier staan.

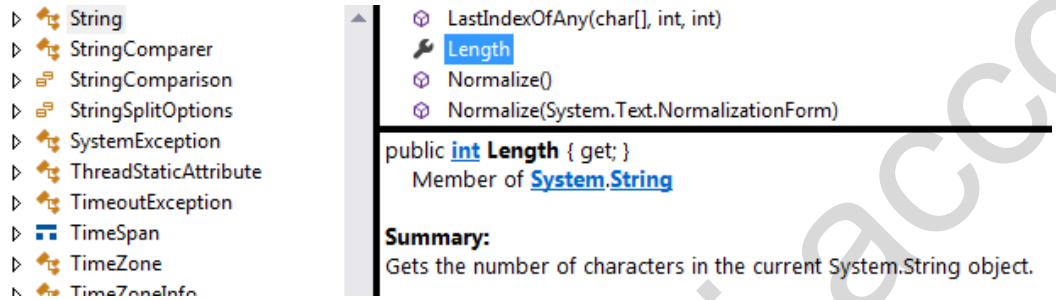
  Net zoals we voor de instantie `_wagen` (klasse `PartnerTepee`) de Property `Autoradio` kunnen opvragen of de methode `GeefPrijs()` kunnen oproepen, hebben we bij de

instantie `autoradioPictureBox` (klasse `PictureBox`) de Property `Visible` en de methoden `Show()` en `Hide()` beschikbaar.

Ook een laatste, interessant voorbeeld past helemaal in deze theorie.

In de Object Browser zal je in het linker venster, onder de namespace `System`, het type `String` terugvinden. In het rechter venster bulkt het hierbij van de methoden, maar kan je ook een enkele Property opsporen.

❖ Property `Length`



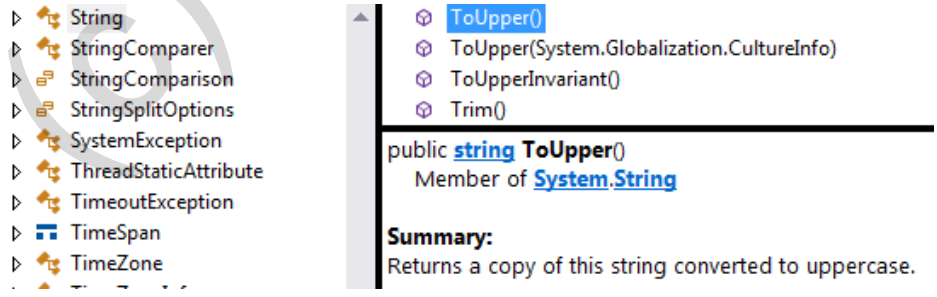
Het type `String` bezit een Read only Property `Length` van het type `int` (Read only, want enkel een get). De Summary beschrijft dat deze Property het aantal karakters in het object telt. We hebben m.a.w. gevonden hoe je **de lengte** van een `String` kan opvragen.

In het venster Immediate:

```
Immediate Window
? String tekstje = "Hallo wereld!";
? tekstje.Length
13
```

`tekstje` is in bovenstaand voorbeeldje een instantie van het type `String`, waaraan we de Property `Length` opvragen.

❖ Methode `ToUpper()`



`ToUpper()` is een methode die GEEN parameters neemt en een waarde retourneert van het type `String`. De Summary zegt dat deze retourwaarde een kopie is van onze `String`, waarbij alles in hoofdletters gezet werd.

In het venster Immediate:

```
Immediate Window
? String tekstje = "Hallo wereld!";
? tekstje.ToUpper()
"HALLO WERELD!"
```

Oefening 4.11 In bovenstaande voorbeelden in het venster Immediate, waar we met de variabele `tekstje` aan de slag gingen, werden na `Length` geen ronde haken getypt, terwijl dit bij `ToUpper()` wel het geval was. Waarom is dit?

Oefening 4.12 **ProjectZonderNaam -** Met deze oefening willen we jullie nog even kennis laten maken met enkele andere `String`-methoden. Misschien zal je je nog één en ander herinneren uit de cursus BlueJ, want ook Java heeft zijn eigen `String`-bewerkingen.

In het project `ProjectZonderNaam` hebben wij de businessklasse `Naam` gedefinieerd. Een `Naam`-object bestaat uit een `String`-veld `_voornaam` en een `String`-veld `_familienaam`. Naast de triviale Properties hebben wij al een methode `VolledigeNaam()` in deze klasse gestopt, die – op basis van een `bool`-parameter – de voor- en familienaam aan elkaar laat plakken.

```
Immediate Window
? Naam naam = new Naam("Muhammed", "Lee");
{ProjectZonderNaam.Naam}
    _familienaam: "Lee"
    _voornaam: "Muhammed"
    Familienaam: "Lee"
    Voornaam: "Muhammed"
? naam.VolledigeNaam(true)
"Lee Muhammed"
```

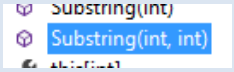
methode `String Initialen()`

Breid de klasse `Naam` uit met de methode `Initialen()`. De retourwaarde van deze methode is de eerste letter van de voornaam, een puntje, de eerste letter van de familienaam en nog een puntje.

```
Immediate Window
? Naam naam = new Naam("Muhammed", "Lee");
? naam.Initialen()
"M.L."
```

Als tip horen we jullie eerst even uit over de methode `Substring()`. Met deze methode kan je uit een gegeven `String` immers een deelstring halen.

Vraag in de Object Browser bij het type `String` (namespace `System`) de methode `Substring()` op die twee `int`-parameters neemt!



- Wat is de betekenis van de eerste `int`-parameter bij de methode `Substring()`?

- Wat is de betekenis van de tweede `int`-parameter bij de methode `Substring()`?

Een voorbeeldje in het venster Immediate om dit te staven:

```
Immediate Window
? String tekstje = "Hallo wereld";
? tekstje.Substring(1,4)
"allo"
```

(*) opgelet in Java hadden deze parameters een iets andere betekenis!

Methode `String AlfaVoornaam()`

Bij het sorteren van namen zorgen speciale tekens (een '-' of een spatie) en accenten ("é", "è") soms voor problemen.

Om deze problemen in de toekomst voor te zijn, voorzien we in de klasse `Naam` een methode `AlfaVoornaam()` die **als retourwaarde** de voornaam teruggeeft. Bij deze retourwaarde zijn echter alle 'probleem'-tekens opgeschoond.

Heel concreet:

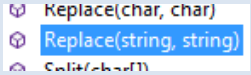
- spaties en koppeltekens laat je verdwijnen;
- de letters "é" en "è" laat je vervangen door een gewone "e";
- retourneer de 'bewerkte' voornaam helemaal in kleine letters.

```
Immediate Window
? Naam n1 = new Naam("Hélène", "Declercq");
? Naam n2 = new Naam("Jean-Pierre", "Declercq");
? n1.AlfaVoornaam()
"helene"
? n2.AlfaVoornaam()
"jeanpierre"
? n1.Voornaam
"Hélène"
```

De laatste instructie hierboven ("`? n1.Voornaam`") tikten we erbij om aan te tonen dat de methode `AlfaVoornaam()` de toestand van het object (meer bepaald de waarde van het veld `_voornaam`) niet verandert.

Ook hier zal je enkele specifieke `String`-methoden nodig hebben om deze oefening efficiënt aan te pakken. Onderstaande vraagjes zetten jullie op weg:

- Vraag in de Object Browser bij het type `String` (namespace `System`) de methode `Replace()` op die twee `String`-parameters neemt!



  - Wat is de betekenis van de eerste `String`-parameter bij de methode `Replace()`?

  - Wat is de betekenis van de tweede `String`-parameter bij de methode `Replace()`?

  Een voorbeeldje in het venster Immediate om dit te staven:

```
Immediate Window
? String tekstje = "Hallo wereld";
? tekstje.Replace("e","€")
"Hallo w€r€ld"
```

  Met wat creativiteit kom je er wel op hoe je met deze methode `Replace()` ook de koppeltekens en spaties uit een `String` kan laten *verdwijnen* (i.p.v. te laten *vervangen*).

- De methode `ToUpper()` kwam al even aan bod. Dan snappen jullie onmiddellijk ook wat de methode `ToLower()` doet.

Methode `String AlfaFamilienaam()`

We willen ook een methode `AlfaFamilienaam()` waar we analoog als bij `AlfaVoornaam()` nu een 'opgekuiste' familienaam retourneren.

```
Immediate Window
? Naam n3 = new Naam("Irma", "Van Den Bos");
? n3.AlfaFamilienaam()
"vandenbos"
```

Om bij beide methoden niet telkens dezelfde code te moeten programmeren, lijkt de introductie van een hulpmethode een veelbelovend idee!

5 Over Solutions, Projecten en Formulieren

Ondertussen, meer dan 100 pagina's ver in de cursus, zijn er al heel wat voorbeelden de revue gepasseerd. We hebben al een tros businessklassen bestudeerd en werkten tevens al meerdere formulieren (= Presentation Layer) uit.

We hebben echter nog geen enkele keer een toepassing "from scratch" opgebouwd. Tot nu kreeg je steeds een bronproject dat in meer of mindere mate al door de leerkracht 'geprepareerd' was.

Dit hiaat zullen we in dit hoofdstuk opvullen. Omdat we hierbij de tijd nemen om alle stappen uitvoerig te becommentariëren, kunnen we het zelfs wat professioneler aanpakken en zullen onze twee lagen, met name de Business- en Presentation Layer, nog explicieter van elkaar scheiden.

We gooien ook de beperking overboord waarbij elke toepassing slechts één formulier bevat. Toepassingen waar we via buttons op het ene formulier, andere formulieren laten openen, worden straks dagelijkse kost.

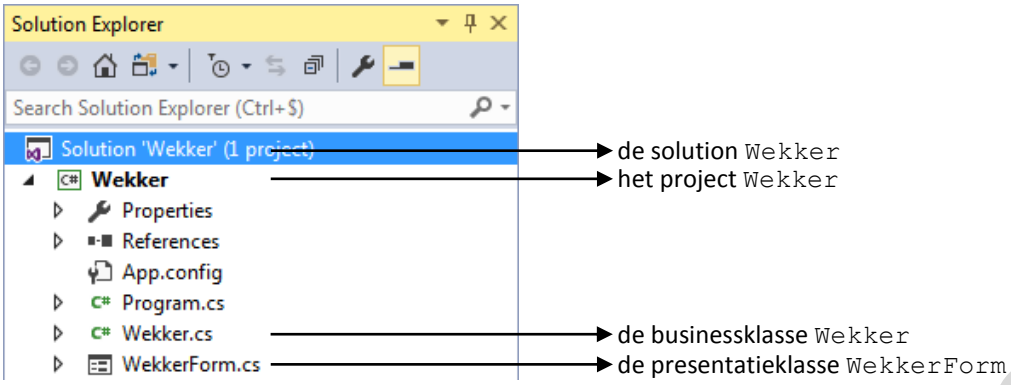
5.1 Terminologie

Tot nu hebben we steeds gesproken over Visual Studio '**projecten**'. Om bijvoorbeeld de Wekker-toepassing uit hoofdstuk 1 en 2 te openen in Visual Studio, verwezen we naar het '**project**' `Wekker`.

Dit was een beetje slordig van ons. Eigenlijk moesten we het dan hebben over de '**solution**' `Wekker`. Om een toepassing te openen in Visual Studio, dubbelklikken we immers op het **.sln**-bestandje in de bronmap. Die extensie "sln" staat voor "solution".

We willen onze onachtzaamheid weer goed maken! Maar dan moet je ons wel de gelegenheid en tijd geven om eens meer in detail naar zo'n Visual Studio project, sorry 'solution', te kijken.

We beginnen ons verhaal bij de **Solution Explorer** in de oude Wekker-toepassing. Bij de brongegevens van hoofdstuk 5 hebben we een kopie van deze toepassing gezet.



De correcte lezing van dit venster is dat we een solution `Wekker` hebben die één project bevat, nl. het project `Wekker`. Dit project `Wekker` bevat op zijn beurt meerdere elementen, waaronder het bestand **Wekker.cs** met de businessklasse `Wekker` en het bestand **Wekker–Form.cs** met de presentatieklasse `WekkerForm`.

De Business Layer (klasse `Wekker`) en de Presentation Layer (klasse `WekkerForm`) zitten dus broederlijk bij elkaar in het project `Wekker`.

Omdat onze solutions tot nu altijd uit één project bestonden en de solution en het project dezelfde naam droegen, was het niet echt noodzakelijk om tussen beide een onderscheid te maken. Solutions kunnen echter een complexere structuur hebben. Zo ook de toepassing die we hieronder "from scratch", steentje per steentje, zullen opbouwen.

5.2 Een toepassing: from zero to hero

Omdat jullie tot nu in deze cursus steeds konden starten van een 'voorgedefinieerd' bron-project, zullen jullie nog enkele noodzakelijke stappen missen om vanaf een blanco situatie een nieuwe toepassing op te zetten. De beste manier om dit te leren, is eens samen met ons de volledige procedure te doorlopen.

Om jullie aan den lijve te laten voelen wat het verschil is tussen 'solutions' en 'projecten', kiezen we ervoor om onze toepassing iets complexer te organiseren dan jullie tot nu gewoon waren. Het wordt **één solution** die uit **twee projecten** zal bestaan. In het ene project werken we de **Business Layer** van de toepassing uit. Het andere project bevat de **Presen–tation Layer**.

Deze nieuwe structuur zal ons enkele nieuwe hindernissen, maar gelukkig net zoveel oplossingen, brengen.

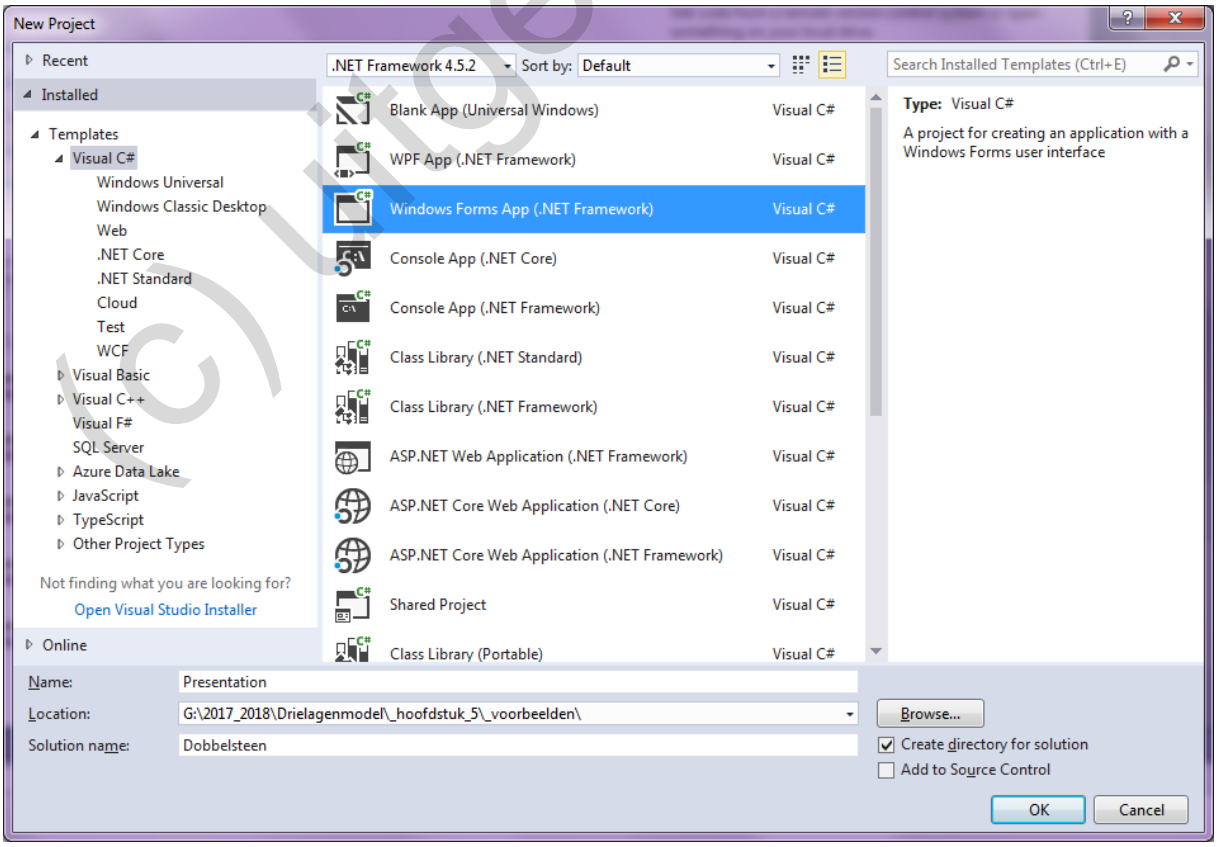
De case waarrond we gaan werken zijn **dobbelstenen**.

We hebben als eindresultaat onderstaand eenvoudig formulier voor ogen, waarmee we een dobbelsteen kunnen simuleren:

5.2.1 Een nieuwe solution maken

Om onze nieuwe solution te beginnen, gaan we uit van een situatie waar er in Visual Studio GEEN andere toepassing open staat. Mocht dit bij jou toch het geval zijn, raden we een : File | Close Solution aan (anders verschillen onderstaande screenshots misschien een beetje).

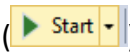
In Visual studio volg je: File | New | Project … Vul het dialoogvenster dat verschijnt als volgt in:



Met deze instellingen zullen we een nieuwe solution aanmaken met de naam **Dobbelsteen**, waarin een project zal steken met de naam **Presentation**. Het type van het project is '**Windows Forms App (.NET Framework)**'.

Wij hebben er met deze instellingen dus al onmiddellijk voor gekozen om de solution en het project een *verschillende naam* te geven.

Met het type van het project ('Windows Forms App') geven we aan dat het een project wordt met **Windows Formulieren**. Formulieren horen bij de **Presentation Layer** van de toepassing. Vandaar natuurlijk dat we dit project ook 'Presentation' genoemd hebben.

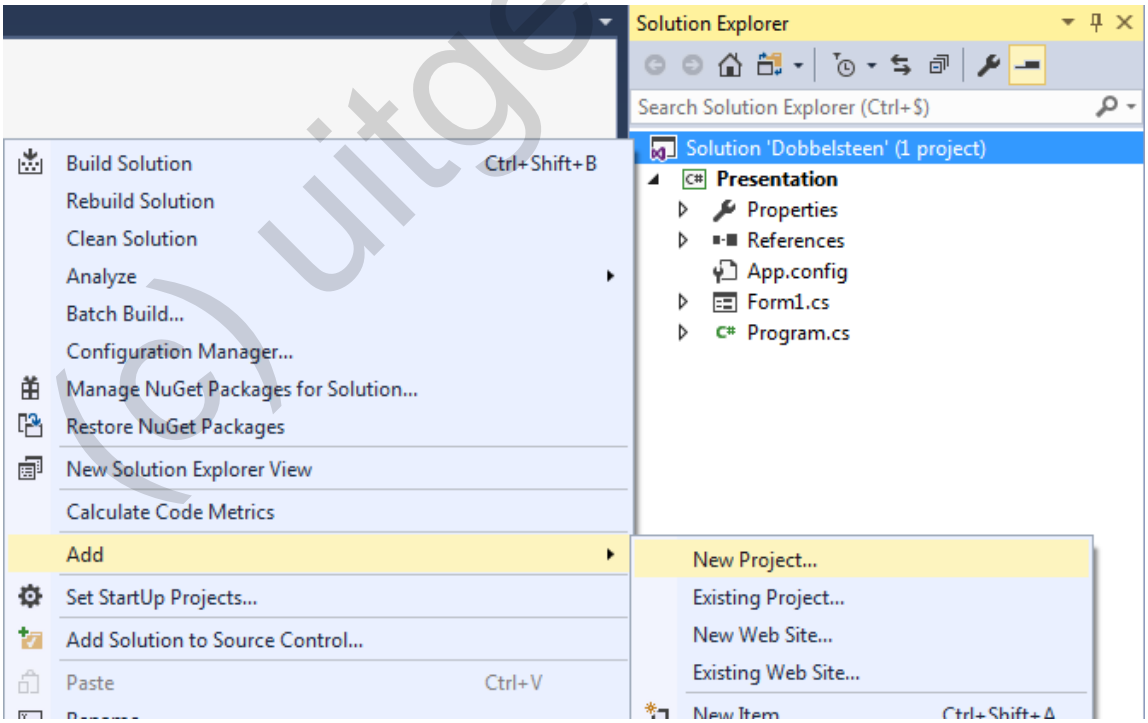
Het resultaat van onze actie is dat we een solution gecreëerd hebben waarin één project steekt. Dit project zal dienst doen als de Presentation Layer. Je kan de toepassing zeker al opstarten (

). Een zielig leeg formulier `Form1` zal geopend worden.

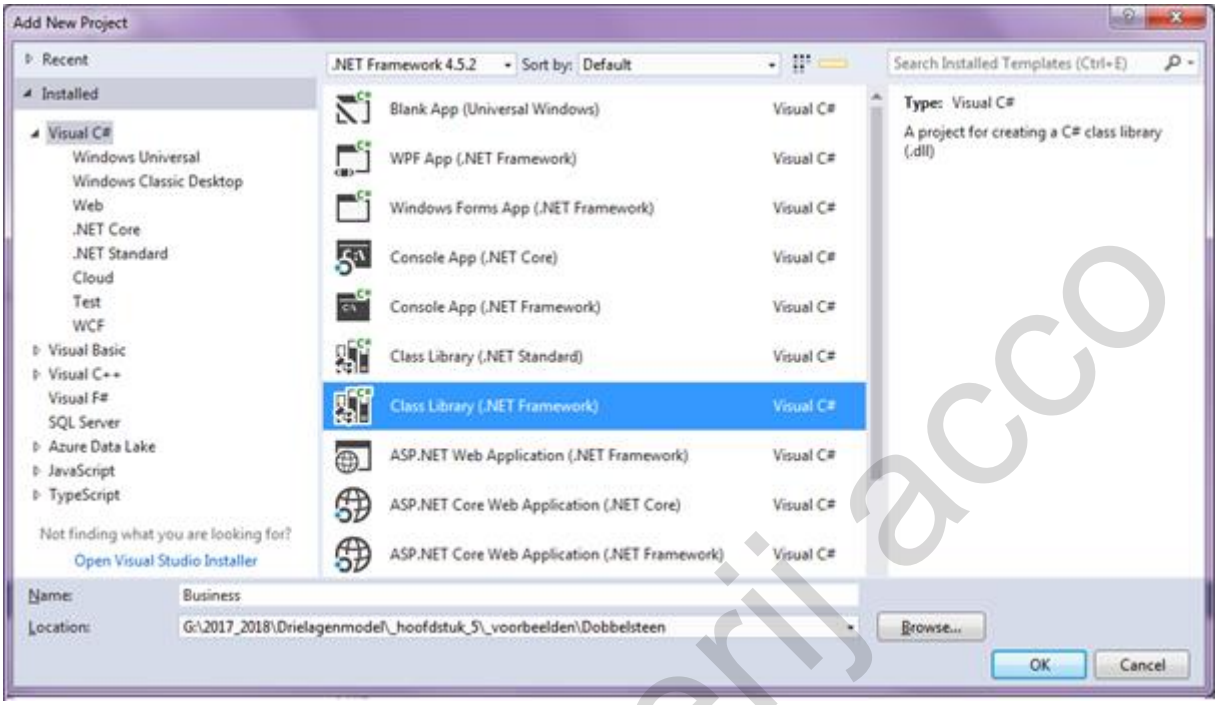
5.2.2 Een project toevoegen aan een solution

Ons opzet is om de Presentation Layer en de Businesscode volledig gescheiden te houden in onze toepassing. De Presentation Layer heeft met het project `Presentation` al zijn eigen hokje gekregen. Voor de Business Layer gaan we een **nieuw project** toevoegen aan de solution.

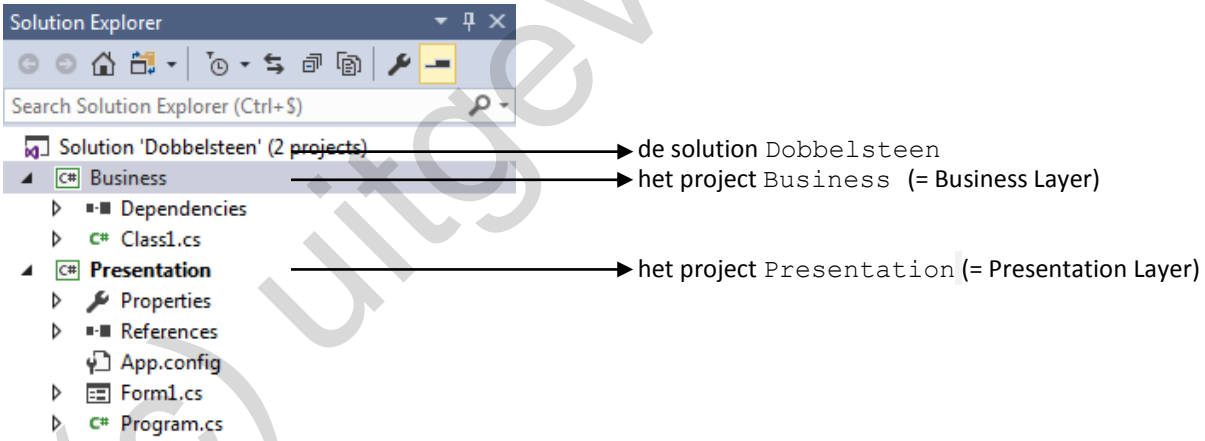
Klik hiervoor in de Solution Explorer rechts op de solution `Dobbelsteen` en volg dan Add | New Project…



In de Business Layer hoeven GEEN formulieren te komen. Als projecttype nemen we daarom deze keer "**Class Library (.NET Framework)**". Omdat dit project de Business Layer van de toepassing zal vormen, hebben we met **Business** een duidelijk naam beet.



In de Solution Explorer hebben we nu volgende constructie:



Eén project in de solution doet dienst als **Startup**-project. Dit betekent dat dit project zal gestart worden als je de toepassing laat lopen (

). In onze solution zou dit natuurlijk het project `Presentation` moeten zijn. Bij het 'starten' willen we immers een 'formulier' te zien krijgen.

In de Solution Explorer staat het Startup project in het vet. Bij ons is dit dus al meteen het juiste! We hebben dit verkregen door in de solution eerst het project `Presentation` te voorzien (en pas daarna het project `Business`). Wie eerst komt, mag de Startup zijn.

Mocht je dit ooit nodig hebben: via het snelmenu (rechts klikken op een project) kan je een ander project als Startup instellen.

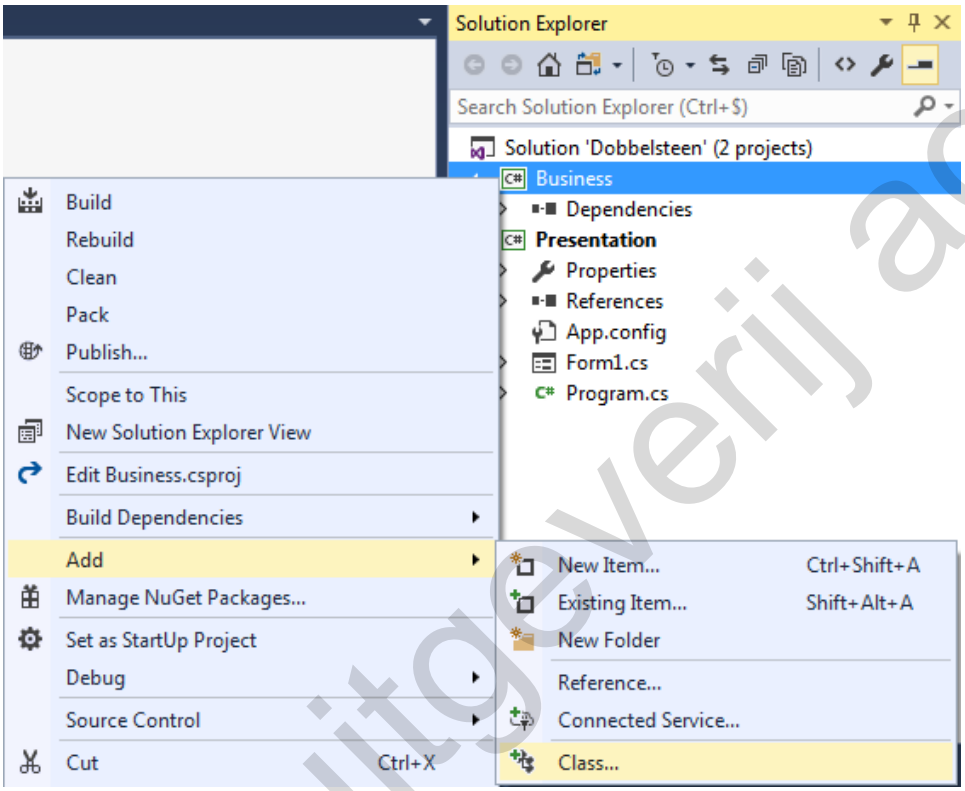
5.2.3 De Business Layer opbouwen

De volgende stap is dat we aan het project `Business` onze businesscode toevoegen.

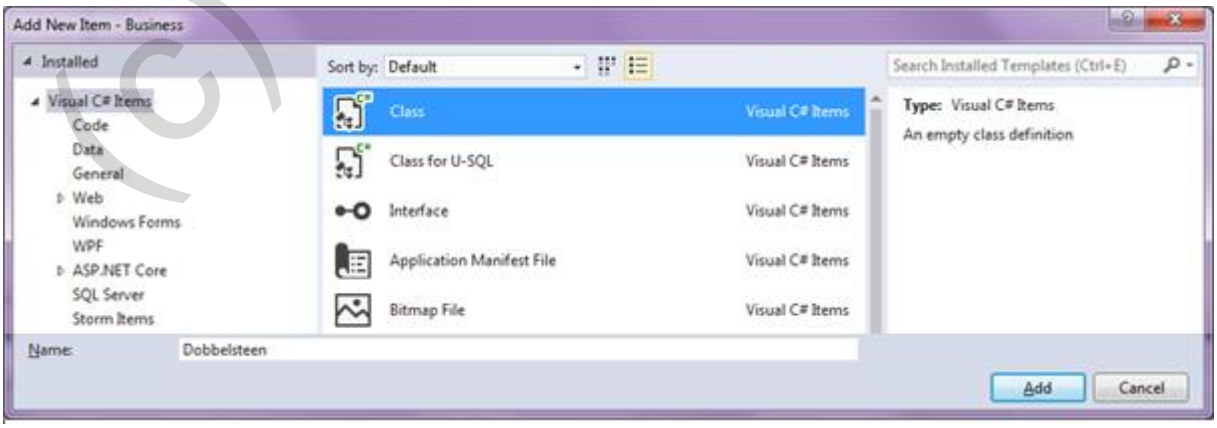
In het project `Business` staat momenteel een 'naamloze' klasse `Class1` (bestand **Class1.cs**). Hier zien we geen heil in, dus doe die klasse maar onmiddellijk weg.

Wat we wel nodig hebben is een klasse waarmee we een dobbelsteen kunnen simuleren. In onze businesslaag willen we hiervoor een nieuwe klasse `Dobbelsteen` voorzien.

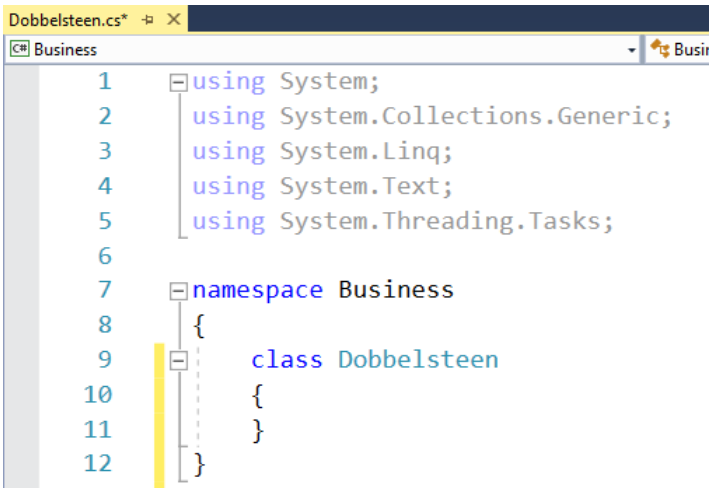
Klik in de Solution Explorer rechts op het project `Business` en volg Add | Class…



De nieuwe klasse noem je **Dobbelsteen**:



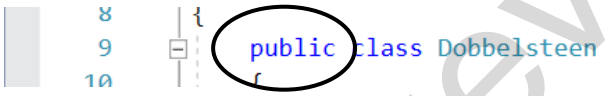
In de weergave Code ziet onze nieuwe klasse `Dobbelsteen` er zo uit:



Opmerkingen:

- Onze klasse `Dobbelsteen` is opgenomen binnen de namespace `Business`. De naam van de namespace komt dus standaard overeen met de naam van het project.
- Vervelend, maar Visual Studio maakt klassen standaard NIET publiek!

  We moeten bij de header van een nieuwe klasse er dus telkens zelf het sleutelwoord 'public' bijtypen:



Dan nog eens goed nadenken welke functionaliteit we in deze klasse `Dobbelsteen` gaan stoppen.

In essentie is een dobbelsteen niets meer dan een ding waarmee je kan werpen, wat je een willekeurige waarde tussen 1 en 6 oplevert. Of wacht eens, er bestaan naast de klassieke dobbelstenen ook exemplaren met een ander aantal zijden. Zo kan je met een 12-zijdige dobbelsteen een willekeurig getal tussen 1 en 12 gooien.

Bij ons gaat het dan nog eens om 'digitale' dobbelstenen (die niet in de echte wereld gefabriceerd moeten worden). Wij hebben dus helemaal geen beperkingen wat het aantal zijden betreft.

Het belangrijkste is alvast dat we in onze klasse `Dobbelsteen` een willekeurig getal kunnen laten genereren, waarbij we rekening houden met het aantal zijden op de dobbelsteen.

We komen aan volgende businesscode voor de klasse `Dobbelsteen`:

```
public class Dobbelsteen
{
    private int _aantalZijden;  // aantal zijden op de dobbelsteen
    private Random _getalGenerator;  // om willekeurige getallen te maken

    public Dobbelsteen(int aantalZijden)
    {
        _aantalZijden = aantalZijden;
        _getalGenerator = new Random();
    }

    public int AantalZijden
    {
        get { return _aantalZijden; }
    }

    public int Werp()
    {
        // retourneer willekeurige zijde op de dobbelsteen
        return _getalGenerator.Next(_aantalZijden) + 1;
    }
}
```

Een 'speed'-uitleg bij deze klasse:

- Bij de constructor, waarmee we nieuwe `Dobbelsteen`-objecten aanmaken, geef je als parameter het aantal zijden op. Dit aantal zijden houden we bij in een veld `int _aantalZijden`.
- Eens de dobbelsteen 'gemaakt' is, kan je het aantal zijden niet meer wijzigen. `AantalZijden` is immers een Read only Property.
- De computer een willekeurig getal laten kiezen, is functionaliteit die we in de systeemklasse `Random` vinden. Vandaar dat we in onze klasse `Dobbelsteen` een object nodig hebben van die klasse `Random`. We noemden dit betreffende veld `_getalGenerator`.

  In de constructor van `Dobbelsteen` laten we `_getalGenerator` initialiseren als een nieuw `Random`-object.
- De methode `Werp()` simuleert het werpen van een dobbelsteen. We laten deze methode bijgevolg een 'geworpen' getal retourneren.

  De methode `Next()` bij ons `Random`-object helpt ons om de computer een geldig willekeurig getal te laten genereren. De info uit de Object Browser bij deze methode:

  public virtual **int** **Next**(**int** *maxValue*)
  Member of **System.Random**

  Summary:
  Returns a non-negative random integer that is less than the specified maximum.

  Parameters:
  maxValue: The exclusive upper bound of the random number to be generated. maxValue must be greater than or equal to 0.

  Snap je waarom we de '+ 1' nodig hebben in de methode `Werp()`?

Een screenshot van het venster Immediate zegt soms meer dan een bladzijde uitleg. We willen er daarom vlug een test met zo'n `Dobbelsteen`-object opzetten:

```
Immediate Window
? Dobbelsteen d = new Dobbelsteen(6);
The type or namespace name 'Dobbelsteen' is not valid in this scope
```

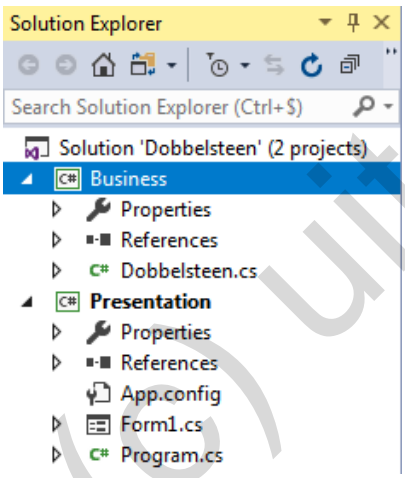
Oei(!), het venster Immediate oppert dat het de klasse `Dobbelsteen` niet kent! Dit is een eerste hindernis die opduikt omdat we onze solution opgedeeld hebben in verschillende projecten.

We lossen het probleem op door telkens als we naar een klasse verwijzen er ook de naam van **de namespace** waartoe onderhavige klasse behoort, bij te vermelden. Omdat de klasse `Dobbelsteen` onder de namespace `Business` valt, noteren we de instructie in het venster Immediate als volgt:

```
Immediate Window
? Business.Dobbelsteen d = new Business.Dobbelsteen(6);
{Business.Dobbelsteen}
    _bovengrens: 6
    _getalGenerator: {System.Random}
```

Dat onze instructies zo iets langer worden is vervelend, maar niet onoverkomelijk!

Een bijkomend punt is dat je in de Solution Explorer ook nog eens moet zorgen dat je er het juiste **project geselecteerd** hebt! Je moet er dus op het project `Business` gaan staan!



Dit allemaal om de methode `Werp()` van onze 6-zijdige dobbelsteen `d` aan het werk te zien:

```
? d.Werp()
1
? d.Werp()
5
? d.Werp()
3
```

Eerst een 1, dan een 5 en een 3, dat klinkt als drie 'willekeurige' dobbelsteenworpen. Jullie kregen bij je test ongetwijfeld een andere 'willekeurige' getallenreeks.

5.2.4 De Presentation Layer opbouwen

De volgende stap is om in de Presentation Layer ons dobbelsteenformulier te ontwerpen en te programmeren. Dit onderdeel kwam al uitvoerig aan bod in hoofdstuk 2 van deze cursus.

In de Presentation Layer (het project `Presentation`) werd standaard al het formulier `Form1` voorzien (bestand **Form1.cs**). Hernoem dit formulier naar **DobbelsteenForm.cs**. Visual Studio zal vragen om op de achtergrond enkele verwijzingen aan te passen. Geef je toestemming.

Kan je de tekst in de titelbalk van het formulier van de nietszeggende 'Form1' aanpassen naar bijvoorbeeld 'Dobbelsteen':

In de weergave Design

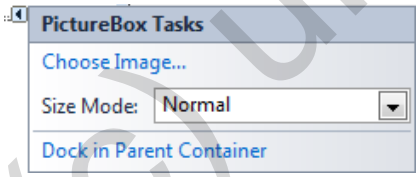
Op het formulier slepen jullie (in die volgorde) een `PictureBox`, een `TextBox` en een `Button`. Noem deze elementen respectievelijk `dobbelsteenPictureBox`, `dobbelsteenTextBox` en `werpenButton`.

Zet in het Properties venster de objecteigenschap `ReadOnly` van het tekstvak op **true**. Zo kunnen jokkebrokken er straks niet gewoon zelf een 6-worp intikken.

Bij de brongegevens van hoofdstuk 5 vinden jullie een afbeelding van een blanco dobbel–steen (**DobbelsteenBlanco.jpg**). Deze afbeelding willen we in de `PictureBox` gebruiken.

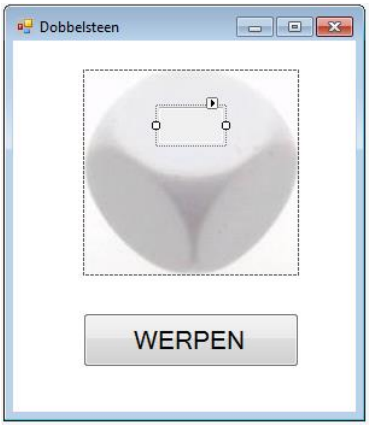
Werkwijze:

- Als je in de weergave Design de `PictureBox` selecteert, kan je er rechts bovenaan, via een pijl-icoontje, een menu openen:



- Via Choose Image… | Local Resource | Import kan je de bewuste afbeelding laten importeren.
- Het keuzelijstje 'Size Mode' bepaalt of de afbeelding steeds zijn originele grootte moet behouden (Normal), mag herschaald worden zodat de ruimte van de `PictureBox` gevuld wordt (StretchImage), …

Schik de elementen als volgt op het formulier:



In de weergave Code

De programmacode in de Presentation Layer zal voor jullie geen grote uitdaging meer zijn. Zeker bij dit 'eenvoudige' formuliertje laten we ons niet afschrikken.

- De logica van het werpen met een dobbelsteen hebben we geïmplementeerd in de businessklasse `Dobbelsteen`. In het formulier (dus in de klasse `DobbelsteenForm`) zullen we een businessobject `_dobbelsteen` van deze klasse `Dobbelsteen` declareren.
- We stellen voor om `_dobbelsteen` in de constructor van de klasse `DobbelsteenForm` te initialiseren als een 6-zijdige dobbelsteen.
- Elke keer als op de 'WERPEN'-knop geklikt wordt, zullen we eens met het businessobject `_dobbelsteen` 'werpen' en tonen we de retourwaarde van de worp in het tekstvak.

  Bij het openen van het formulier zouden we de `Dobbelsteen _dobbelsteen` al onmiddellijk een eerste keer kunnen werpen.

Bovenstaande puntjes kunnen we in volgende code vertalen:

```
public partial class DobbelsteenForm : Form
{
    // businessobject _dobbelsteen declareren
    private Dobbelsteen _dobbelsteen;

    public DobbelsteenForm()
    {
        InitializeComponent();

        // businessobject _dobbelsteen initialiseren met 6 zijden
        _dobbelsteen = new Dobbelsteen(6);

        // dobbelsteen werpen en resultaat in tekstvak tonen
        dobbelsteenTextBox.Text = _dobbelsteen.Werp().ToString();
    }

    private void werpenButton_Click(object sender, EventArgs e)
    {
        // dobbelsteen werpen en resultaat in tekstvak tonen
        dobbelsteenTextBox.Text = _dobbelsteen.Werp().ToString();
    }
}
```

Bij het overtikken van deze code kregen jullie waarschijnlijk ook volgend slecht nieuws:

```
15          // businessobject _dobbelsteen declareren
16          private Dobbelsteen _dobbelsteen;
17
18          public Do
19          {
20              InitializeComponent();
21
22              // businessobject _dobbelsteen initialiseren met 6 zijden
23              _dobbelsteen = new Dobbelsteen(6);
```

The type or namespace name 'Dobbelsteen' could not be found (are you missing a using directive or an assembly reference?)

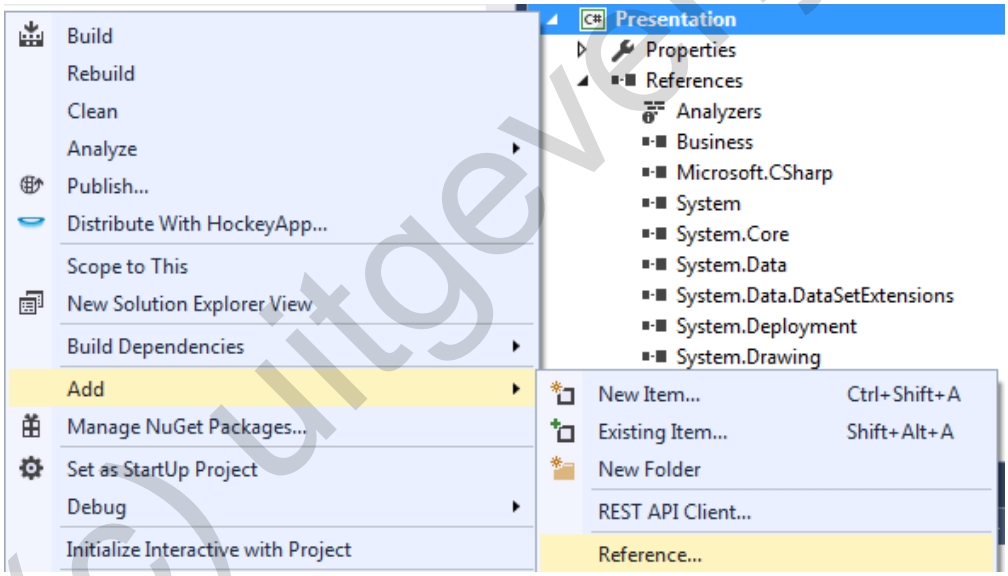
Show potential fixes (Alt+Enter or Ctrl+;)

De foutboodschap wijst erop dat de klasse `Dobbelsteen` niet gekend is.

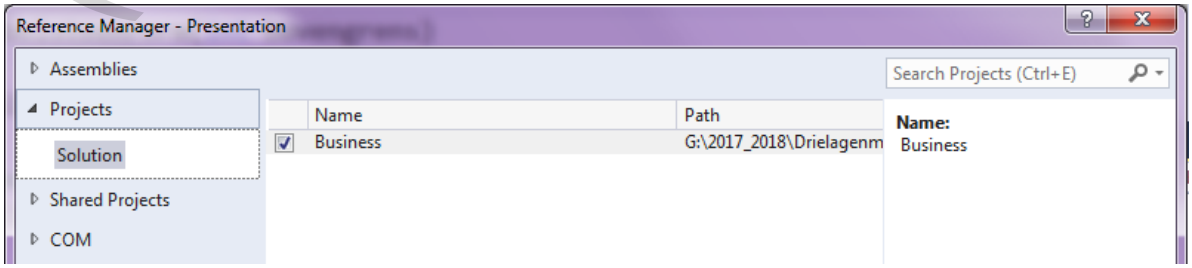
Dat de businesscode (klasse `Dobbelsteen`) en de presentatiecode (klasse `Dobbelsteen-Form`) in aparte projecten ondergebracht zijn, zorgt ook hier voor een hapering.

Het vraagt enkele acties om dit probleem op te lossen:

1. Eerst zullen we aan het project `Presentation` moeten kenbaar maken dat er ook nog een project `Business` is. We doen dit door in het project `Presentation` een **reference** toe te voegen naar het project `Business`. Klik in de Solution Explorer rechts op het project `Presentation` en doe Add | Reference.



In het dialoogvenster zet je een vinkje bij het project `Business` (zie Projects | Solution).



Het project `Presentation` heeft nu zijn verwijzing (reference) naar het project `Business`. In de weergave Code kleuren de lijntjes echter nog even rood:

```
15          // businessobject _dobbelsteen declareren
16          private Dobbelsteen _dobbelsteen;
17
18          public DobbelsteenForm()
19          {
20              InitializeComponent();
21
22              // businessobject _dobbelsteen initialiseren met 6 zijden
23              _dobbelsteen = new Dobbelsteen(6);
```

2. Omdat de klasse `Dobbelsteen` in een andere **namespace** staat (nl. in de namespace `Business`), moeten we in `DobbelsteenForm` (= namespace `Presentation`) een beetje *duidelijker* zijn.

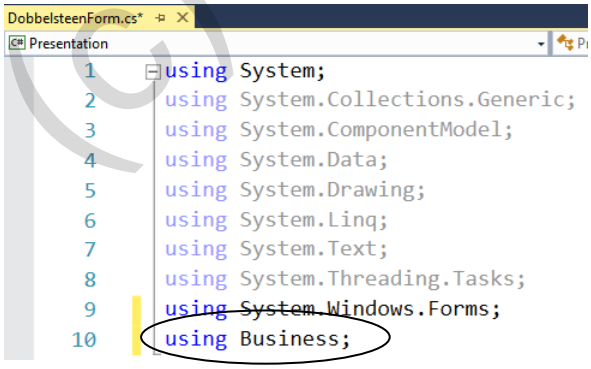
   Telkens we in deze code naar de klasse `Dobbelsteen` verwijzen, zetten we er de betreffende namespace `Business` vóór.

```
15          // businessobject _dobbelsteen declareren
16          private Business.Dobbelsteen _dobbelsteen;
17
18          public DobbelsteenForm()
19          {
20              InitializeComponent();
21
22              // businessobject _dobbelsteen initialiseren met 6 zijden
23              _dobbelsteen = new Business.Dobbelsteen(6);
```

   Op dit moment zijn de Errors uit de code verdwenen en zou je de toepassing moeten kunnen opstarten met een werkend dobbelsteenformulier als resultaat.

3. Een trucje om in je code niet telkens de namespace vóór de klasse te moeten typen, is bovenaan een using-statement op te nemen naar de betreffende namespace.

   Wij kunnen dus in de programmacode bij `DobbelsteenForm` zo'n using zetten met de namespace `Business`:



   De twee `Business`-vermeldingen die we in stap 2 ingetikt hebben (en die hierboven omcirkeld staan), mogen door dat ene extra using-lijntje dus weer weg.

Opgelet:

- Deze 'using' vervangt NIET de reference die we bij stap 1 gelegd hebben! Zonder die reference naar het project `Business`, haalt het using-statement niets uit.

En zo zijn we helemaal rond met de procedure om startend met lege handen een nieuwe toepassing uit de grond te stampen (en zijn jullie niet meer afhankelijk van de goodwill van je leerkacht om vooraf bronbestanden te preparen).

We hebben het onszelf bij deze dobbelsteen-toepassing wat extra moeilijk gemaakt door de Business en Presentation Layer helemaal van elkaar te scheiden en beide lagen in een afzonderlijk project onder te brengen.

Dit is echter de goede praktijk(!). Het gaf ons tevens de kans om enkele begrippen (zoals solution, project, reference, …) wat uit te diepen.

5.3 Een project met meerdere formulieren

Met onze dobbelsteen-toepassing hebben we in `DobbelsteenForm` een digitale dobbelsteen in elkaar geknutseld. Elke keer als we op het knopje 'WERPEN' klikken, krijgen we een nieuw willekeurig getal tussen 1 en 6.

Bij de businessklasse `Dobbelsteen` hebben we rekening gehouden met de mogelijkheid om een alternatieve dobbelsteen te maken. We bedoelen hiermee dobbelstenen die een ander aantal zijden hebben.

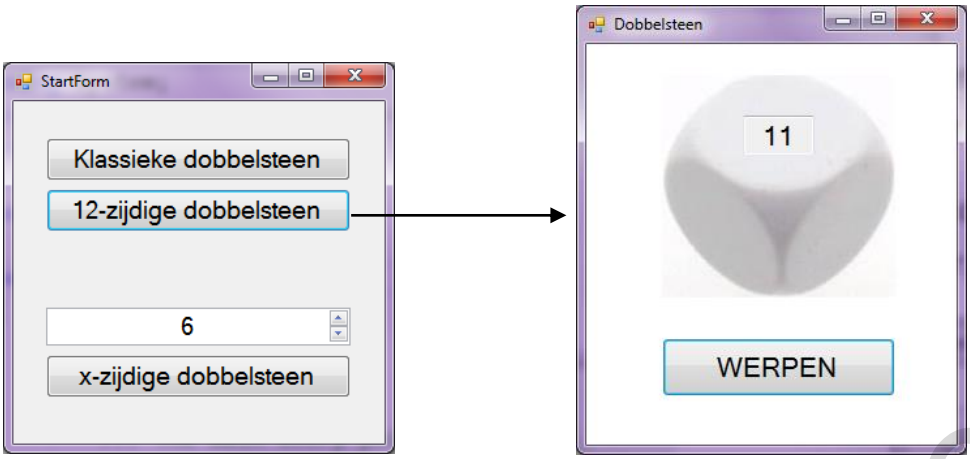
Stel dat we in het formulier met een 12-zijdige dobbelsteen willen gaan werpen, moeten we in de klasse `DobbelsteenForm` het `Dobbelsteen`-businessobject `_dobbelsteen` gewoon net iets anders initialiseren:

```
// businessobject _dobbelsteen initialiseren met opgegeven aantal zijden
_dobbelsteen = new Dobbelsteen(6);
_dobbelsteen = new Dobbelsteen(12);
```

Het aantal zijden van de dobbelsteen in het formulier `DobbelsteenForm` veranderen, kan dus zeker, maar het vraagt wel dat we een beetje in de programmacode gaan prutsen.

Als je het ons heel eerlijk vraagt, vinden we dit toch een beperking van onze toepassing. Er moet immers een ervaren C#-programmeur aangesproken worden om een andere digitale dobbelsteen te selecteren in het formulier. Moeten we het in onze toepassing niet ten–minste mogelijk maken dat de gebruiker *in run-time* zelf kan kiezen met welke dobbelsteen hij/zij wil spelen?

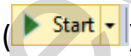
Om ons formulier `DobbelsteenForm` niet te belasten, zouden we het als volgt kunnen oplossen:



De gebruiker krijgt in dit voorstel eerst een ander (start)formuliertje te zien waar hij/zij kan kiezen welk soort dobbelsteen hij/zij wil. De verschillende knoppen openen elk ons gekende formulier `DobbelsteenForm` waarbij onmiddellijk de gevraagde dobbelsteen geselecteerd staat.

In bovenstaande screenshots vroeg de gebruiker een 12-zijdige dobbelsteen op. Dan is het inderdaad mogelijk dat de eerste worp een 11 is.

Met meerdere formulieren in één project werken, roept enkele nieuwe vragen op:

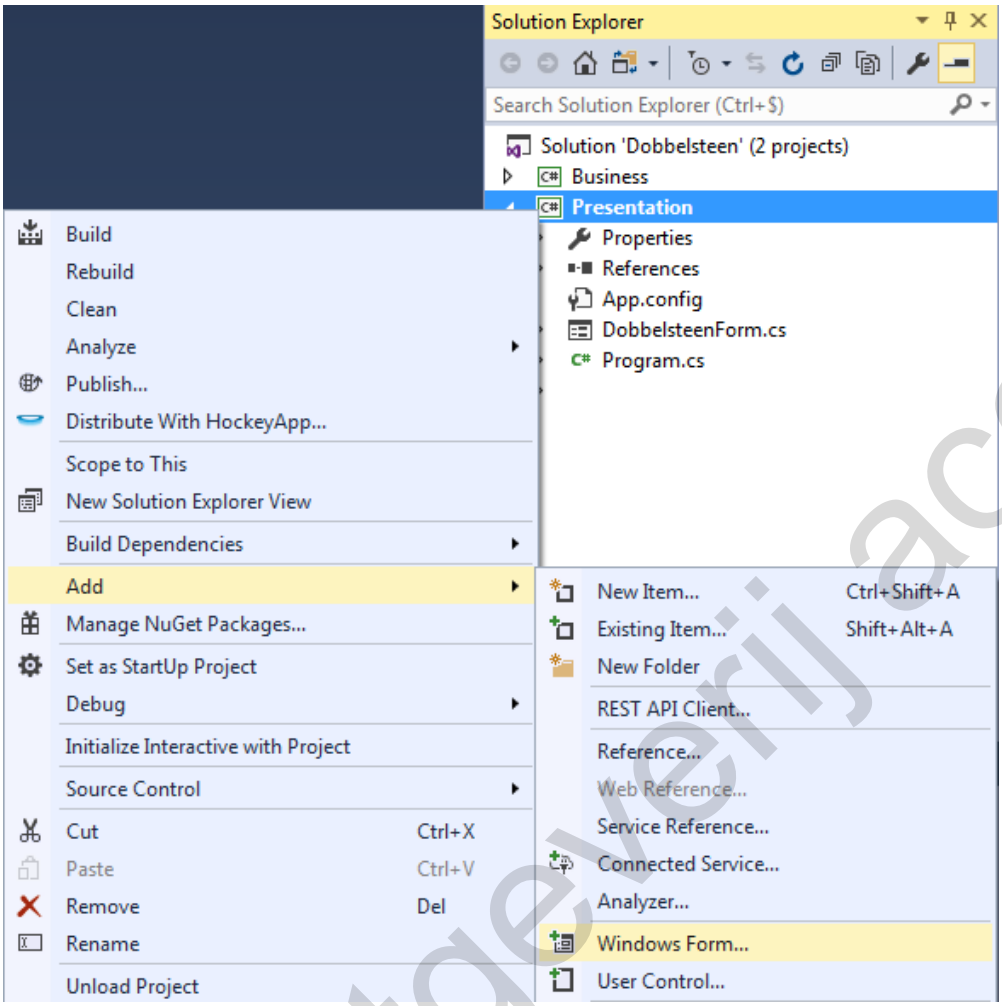
- Bij het starten van de toepassing (

), zou nu initieel het nieuwe formulier (om de dobbelsteen te kiezen) geopend moeten worden. Hoe kunnen we bepalen wat het startformulier is van een toepassing?
- Na het klikken op een knop op dit keuzeformulier moeten we `DobbelsteenForm` laten openen. Welke instructies hebben we nodig om een formulier te laten openen?
- En hoe zal ons formulier `DobbelsteenForm` weten welke dobbelsteen hij moet gebruiken?

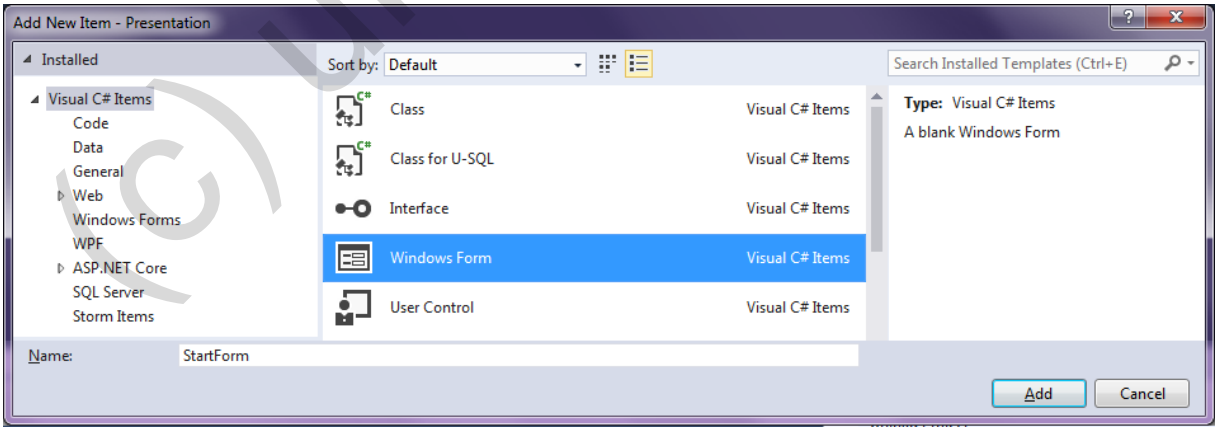
In onderstaande uitleg, vind je alle antwoorden op deze drie vragen.

Eerst moeten we natuurlijk het nieuwe keuzeformuliertje aanmaken. Dit formulier hoort overduidelijk bij de presentatielaag van de toepassing. Het nieuwe formulier zal dus in het project `Presentation` moeten komen.

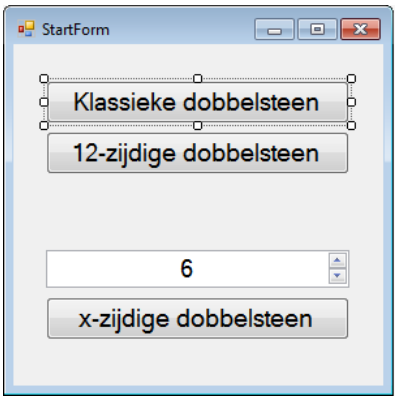
In de Solution Explorer klik je hiervoor rechts op het project `Presentation` en volg je: Add | Windows Form…



StartForm is een uitstekende naam voor ons nieuwe formulier:



De drie buttons (`Name`: `dobbelsteen6Button`, `dobbelsteen12Button` en `dobbelsteenXButton`) en de `NumericUpDown` (`Name`: `zijdenNumericUpDown`, `Value`: 6, `Minimum`: 1, `Maximum`: 100) krijgen jullie wel op dit formulier.



Het startformulier in een project

Hoewel we nog niets geprogrammeerd hebben, willen we dit nieuwe formuliertje al eens testen, maar met de Start komen we er nog niet. De toepassing starten laat immers nog steeds `DobbelsteenForm` openen en niet onze nieuwe `StartForm`.

Om het startformulier van het project `Presentation` te wijzigen, zal je naar het bestand **Program.cs** moeten. Dit bestand kwam in hoofdstuk 2 (zie pagina 59) uitgebreid aan bod. Fris er je kennis nog eens op.

We vervangen bij de laatste instructie in de methode `Main()` het formulier dat initieel geopend moet worden:

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new DobbelsteenForm());
    Application.Run(new StartForm());
}
```

Onthoud 5.1 In **program.cs** stel je het **startformulier** van het project in. Dit is het formulier dat als eerste geopend wordt als je het project opstart (Start).

De instructie waarmee je dit in de methode `Main()` regelt:

```
Application.Run(new FORM());
```

❖ Met *FORM* het gewenste startformulier.

Programmacode om een formulier te openen

Klikken op de drie knoppen op `StartForm` zou telkens het formulier `DobbelsteenForm` moeten laten openen.

We tonen bij de eerste knop ('Klassieke dobbelsteen') hoe het moet. Voeg aan de klasse `StartForm` onderstaande methode toe die reageert op de gebeurtenis `Click`:

```
private void dobbelsteen6Button_Click(object sender, EventArgs e)
{
    DobbelsteenForm formulier = new DobbelsteenForm();
    formulier.Show();
}
```

Twee stappen:

- Eerst declareren en initialiseren we een nieuw object `formulier` van de klasse `DobbelsteenForm`.

  ```
  DobbelsteenForm formulier = new DobbelsteenForm();
  ```

  `DobbelsteenForm` is de **klasse** die het dobbelsteenformulier beschrijft. Met deze instructie laten we dus m.a.w. een nieuwe **instantie** van die klasse `DobbelsteenForm` aanmaken.

  Van deze instructie mogen jullie absoluut niet schrikken. Dergelijke programmaregel hebben jullie al 1000 keer geschreven!

  Een object declareren en initialiseren met behulp van een constructor is usual stuff. Het enige verschil is dat het nu om een **formulierklasse** gaat.

- We moeten nog eens de methode `Show()` oproepen bij het formulierobject om dit formulier op het scherm te brengen.

  ```
  formulier.Show();
  ```

Het gaat vooruit …

Met 

wordt ons keuzeformulier geopend en de bovenste knop 'Klassieke dobbelsteen' laat `DobbelsteenForm` zien.

We kunnen met diezelfde code ook bij de twee andere knoppen ('12-zijdige dobbelsteen' en 'x-zijdige dobbelsteen') een instantie van `DobbelsteenForm` laten openen, maar hoe gaan we nu in godsnaam vertellen **wélke dobbelsteen** (een 6-, een 12- of een x-zijdige) er in `DobbelsteenForm` moet gebruikt worden? Jullie hebben nog één zetje nodig…

Parameters bij formulieren

Als we bij de constructor van `DobbelsteenForm` ons object `_dobbelsteen` als volgt laten initialiseren, zal in dit formulier ALTIJD een **6**-zijdige dobbelsteen gebruikt worden:

```
public DobbelsteenForm()
{
    InitializeComponent();

    // businessobject _dobbelsteen initialiseren
    _dobbelsteen = new Dobbelsteen(6);

    ...
```

De parameter die we bij de contructor van de klasse `Dobbelsteen` vermelden, heeft immers als betekenis het *aantal zijden* bij de nieuwe dobbelsteen.

Dit zouden we op één of andere manier **flexibeler** moeten kunnen maken. Het aantal zijden mag GEEN vast gegeven zijn. Door er nu een letterlijke '6' te zetten, zit de 6-zijdige dobbelsteen echter **ingebakken** in de klasse `DobbelsteenForm`.

Een creatieve oplossing is om bij de constructor van de klasse `DobbelsteenForm` ook een parameter toe te voegen:

```
public DobbelsteenForm(int aantalZijden)
{
    InitializeComponent();

    // businessobject _dobbelsteen initialiseren met opgegeven aantal zijden
    _dobbelsteen = new Dobbelsteen(aantalZijden);

    ...
```

Als we vanaf nu ergens een nieuw `DobbelsteenForm`-object aanmaken, zullen we via de parameter `aantalZijden` moeten opgeven hoeveel zijden we willen bij de dobbelsteen op het formulier. Deze parameter gebruiken we dan uiteraard om de `Dobbelsteen _dobbelsteen` te initialiseren.

Waar maken we nieuwe `DobbelsteenForm`-objecten aan? Inderdaad in het formulier `StartForm` waar we bij het klikken op elk van de drie knoppen een `DobbelsteenForm`-instantie declareren, initialiseren en op het scherm brengen.

Vraag bijvoorbeeld in `StartForm` de code achter de knop 'Klassieke dobbelsteen' op:

```
20      private void dobbelsteen6Button_Click(object sender, EventArgs e)
21      {
22          DobbelsteenForm formulier = new DobbelsteenForm();
23          formulier.Show();
24      }
```

DobbelsteenForm.DobbelsteenForm(int aantalZijden)
There is no argument given that corresponds to the required formal parameter 'aantalZijden' of 'DobbelsteenForm.DobbelsteenForm(int)'

De rood onderlijnde fout in de code hebben we natuurlijk zelf gezocht.

De instructie waarmee we het `DobbelsteenForm`-object `formulier` initialiseren is NIET meer geldig! De constructor van de klasse `DobbelsteenForm` verwacht voortaan immers een parameter! Bij een nieuw dobbelsteenformulier moeten we nu immers via die parameter het aantal zijden voor de gebruikte dobbelsteen doorgeven!

Omdat we met dit stukje code een klassieke dobbelsteen (6 zijden) willen gebruiken, maken we er het volgende van:

```
private void dobbelsteen6Button_Click(object sender, EventArgs e)
{
    // formulier met 6 zijdige dobbelsteen
    DobbelsteenForm formulier = new DobbelsteenForm(6);
    formulier.Show();
}
```

In het formulier `StartForm` programmeren we tenslotte de twee andere knoppen waarmee we een nieuw dobbelsteenformulier openen.

Bij de knop '12-zijdige dobbelsteen' doen we het nu zo:

```
private void dobbelsteen12Button_Click(object sender, EventArgs e)
{
    // formulier met 12 zijdige dobbelsteen
    DobbelsteenForm formulier = new DobbelsteenForm(12);
    formulier.Show();
}
```

Bij de knop 'x-zijdige dobbelsteen' bepaalt de `NumericUpDown` hoeveel zijden de dobbelsteen moet hebben. Of dus:

```
private void dobbelsteenXButton_Click(object sender, EventArgs e)
{
    // opvragen in NumericUpDown hoeveel zijden de dobbelsteen moet hebben
    int aantalZijden = (int) zijdenNumericUpDown.Value;

    // formulier met dobbelsteen met opgegeven aantal zijden
    DobbelsteenForm formulier = new DobbelsteenForm(aantalZijden);
    formulier.Show();
}
```

Opmerking:

- Soms heeft men wat moeite om ons trucje te doorzien waarmee we via een parameter bij de constructor een waarde laten doorgeven aan een *formulier*. Wij argumenteren dat dit eigenlijk geen issue zou mogen zijn.

  Parameters bij een constructor kwamen immers in de eerste lessen BlueJ al aan bod! Eens je inziet dat een formulierklasse (bijv. `DobbelsteenForm`) een klasse als een andere is, besef je dat we gewoon een techniek toepassen uit het basishandboek objectgeoriënteerd programmeren.

Oefening 5.1 Stel: Je start onze huidige dobbelsteen-toepassing op en in je enthousiasme klik je in het startformulier eens op elk van de drie buttons:

De bewering "mijn scherm staat vol met venstertjes" is waar, maar we willen het iets preciezer. **Hoeveel instanties** van **welke formulierklassen** staan er hier geopend?

Oefening 5.2 Aftelklok - We halen de aftelklok-toepassing nog eens boven, want hoofdstuk 5 gaf ons inspiratie om er nog enkele verbeteringen aan te brengen.

Open de solution `Aftelklok` in Visual Studio.

❖ Hoeveel **projecten** zie je in de Solution Explorer binnen de solution `Aftelklok` staan? Geef de projectna(a)m(en).

We hebben hier m.a.w. te maken met de 'oude' structuur waar de businesslaag (klasse `Aftelklok`) en de presentatielaag (klasse `AftelklokForm`) niet in afzonderlijke projecten ondergebracht werden.

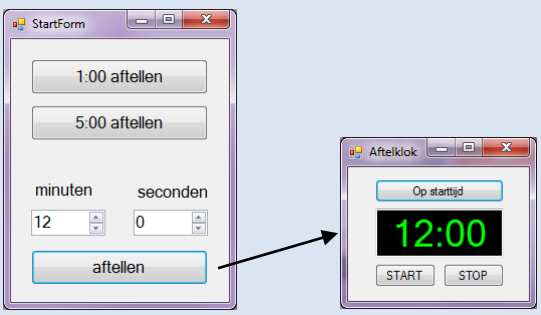
Als je het project opstart, opent een formulier met een klokje dat op **1:30** staat. De START-knop geeft het signaal dat de klok mag aftellen.

De reden waarom de klok op 1:30 start is natuurlijk gemakkelijk aan te wijzen bij de constructor van de formulierklasse `AftelklokForm`. In dat formulier gebruiken we het `Aftelklok`-object `_klokje` als businessobject. Dit object `_klokje` wordt daar op een minuut en een half geïnitialiseerd:

```
public partial class AftelklokForm : Form
{
    private Aftelklok _klokje;  // het veld _klokje declareren

    public AftelklokForm()
    {
        InitializeComponent();

        // het veld _klokje initialiseren als een nieuw Aftelklok-object
        // _klokje wordt hierbij ingesteld op 1 minuut 30 seconden
        _klokje = new Aftelklok(1, 30);

        // de tijd van _klokje in het tekstvak tonen
        displayTextBox.Text = _klokje.ResterendeTijd();
    }
```

Als we een andere starttijd willen, dan moeten we voorlopig nog programmacode gaan aanpassen. Dat moet flexibeler en beter.

We stellen opnieuw een extra startformulier voor, waar we kunnen kiezen op welke tijd de aftelklok moet starten. Willen we bijvoorbeeld de coopertest lopen:



We zetten jullie even op weg:

- Voeg aan het project `Aftelklok` een tweede formulier toe. Noem dit formulier `StartForm`.

  In de screenshot hierboven zie je dat er 3 knoppen, 2 labels en 2 `NumericUpDown`-elementen nodig zijn.

  `StartForm` programmeren doen we straks.

- Stel via **program.cs** in dat `StartForm` het **startformulier** in je project wordt. De toepassing opstarten zou dus voortaan `StartForm` en niet meer `AftelklokForm` moeten laten openen.

- We moeten enkele voorbereidingen treffen zodat het formulier `AftelklokForm` niet meer vastgeroest staat op 1 minuut en 30 seconden.

  Pas de header van de **constructor** van `AftelklokForm` aan zodat deze **twee parameters** neemt! De eerste parameter stelt de minuten voor waarop de klok moet starten; de tweede parameter zijn de seconden.

  Zorg dat het businessobject `_klokje` nu geïnitialiseerd wordt met deze parameters.

- Bij de drie opdrachtknoppen op `StartForm` gaan we code schrijven waarmee we telkens een `AftelklokForm`-formulier openen met de juiste starttijd.

  Een formulier openen, gebeurt in twee stappen:

  1) In de eerste instructie declareer en initialiseer je een nieuw `AftelklokForm`-object. Vergeet niet dat de constructor van `AftelklokForm` twee parameters vraagt!

  2) Het formulier zal pas getoond worden als je de methode `Show()` oproept bij dit object.

De situatie zou nu zo moeten zijn dat je met één klik op een knop een aftelklokje op 1 of op 5 minuten kan openen en dat je via de `NumericUpDown`-elementjes en de derde knop de starttijd helemaal kan personaliseren.

Wat nu echter nog mank loopt bij `AftelklokForm` is de knop 'Op starttijd'. Elke keer zet die knop de tijd terug op die 'oude' 1:30. Dit zou natuurlijk de 'starttijd' moeten zijn die we in `StartForm` gekozen hebben.

In onderstaand extraatje gaan we dit ook aanpakken. Het zandlopertje betekent dat we dit toch een moeilijker oefening vinden.

(⌛) Extra

Zorg dat de knop 'Op starttijd' op het formulier `AftelklokForm` zich ook **logisch** gedraagt. M.a.w. deze knop zou het klokje terug op de starttijd moeten zetten waarmee het formulier geopend werd.

We geven jullie volledig de vrije hand **hoe** je dit oplost en **waar** je in de toepassing code gaat aanpassen.

Als je echter onze gerespecteerde mening vraagt: "De 'starttijd' bijhouden voor een aftelklok vinden wij eerder een taak voor de **Business Layer**(!), of dus voor de klasse `Aftelklok`".

5.4 Een uitsmijter

Het principe waar we parameters laten doorgeven bij (de constructor van) een formulier-klasse heb je ondertussen via een oefening nog wat beter in je vingers kunnen krijgen. We durven het ondertussen wel aan om nog een stapje verder te gaan en deze techniek te perfectioneren!

Waar waren we ook alweer gebleven in onze dobbelsteen-toepassing?

In het startformulier `StartForm` laten we de dobbelsteen-formulieren openen met een instructie van de volgende vorm:

```
DobbelsteenForm formulier = new DobbelsteenForm(6);
```

Merk op dat we hierbij aan de constructor van de klasse `DobbelsteenForm` een getalletje (`int`) als parameter meegeven en zo duidelijk maken welke dobbelsteen (hoeveel zijden) we in het formulier wensen. Hier zeggen we dat we **6** zijden willen.

Deze parameter (deze `int`-waarde) wordt bij de constructor van de klasse `DobbelsteenForm` dan gebruikt om de juiste dobbelsteen (met het gevraagde aantal zijden) te initialiseren:

```
public partial class DobbelsteenForm : Form
{
    // businessobject _dobbelsteen declareren
    private Dobbelsteen _dobbelsteen;

    public DobbelsteenForm(int aantalZijden)
    {
        InitializeComponent();

        // businessobject _dobbelsteen initialiseren met opgegeven aantal zijden
        _dobbelsteen = new Dobbelsteen(aantalZijden);

        ...
    }

    ...
}
```

Om aan `DobbelsteenForm` duidelijk te maken welke dobbelsteen gebruikt moet worden, vertellen we dus *hoeveel zijden* die dobbelsteen moet hebben. Dit konden we met de `int`-parameter `aantalZijden` regelen.

En nu komt onze switch:

Eigenlijk zouden we aan `DobbelsteenForm` rechtstreeks kunnen doorgeven *welke dobbelsteen* hij moet gebruiken! Het subtiele verschil is hierbij dat terwijl we de *'hoeveel zijden'* nog konden vatten in een `int`-parameter, het bij *'welke dobbelsteen'* effectief gaat over een volwaardig `Dobbelsteen`-object.

Het wordt duidelijker als we eens naar de alternatieve code kijken die we nodig hebben om deze switch te maken.

In de klasse `StartForm` zetten we achter de knop 'Klassieke dobbelsteen' nu volgende code:

```
private void dobbelsteen6Button_Click(object sender, EventArgs e)
{
    // een 6-zijdige dobbelsteen declareren en initialiseren
    Dobbelsteen dobbelsteen = new Dobbelsteen(6);

    DobbelsteenForm formulier = new DobbelsteenForm(dobbelsteen);
    formulier.Show();
}
```

Zoals je hierboven in het vet ziet, declareren en initialiseren we er nu een volwaardige (6-zijdige) dobbelsteen. We geven dit `Dobbelsteen`-object mee als parameter bij het aanmaken van ons nieuw `DobbelsteenForm`-formulier!

Dat we via de constructor van `DobbelsteenForm` nu een voldragen `Dobbelsteen`-object doorgeven (en niet meer een simpele `int`-waarde), betekent dat de constructor in `DobbelsteenForm` ook iets omgebouwd werd:

```
public partial class DobbelsteenForm : Form
{
    // businessobject _dobbelsteen declareren
    private Dobbelsteen _dobbelsteen;

    public DobbelsteenForm(Dobbelsteen dobbelsteen)
    {
        InitializeComponent();

        // businessobject _dobbelsteen initialiseren
        _dobbelsteen = dobbelsteen;

        ...
        }

    ...
}
```

Je ziet dat we het type van de parameter aangepast hebben:

```
public DobbelsteenForm(Dobbelsteen dobbelsteen)
```

We gebruiken het `Dobbelsteen`-object dat we via deze parameter binnenkrijgen (en die we aanspreken met de naam `dobbelsteen`), om ons businessobject `_dobbelsteen` te initialiseren:

```
// businessobject _dobbelsteen initialiseren
_dobbelsteen = dobbelsteen;
```

De code om in `StartForm` het knopje '12-zijdige dobbelsteen' aan te sturen is helemaal analoog aan de code bij de 'klassieke dobbelsteen'.

Voor de volledigheid misschien toch eens uitschrijven wat we van het knopje 'x-zijdige dobbelsteen' gemaakt hebben:

```
private void dobbelsteenXButton_Click(object sender, EventArgs e)
{
    // opvragen in NumericUpDown hoeveel zijden de dobbelsteen moet hebben
    int aantalZijden = (int) zijdenNumericUpDown.Value;

    // een x-zijdige dobbelsteen declareren en initialiseren
    Dobbelsteen dobbelsteen = new Dobbelsteen(aantalZijden);

    DobbelsteenForm formulier = new DobbelsteenForm(dobbelsteen);
    formulier.Show();
}
```

Opmerking:

- We hebben in dit hoofdstuk twee versies van de dobbelsteen-toepassing besproken. Enerzijds een versie waar we een `int`-parameter met het aantal zijden doorgeven aan `DobbelsteenForm`, anderzijds een versie waar de constructor van `DobbelsteenForm` een parameter van het type `Dobbelsteen` neemt.

  In dit voorbeeld is niet zo één, twee, drie aan te duiden welke versie de *beste* is.

  In andere toepassingen (lees: bij meerdere voorbeelden die in deze cursus nog volgen), blijkt de uitwerking waar we *objecten* doorgeven via de constructor van een formulier-klasse echt wel superieur (en noodzakelijk).

  We beseffen dat het tijd en moeite vraagt om goed uit te pluizen hoe de dobbelsteen-toepassing precies in elkaar gepuzzeld is. De arbeid die je hierin steekt, zal beloond worden doordat je bij toekomstige projecten beter zal begrijpen hoe deze opgebouwd werden, hoe de verschillende elementen er met elkaar samenwerken, …

  No sweat, no glory!

Oefening 5.3 Monopoly - We komen graag nog eens terug op ons Monopoly-voorbeeldje dat we in hoofdstuk 2 introduceerden (zie pagina 51).

Deze toepassing dateert nog uit een tijd waar we de Business Layer en de Presentation Layer samen in één project onderbrachten. De 'één solution met twee projecten'-structuur is hier dus nog niet van toepassing.

In de klasse `MonopolyStraat` hebben we er ondertussen wel de accessormethoden laten vervangen door overeenkomstige Properties.

Eerst terug even opfrissen …

Businessklasse `MonopolyStraat`

Je vindt in de toepassing een businessklasse `MonopolyStraat`. Met deze klasse kunnen we een straat uit het Monopoly-bordspel voorstellen.

De belangrijkste functionaliteit is dat we bij zo'n `MonopolyStraat`-object huizen en een hotel kunnen kopen en we steeds kunnen opvragen welke huurprijs dit oplevert als een tegenspeler op deze straat landt.

Onderstaand tabelletje toont welke huurprijzen er in Monopoly toegepast worden op bepaalde straten:

Eigendom	Onbebouwd	1 huis	2 huizen	3 huizen	4 huizen	1 hotel
Lakenstraat	22	110	330	800	975	1150
Veldstraat	22	120	360	850	1025	1200
Huidevettersstraat	26	130	390	900	1100	1275

Als je een nieuw `MonopolyStraat`-object aanmaakt, vraagt de constructor naast de straatnaam en de stad, nog eens alle afzonderlijke huurprijzen. De klasse `MonopolyStraat` is daarmee de onbetwiste winnaar van de competitie "meeste parameters bij de constructor".

In onderstaande screenshot in het venster Immediate laten we het object `veldstraat` declareren en initialiseren. We vragen vervolgens in twee stappen op wat de huurprijs is als er één huis in `veldstraat` gebouwd werd:

```
Immediate Window
? MonopolyStraat veldstraat = new MonopolyStraat("Veldstraat","Gent",22,120,360,850,1025,1200);
? veldstraat.KoopHuis();
? veldstraat.GeefHuur()
120
```

Presentatieklasse `StraatForm`

Met het formulier `StraatForm` verzekerden we ons van een handige user interface waar we met twee knopjes huizen en een hotel op zo'n straat kunnen plaatsen, terwijl de te betalen huur hierbij steeds geactualiseerd wordt:

Dit formulier (of de klasse `StraatForm`) gebruikt uiteraard een businessobject van de klasse `MonopolyStraat`. Dit object zal bijhouden over welke straat het gaat (straatnaam en stad) en wat de toestand van die straat is (bijvoorbeeld hoeveel huizen?). Aan dit object kunnen we steeds de huurprijs opvragen.

In onderstaand codefragment uit de klasse `StraatForm` zie je dat we dit businessobject `_straat` genoemd hebben en dat we `_straat` hier lieten initialiseren als de Monopoly-Veldstraat in Gent:

```
public partial class StraatForm : Form
{
    private MonopolyStraat _straat;

    public StraatForm()
    {
        InitializeComponent();

        _straat =
            new MonopolyStraat("Veldstraat", "Gent", 22, 120, 360, 850, 1025, 1200);

        ...
```

Als we met dit formulier niet langer met de Veldstraat uit Gent, maar bijvoorbeeld eens met de net iets minder prestigieuze Lakenstraat uit Brussel willen spelen, moeten we in bovenstaande code de initialisatie van het businessobject `_straat` als volgt vervangen:

```
_straat =
      new MonopolyStraat("Lakenstraat", "Brussel", 22, 110, 330, 800, 975, 1150);
```

En hiermee hebben we meteen de vinger op de wond gelegd. Het formulier `StraatForm` werkt maar voor één specifieke Monopoly-straat. Willen we een andere straat, dan moet er

geprogrammeerd worden (of concreter bij de initialisatie van het businessobject `_straat` moeten acht parameters nagekeken worden).

Als we de Monopoly-fanclub willen verblijden met een toepassing die alle tweeëntwintig straten kent, moeten we dus iets anders bedenken…

Formulier `BordForm`

In het project vind je het nieuwe formulier `BordForm` terug. We hebben hier een poging gedaan om het volledige Monopoly-bord na te maken. Elke knop stelt een vakje op het bord voor.

Je snapt wel wat de bedoeling is. Klik je op het knopje 'Veldstraat Gent' dan zou ons oude formulier `StraatForm` geopend moeten worden, maar dan uiteraard met alle 'personalia' van die specifieke straat. En dit principe willen we dan voor elk van de tweeëntwintig straten.

Kan je alle onderstaande stappen uitwerken, dan is het gefixt.

❖ Zorg dat `BordForm` het **startformulier** wordt in het project. Dit regel je in **Program.cs**.

We pikken er als voorbeeld even de knop 'Kortrijksestraat Gent' (of `vak23Button`) uit.

❖ Bij deze knop de code schrijven om het formulier `StraatForm` te openen, is nog niet de grootste uitdaging. Het lukt al met volgende instructies (neem de code over):

```
StraatForm formulier = new StraatForm();
formulier.Show();
```

Met de eerste instructie maken we een nieuw object aan van de klasse `StraatForm`. De tweede instructie laat dit formulier-object verschijnen op je scherm.

Zo eenvoudig is het natuurlijk niet, want `StraatForm` moet nu nog weten wélke Monopoly-straat hij moet weergeven (bijvoorbeeld hier de Kortrijksestraat uit Gent).

We hebben in dit hoofdstuk geleerd dat we deze informatie via parameters zouden kunnen doorgeven aan `StraatForm`. Maar met **twee** `String`-parameters waarmee we de straat en de stad opgeven, komen we er nog niet. Bij elke Monopoly-straat horen immers ook nog eens **zes** unieke afzonderlijke huurprijzen. **Acht parameters**, dat is toch wat veel.

Als we aan `StraatForm` moeten duidelijk maken welke Monopoly-straat moet weergegeven worden, zouden we dit natuurlijk ook in de vorm van een `MonopolyStraat`-object kunnen doen. En dit zou dus maar **één parameter** zijn!

We gaan het op die laatste manier programmeren!

❖ Aanpassingen aan de klasse `StraatForm`:

Zorg dat de **constructor** van de klasse `StraatForm` voortaan een **parameter** neemt van het type `MonopolyStraat`.

Deze parameter stelt voor welke Monopoly-straat in het formulier gebruikt moet worden. Denk na wat je in de constructor met deze parameter moet aanvangen.

❖ Aanpassingen aan de klasse `BordForm`:

Bij de knop 'Kortrijksestraat Gent' (of `vak23Button`) hadden we al code om het formulier `StraatForm` te openen.

Door onze aanpassingen moeten we nu echter bij de new-instructie waarmee we het `StraatForm`-object aanmaken, een `MonopolyStraat`-object meegeven als parameter! We maken hiermee aan `StraatForm` duidelijk welke Monopoly-straat hij in het formulier moet gebruiken!

Bij dit knopje gaat het over de 'Kortrijksestraat' uit 'Gent' waarbij de zes huurprijzen 18 (onbebouwd), 90 (1 huis), 250 (2 huizen), 700 (3 huizen), 875 (4 huizen) en 1050 (1 hotel) zijn.

Maak hier bijgevolg eerst een nieuw `MonopolyStraat`-object aan met bovenstaande gegevens en geef dit nieuwe object als parameter door aan `StraatForm`.

Met bovenstaande stappen hebben we al een werkend knopje voor de Kortrijksestraat uit Gent, maar … er staan nog eenentwintig andere straten op het spelbord.

Omdat het natuurlijk ook niet de bedoeling kan zijn om jullie een halfuur bezig te houden met het kopiëren en plakken van code, stellen we voor dat we ons op het bord verder beperken tot het Gentse.

Zorg dus dat minstens de knoppen 'Veldstraat Gent' en 'Vlaanderenstraat Gent' nog hun ding doen. We verwijzen naar het Excel-documentje **straten.xlsx** bij de bronbestanden voor de correcte huurprijzen.

(⌛) Kleur instellen op `StraatForm`

Er heeft iemand op `BordForm` de moeite genomen om alle straat-knoppen van de juiste kleur te voorzien. Laten we echter via één van de (Gent-)knopjes het formulier `StraatForm` openen, dan is het gekleurde vlak daar bovenaan altijd geel.

Zou het niet toffer zijn mocht in `StraatForm` de kleur van het knopje overgenomen worden?

Dit was natuurlijk een retorische vraag. Hup, aan de slag.

We helpen jullie vooruit met drie extra tips:

- Het formulier `BordForm` zou voortaan aan de nieuwe `StraatForm`-formulieren ook moeten vertellen **welke kleur** er gebruikt moet worden.

  Je kan dit regelen door bij de constructor van de klasse `StraatForm` hiervoor een **extra parameter** te voorzien.

  Herinner je je nog dat kleuren voorgesteld worden als objecten van de klasse `System.Drawing.Color`, waarbij `System.Drawing` de namespace en `Color` het eigenlijke type is. (*PS: `System.Drawing` is ook één van de namespaces die bij de using-statements opgesomd worden*).

- Het gekleurde vlak bovenaan op `StraatForm` is een `Panel` (zie Toolbox). In het Properties venster vind je de naam van dit `Panel` wel. Zoek de objecteigenschap waarmee je de kleur van dit `Panel` instelt.

- Bij het klikken op elk knopje op `BordForm`, waarmee je een nieuw `StraatForm`-formulier aanmaakt (en opent), zal je nu de gewenste kleur moeten meegeven. Die kleur is steeds de achtergrondkleur van de betreffende knop zelf.

6 Collecties in C#

In het zesde hoofdstuk van de cursus BlueJ, maakten jullie kennis met **collecties** in **Java**. Een collectie is een datastructuur waarin je meerdere objecten kan opslaan. Je herinnert je waarschijnlijk nog de **`ArrayList`** en de **`Array`**.

Om even grof het verschil tussen deze twee collecties te schetsen:

- De `ArrayList` was de structuur waar nieuwe elementen steeds achteraan toegevoegd werden. Elk element in de `ArrayList` kreeg hierbij automatisch een (opeenvolgend) indexnummer.
- Bij een `Array` moest je bij het initialiseren al onmiddellijk opgeven voor hoeveel elementen je plaats voorzag. Wilde je een element toevoegen, bepaalde je zelf op welke positie (welk indexnummer) in de `Array` dit element moest komen.

Het zal jullie geenszins verwonderen dat C# ook collecties kent.

We gaan ons in dit hoofdstuk concentreren op de **`List`**. Een `List` is de **C#**-variant van de `ArrayList` in Java.

Alle principes die je vorig jaar bij de `ArrayList` leerde kennen, zijn ook bij de `List` in C# van toepassing. De syntax (of hoe je de zaken precies noteert) verschilt wat.

Om jullie niet onnodig in verwarring te brengen, zullen we in deze cursus niet meer naar de Java-programmacode met de `ArrayList` verwijzen. De principes - die zeker nog ergens in jullie achterhoofd sluimeren - leren jullie dan onmiddellijk in de C#-variant toepassen.

In dit hoofdstuk leren we jullie met de `List`-collectie werken. We zullen hier enkel maar businesscode schrijven. Pas in hoofdstuk 7 kijken we na wat we hiermee in de Presentation Layer kunnen aanvangen.

De `Array`-collectie bestaat ook in C#, zij het opnieuw met een iets andere syntax dan bij Java. Omdat we in deze cursus niet *alles* kunnen behandelen, hebben we er na veel wikken en wegen voor geopteerd om de `Array` achterwege te laten (al voelden we ons na deze beslissing onmiddellijk een beetje triestig, want kiezen is altijd verliezen).

6.1 De objecten in onze lijst

In de solution `Reizen` willen we straks alle buitenlandse reizen die we in een bepaald jaar gemaakt hebben, bijhouden in een lijst.

De bedoeling is dat we dan enkele statistieken kunnen opvragen over deze reizen, zoals:

- Hoeveel reizen hebben we dat jaar gemaakt?
- Hoeveel hebben alle buitenlandse reizen ons samen gekost?
- Hoeveel dagen hebben we in een bepaald land doorgebracht?

Omdat we straks 'reizen' gaan bijhouden in een lijst, definiëren we eerst onze basisklasse `Reis`. Elk `Reis`-object stelt dan een afzonderlijke reis voor naar een bepaald land, voor een bepaald aantal dagen, met een bepaalde kostprijs:

```
public class Reis
{
    private String _land;
    private int _aantalDagen;
    private decimal _kostprijs;

    public Reis(String land, int aantalDagen, decimal kostprijs)
    {
        _land = land;
        _aantalDagen = aantalDagen;
        _kostprijs = kostprijs;
    }

    public String Land
    {
        get { return _land; }
        set { _land = value; }
    }

    public int AantalDagen
    {
        get { return _aantalDagen; }
        set { _aantalDagen = value; }
    }

    public decimal Kostprijs
    {
        get { return _kostprijs; }
        set { _kostprijs = value; }
    }

    public String Omschrijving()
    {
        String uitvoer = "";

        if (_aantalDagen == 1)
        {
            uitvoer = "1 dag naar " + _land + " voor " +
                      _kostprijs.ToString("C") + ".";
        }
        else
        {
            uitvoer = _aantalDagen.ToString() + " dagen naar " + _land +
                      " voor " + _kostprijs.ToString("C") + ".";
        }
        return uitvoer;
    }
}
```

Bekijk toch eventjes aandachtig bovenstaande code, zodat je voldoende vertrouwd bent met deze klasse.

In het project vind je ook een blanco klasse `JaarboekReizen`. Die klasse zullen we hieronder stap voor stap programmeren. Binnen deze klasse zullen we in een `List` alle buitenlandse reizen bijhouden die een persoon in een bepaald jaar maakte. We werken in de klasse `JaarboekReizen` dus met een `List` die objecten van de klasse `Reis` zal bevatten.

6.2 `List`-bewerkingen

6.2.1 Een `List` declareren

In de klasse `JaarboekReizen` voorzien we volgende twee velden:

```
public class JaarboekReizen
{
    private int _jaar;
    private List<Reis> _reizen;
}
```

In het veld `_jaar` zullen we het jaar opslaan waarvan we de reizen bijhouden.

Het tweede veld in de klasse hebben we `_reizen` genoemd. Dit veld `_reizen` is de **lijst**, waarin we onze reizen (die we dat gegeven jaar gemaakt hebben) zullen verzamelen.

Als je een `List`-variabele **declareert**, dan moet je tussen <> extra informatie geven. Je plaatst er het type van de elementen die je in de lijst zal stoppen. Met `List<Reis>` geven we m.a.w. aan dat ons veld `_reizen` een lijst is met `Reis`-objecten.

Onthoud 6.1 De declaratie van een `List`-variabele gebeurt altijd met een instructie van de vorm:

`List<Type> lijst;`

Met hierbij:

- `List` -> geeft aan dat het om een lijst gaat;
- *`<Type>`* -> geeft aan van welk type de objecten zijn die we in de lijst zullen stoppen;
- *`lijst`* -> de naam die je aan de lijst geeft (=de naam van het veld/variabele).

6.2.2 Een `List` initialiseren

Een vuistregel is dat je in de constructor van een klasse altijd alle velden van die klasse **initialiseert**. Voor onze klasse `JaarboekReizen` ziet de constructor er zo uit:

```
public JaarboekReizen(int jaar)
{
    _jaar = jaar;
    _reizen = new List<Reis>();
}
```

Het veld `_jaar` laten we in de constructor initialiseren via de parameter `jaar`.

We initialiseren onze lijst `_reizen` door de constructor van de klasse `List` op te roepen. Ook bij de initialisatie moet tussen <> het type komen van de objecten die we in de lijst zullen stoppen. In onze lijst `_reizen` zullen, zoals eerder gezegd, `Reis`-objecten komen.

Vergeet niet om achteraan de ronde haakjes te vermelden! We roepen hier namelijk de constructor van de klasse `List` op en het blijkt dat dit een constructor zonder parameters is!

Onthoud 6.2 Een `List` initialiseren gebeurt altijd met een instructie van de vorm:

```
lijst = new List<type>();
```

Met hierbij:

- *lijst* -> de lijst-variabele die we gaan initialiseren;
- `List` -> geeft aan dat het om een lijst gaat;
- *<Type>* -> geeft aan van welk type de objecten zijn die in de lijst komen.

6.2.3 Een Property van het type `List`

In de klasse `JaarboekReizen` definiëren we voor onze beide velden een Read only Property. Zo krijgen we straks de mogelijkheid om aan een `JaarboekReizen`-object het bewuste jaartal en de volledige lijst met reizen op te vragen.

```
public int Jaar
{
    get { return _jaar; }
}

public List<Reis> Reizen
{
    get { return _reizen; }
}
```

De Property `Jaar` is ondertussen triviaal.

Al jullie aandacht gaat natuurlijk naar de Property `Reizen`! Let er vooral op het **retourtype**. Deze eigenschap levert ons een lijst met 'reizen' op. Een `List` van `Reis`-objecten, dat is dus het type: `List<Reis>`.

6.2.4 Een object toevoegen aan een `List`

Als een `List` een structuur is waarin je meerdere objecten kan opslaan, willen jullie natuurlijk als eerste horen hoe je **een object toevoegt** aan zo'n lijst.

Wij werken hiervoor een methode `void VoegReisToe(Reis)` uit, die het `Reis`-object dat je meegeeft als parameter laat toevoegen aan onze `List _reizen`.

```
public void VoegReisToe(Reis reis)
{
    _reizen.Add(reis);
}
```

De nieuw toegevoegde reis komt altijd achteraan in de lijst terecht!

Onthoud 6.3 Een object toevoegen aan een `List` gebeurt met een instructie van de vorm:

`lijst``.Add(`*`object`*`);`

Met hierbij:

- *`lijst`* -> de lijst waaraan je een object wilt toevoegen;
- `Add()` -> de methode om een object toe te voegen aan een lijst;
- *`object`* -> het object dat je aan de lijst wilt toevoegen.

Het spreekt voor zich dat als je een lijst declareert van het type `List<Reis>` je daarna enkel objecten van het type `Reis` in die lijst kan stoppen.

6.2.5 Hoeveel objecten in een `List`

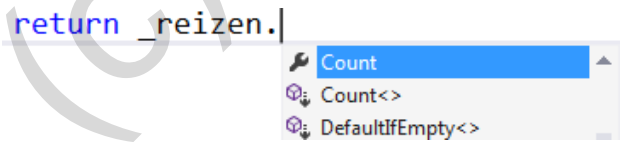
"**Hoeveel elementen** zitten er in de lijst?" is een heel frequente vraag. In de methode `int GeefAantalReizen()` tonen we hoe je dit te weten komt:

```
public int GeefAantalReizen()
{
    return _reizen.Count;
}
```

`Count` is de eigenschap waarmee je het aantal elementen in een `List` telt.

Dat `Count` een Property is (en geen methode) kan je afleiden uit:

- Er staan geen ronde haakjes na `Count`.
- Bij het intikken van deze instructie zie je in het invulmenu bij `Count` het Property-symbooltje staan:



Onthoud 6.4 Met de Property `Count` kan je aan een `List` opvragen hoeveel elementen die bevat. We krijgen dus een expressie van de vorm:

`lijst``.Count`

Met hierbij:

- *`lijst`* -> de lijst waarvoor je wilt weten hoeveel elementen deze bevat;
- `Count` -> de eigenschap die het aantal elementen telt.

Nu we voldoende methoden in onze klasse `JaarboekReizen` voorzien hebben, kunnen we al een kleine demonstratie opzetten in het venster Immediate.

```
Immediate Window
? Reis reis1 = new Reis("Zuid-Afrika",14,850);
? Reis reis2 = new Reis("Londen",4,310);
? JaarboekReizen opReis = new JaarboekReizen(2016);
? opReis.VoegReisToe(reis1);
Expression has been evaluated and has no value
? opReis.VoegReisToe(reis2);
Expression has been evaluated and has no value
? opReis.GeefAantalReizen()
2
```

Je ziet dat het ons lukte om met de methode `VoegReisToe()` twee `Reis`-objecten toe te voegen aan het `JaarboekReis`-object `opReis`. De methode `GeefAantalReizen()` telde immers correct twee elementen.

We vragen ons af of we in het venster Immediate eigenlijk de inhoud van een `List` kunnen printen? We testen het verder uit voor ons object `opReis`.

Via de Property `Jaar` vragen we het jaartal op aan ons object `opReis`. Dat lukt natuurlijk perfect. Zoals verwacht wordt mooi het jaar 2016 geprint.

```
? opReis.Jaar
2016
```

Via de Property `Reizen` vragen we de (volledige) `List` op met de <`Reis`>-objecten. Laten we in het venster Immediate deze lijst printen, dan verschijnt een eerder vreemd resultaat:

```
? opReis.Reizen
Count = 2
    [0]: {Reizen.Reis}
    [1]: {Reizen.Reis}
```

We krijgen te zien dat de lijst twee elementen bevat (Count is 2); de [0] en de [1] geven aan dat in de `List` een object steekt op indexnummer 0 en 1.

Met de tekst tussen de accolades (`{Reizen.Reis}`) kunnen we eigenlijk niet zo veel. Hier wordt aangegeven van welk type het object op die positie in de `List` is. Het type `Reizen.Reis` is samengesteld uit de namespace `Reizen` en de eigenlijke klasse `Reis`.

In hoofdstuk 7 komen we hier nog eens op terug en leer je hoe je in het venster Immediate tussen deze accolades veel nuttiger informatie kan laten printen.

6.2.6 Eén object uit een `List` opvragen

Elk element in een lijst krijgt automatisch een *opeenvolgend* **indexnummer**. Ook C# volgt de *vreemde* traditie om de telling te starten met **het nummer 0**. Als een `List` dus drie object-en bevat, zullen deze de indexnummers 0, 1 en 2 toegewezen krijgen.

Op basis van dit indexnummer kan je een afzonderlijk element uit een lijst opvragen. C# gebruikt hiervoor de **[]-notatie**, waarbij je tussen de rechte haken het gewenste indexnummer plaatst.

In de klasse `JaarboekReizen` bij de methode `Reis GeefReis(int)` passen we deze notatie toe:

```
public Reis GeefReis(int nr)
{
    if (nr >= 0 && nr < _reizen.Count)
    {
        return _reizen[nr];
    }
    else
    {
        return null;
    }
}
```

Wij geven bij de methode `GeefReis()` een nummer mee als parameter. Deze methode retourneert dan het `Reis`-object dat bij dit gegeven indexnummer staat in onze lijst `_reizen`. Vandaar dat we ook het retourtype van deze methode instelden op het type `Reis`!

Om fouten te voorkomen, hebben we in deze methode zelf de controle ingebouwd of er geen te klein / te groot indexnummer meegegeven werd. Zolang dit indexnummer positief is en kleiner blijft dan het aantal elementen in de lijst (`_reizen.Count`) is er geen probleem.

Bij een ongeldig indexnummer retourneert de methode **null**.

Onthoud 6.5 Om in een `List` te verwijzen naar een afzonderlijk object, gebruik je de **[]-notatie**. Tussen de rechte haken plaats je het indexnummer van het gewenste element. We krijgen dus een expressie van de vorm:

lijst`[`*nummer*`]`

Met hierbij:

- *lijst* -> de lijst waarvan je een elementje wilt opvragen;
- *nummer* -> het indexnummer van het element dat je in de lijst wilt aanspreken.

Opgelet: het indexnummer is **zero-based** (begint te tellen vanaf 0)!

We gaan verder in het venster Immediate. Voor ons `JaarBoekReizen`-object `opReis` kunnen we volgende expressies opstellen:

```
? opReis.GeefReis(1)
{Reizen.Reis}
    _aantalDagen: 4
    _kostprijs: 310
    _land: "Londen"
    AantalDagen: 4
    Kostprijs: 310
    Land: "Londen"
? opReis.GeefReis(0).Land
"Zuid-Afrika"
```

Bij de instructie `"? opReis.GeefReis(1)"` krijgen we een volledig `Reis`-object als resultaat terug (nl. het object met indexnummer 1 in de lijst). In het venster Immediate wordt hierdoor de volledige toestand van het object geprint.

Bij de instructie `"? opReis.GeefReis(0).Land"` vragen we via de Property `Land` de bestemming op van de reis die op indexnummer 0 staat. Het antwoord is hier van het type `String`.

6.2.7 Met een foreach-lus de objecten in een `List` doorlopen

Als we willen weten welk bedrag we aan al onze reizen samen gespendeerd hebben, is de enige mogelijkheid om **alle reizen** in de lijst `_reizen` te **overlopen** en ondertussen de totale kostprijs te berekenen.

De makkelijkste manier om de elementen in een lijst te doorlopen is met de **foreach-lus**.

```
public decimal TotaleKostprijs()
{
    decimal totaal = 0;

    foreach (Reis reis in _reizen)
    {
        totaal = totaal + reis.Kostprijs;
    }

    return totaal;
}
```

Bij een foreach-lus is er altijd sprake van **een collectie**. Deze collectie wordt tussen de haakjes vermeld na het sleutelwoord `in`. In onze methode `TotaleKostprijs` is de collectie de lijst met alle reizen of dus ons veld `_reizen`:

```
foreach (Reis reis in _reizen)
```

Onze collectie `_reizen` kan enkel objecten van het type `Reis` bevatten. Daarom declareren we binnen de haakjes ook een hulpvariabele `reis` (van dit type `Reis`):

```
foreach (Reis reis in _reizen)
```

De naam van deze variabele mag je uiteraard zelf kiezen (i.p.v. `reis` had dit bijvoorbeeld ook gewoon `r` kunnen zijn).

Een foreach-lus heeft volgend mechanisme: De variabele `reis` zal één na één elk object uit de lijst eens aanwijzen. De eerste keer dat de lus uitgevoerd wordt, zal `reis` wijzen naar het eerste element in de lijst (of het element met indexnummer 0), de tweede keer dat de lus uitgevoerd wordt, zal `reis` wijzen naar het tweede element in de lijst (of het element met indexnummer 1), enz…

Omdat we in de body van de lus telkens de kostprijs opvragen aan onze `reis`-variabele en die kostprijs bij een totaal optellen (`"totaal = totaal + reis.Kostprijs;"`), zal op het einde in de variabele `totaal` het bedrag staan dat we aan alle reizen samen besteed hebben.

Het alternatief `"totaal += reis.Kostprijs;"` om de prijzen op te tellen, mag natuurlijk ook.

Onthoud 6.6 Een foreach-lusje heeft altijd de vorm:

```
foreach(Type variabele in collectie)
{
    // body van de lus
}
```

Met hierbij:

- *collectie* -> de collectie waarvan je de objecten wilt overlopen; wij zullen hier dus een `List` gebruiken;
- *Type* -> het type van de elementen in de collectie;
- *variabele* -> de variabele die één na één alle objecten in de collectie zal aanwijzen.

In een tweede voorbeeldje op de foreach-lus laten we in een methode `int AantalDagenInLand(String)` berekenen hoeveel dagen we in een specifiek land op reis geweest zijn. Het gewenste land geven we mee als parameter met de methode.

```
public int AantalDagenInLand(String land)
{
    int totaalDagen = 0;

    foreach (Reis reis in _reizen)
    {
        if (reis.Land == land)
        {
            totaalDagen = totaalDagen + reis.AantalDagen;
        }
    }

    return totaalDagen;
}
```

Met bovenstaande foreach-lus overlopen we terug alle reizen uit de lijst `_reizen`. Het aantal dagen tellen we deze keer echter enkel bij het totaal op als de reis plaatsvond in het opgegeven land.

En zo hebben we jullie een voorbeeldje voorgeschoteld waar we de foreach-lus combineren met een if. In de cursus BlueJ vorig jaar was dit vaste koek!

6.2.8 Een object uit een `List` wissen

Als je elementen kan toevoegen aan een lijst (met de methode `Add`), moet het ook mogelijk zijn om een element weer uit een lijst te **verwijderen**.

De methode `void VerwijderReis(int)` toont hoe het moet:

```
public void VerwijderReis(int nr)
{
    if (nr >= 0 && nr < _reizen.Count)
    {
        _reizen.RemoveAt(nr);
    }
}
```

Bij deze methode geven we **het indexnummer** mee van het object dat we willen wissen uit de lijst. Het if-testje voorkomt dat we een element proberen te wissen op een onbestaand indexnummer (want dan crasht je programma).

Onthoud 6.7 Een element wissen uit een `List` kan met een instructie van de vorm:

```
lijst.RemoveAt(nummer);
```

Met hierbij:

- `lijst` -> de lijst waaruit je een object wilt wissen;
- *`nummer`* -> het indexnummer van het element dat je wilt wissen uit de lijst.

In de methode `void VerwijderReis2(Reis)` demonstreren we een alternatieve manier om een object uit een lijst te wissen:

```
public void VerwijderReis2(Reis reis)
{
    _reizen.Remove(reis);
}
```

Deze keer vertellen we niet op *welk indexnummer* we een object uit de lijst willen verwijderen, maar nu zeggen we rechtstreeks **welk object** uit de lijst weg moet. Je ziet dat we aan deze methode dan ook een `Reis`-object als parameter meegeven.

Onthoud 6.8 Een element wissen uit een `List` kan met een instructie van de vorm:

```
lijst.Remove(object);
```

Met hierbij:

- `lijst` -> de lijst waaruit je een object wilt wissen;
- *`object`* -> het object dat je wilt wissen uit de lijst.

Opmerking:

- Mocht eenzelfde object meerdere keren in de `List` zitten, dan zal de methode `Remove()` enkel het eerste exemplaar verwijderen.

Nog eens samenvatten:

- De methode `RemoveAt()` laat uit een `List` een element wissen op een gegeven **indexnummer**. Dit indexnummer geef je mee als parameter.
- De methode `Remove()` laat uit een `List` een gegeven **object** wissen. Dit object geef je mee als parameter.

In het venster Immediate toch even bewijzen dat de methoden `VerwijderReis()` en `VerwijderReis2()` beiden hun werk doen, alhoewel ze een verschillend type parameter nemen.

```
Immediate Window
? Reis reis1 = new Reis("Zuid-Afrika", 14, 850);
? Reis reis2 = new Reis("Londen",4,310);
? JaarboekReizen opReis = new JaarboekReizen(2016);
? opReis.VoegReisToe(reis1);
? opReis.VoegReisToe(reis2);
? opReis.GeefAantalReizen()
2
? opReis.VerwijderReis(0);
? opReis.GeefAantalReizen()
1
? opReis.VerwijderReis2(reis2);
? opReis.GeefAantalReizen()
0
```

6.2.9 Meerdere objecten uit een `List` wissen met een lusje

Je weet ondertussen dat je met een foreach-lus alle elementen uit een lijst kan doorlopen.

Als je tijdens die foreach echter elementen uit de lijst gaat wissen, dan raakt de foreach zo in de war, dat je programma gegarandeerd zal vastlopen! Wil je toch meerdere elementen wissen uit een lijst, dan zal je daarom een toevlucht moeten nemen tot een ander type lus!

Onze truc is om in zo'n situatie met een **for-lus** de elementen in de lijst van **achteren naar voren** te overlopen. We maken hierbij dankbaar gebruik van de **[]-notatie** om de afzonderlijke element uit de lijst aan te kunnen spreken.

Hieronder de methode `void WisReizenLand(String)` waarmee we als parameter een bepaald land meegeven. Alle reizen naar dat land, worden uit de lijst geduwd.

```
public void WisReizenLand(String land)
{
    for (int i = _reizen.Count - 1; i >= 0; i--)
    {
        if (_reizen[i].Land == land)
        {
            _reizen.RemoveAt(i);
        }
    }
}
```

Door in de for-lus de variabele `i` initieel te laten instellen op het grootste indexnummer in de lijst (`_reizen.Count-1`) en dan te laten aftellen (`i--`) tot de waarde 0 (`i >= 0`), zullen we met de expressie `_reizen[i]` alle elementen in de lijst (van achteren naar voren) eens laten aanwijzen.

Misschien ging het in bovenstaande alinea iets te snel. We willen daarom hieronder graag de werking van een for-lus, met iets meer structuur, uitleggen.

Onthoud 6.9 De for-lus is altijd van volgende vorm:

```
for (teller initialiseren; teller controleren; teller veranderen)
{
}
```

Bijvoorbeeld:

```
for (int i=1; i<=5; i++)
{
          // body van de lus
}
```

Bovenstaande for-lus wordt als volgt geëvalueerd:

- om te starten initialiseren we onze teller `i` op de waarde 1;
- zolang "`i<=5`" waar is, blijven we de body van de lus uitvoeren;
- elke keer als de body van de lus doorlopen werd, doen we `i++`, of m.a.w. onze teller wordt met eentje verhoogd.

Het bovenstaand lusje zal dus precies 5 keer uitgevoerd worden.

Nu we alle belangrijke `List`-bewerkingen behandeld hebben, is het aan jullie om dit eens op je eentje te oefenen.

6.3 Even oefenen

Oefening 6.1 **CocktailBar -** Als we eventjes nadenken over het begrip *cocktail* (= de klasse) en enkele voorbeeldjes (= objecten) op papier zetten dan kom je bijvoorbeeld aan:

Naam: Bloody Mary;
Kostprijs: € 8,25;
Ingrediënten: 100 ml tomatensap, 50 ml wodka en 8 ml citroensap.

Naam: Cuba Libre;
Kostprijs: € 7,50;
Ingrediënten: 100 ml cola, 50 ml rum en 8 ml limoensap.

Naam: Mojito;
Kostprijs: € 8,50;
Ingrediënten: 90 ml rum, 45 ml bruiswater, 15 ml suikersiroop en 8 ml limoensap.

Of m.a.w. elke cocktail heeft een naam en een prijs en bezit een **lijst** ingrediënten.

Bij de brongegevens van de oefeningen vind je de solution `CocktailBar` terug. Deze solution bevat één gelijknamig project `CocktailBar`. We splitsten deze toepassing dus NIET op in verschillende projecten.

Klasse `Ingredient`

Om straks bij een cocktail een lijst van ingrediënten bij te kunnen houden, zullen we eerst de basisklasse `Ingredient` definiëren.

Jullie moeten in het project `CocktailBar` zelf deze klasse `Ingredient` (of het klassen-bestand **Ingredient.cs**) aanmaken. Dit door in de Solution Explorer rechts te klikken op de projectnaam `'CocktailBar'` (staat er in het vet) en dan **Add** | **Class** uit te voeren!

Vergeet niet om de nieuwe klasse `Ingredient` publiek (public) te maken.

De velden van de klasse `Ingredient`

Uit de voorbeelden hierboven haalde je voor een ingrediënt misschien al de velden `String _naam` (= de naam van het ingrediënt) en `int _hoeveelheid` (de hoeveelheid van het ingrediënt - uitgedrukt in ml).

Om de oefening nog iets pittiger te maken, voegen we nog een extra veld `double _alcoholpercentage` toe, waarin we het alcoholpercentage van het ingrediënt bijhouden. Voor rum is dit bijvoorbeeld 40% (gaan wij opslaan als 0.40).

De constructor `Ingredient(String, int, double)`

Via drie overeenkomstige parameters laat je aan de constructor de waarden meegeven om de drie velden in te stellen.

De Properties bij de klasse `Ingredient`

Voor elk van de drie velden voorzie je een Property met zowel een get- als set-statement. Dus de klasse zal beschikken over de Properties `String Naam`, `int Hoeveelheid` en `double Alcoholpercentage`.

De methode `bool IsAlcoholisch()`

Tenslotte doen we er nog een methode bij met als retourwaarde een `bool` die aangeeft of het ingrediënt al dan niet alcoholisch is (of dus een alcoholpercentage groter dan 0 heeft).

Klasse `Cocktail`

En nu het grote werk! Voeg aan het project `CocktailBar` ook op de juiste manier de klasse `Cocktail`, of dus het klassenbestand **Cocktail.cs**, toe. Ook deze klasse zal je zelf publiek (public) moeten maken.

De velden van de klasse `Cocktail`

De 'gemakkelijke' velden zijn hier `String _naam` en `decimal _kostprijs`.

In een derde veld `_ingredienten` ga je alle ingrediënten opslaan om de cocktail samen te stellen. Het veld `_ingredienten` is dus een `List` van `Ingredient`-objecten. Zorg dat je bij de **declaratie** van dit veld het correcte type kiest!

De constructor `Cocktail(String, decimal)`

Deze constructor neemt twee parameters met de naam en prijs van de cocktail.

Laat er ook het veld `_ingredienten` **initialiseren** als een `List` van `Ingredient`-objecten!

De Properties bij de klasse `Cocktail`

Voor de velden `_naam` en `_kostprijs` voorzie je een Property met zowel een get- als set-statement. Houd je aan de naming conventions bij de naamgeving van deze Properties.

We willen ook een **Read only Property** `Ingredienten` waarmee we de lijst van ingrediënten (dus het veld `_ingredienten`) kunnen opvragen. Let goed op het type dat je deze Property geeft!

De methode `void VoegIngredientToe(Ingredient)`

In de klasse `Cocktail` definieer je de methode `VoegIngredientToe` waarmee je het ingrediënt dat je als parameter meegeeft, **toevoegt** aan de ingrediëntenlijst van de cocktail.

De methode `int HoeveelIngredienten()`

Deze methode retourneert het **aantal** ingrediënten in de ingrediëntenlijst van de cocktail. Voor bijvoorbeeld de Mojito zal dit 4 zijn (rum, spuitwater, suikersiroop en limoensap).

En nu eventjes testen ...

Bij de inleiding van deze oefening kon je lezen wat de prijs is van bijvoorbeeld een Cuba Libre en met welke ingrediënten je deze cocktail kan samenstellen.

Je opdracht is om in het **venster Immediate** alle instructies in te geven om zo'n Cuba Libre-cocktail, met alle nodige ingrediënten, aan te maken.

Bij deze taak zal je dus al zeker één `Cocktail` en drie `Ingredient`-objecten moeten declareren en initialiseren. Ter informatie: rum bevat 40% alcohol.

De methode `int InhoudCocktail()`

Deze methode retourneert hoe 'groot' (hoeveel ml) de cocktail is. Je maakt m.a.w. de som van alle hoeveelheden van de ingrediënten voor die cocktail.

De methode `int HoeveelAlcoholischeIngredienten()`

Deze methode telt hoeveel alcoholische ingrediënten de cocktail bevat en geeft dit aantal als retourwaarde terug.

De methode `bool IsAlcoholisch()`

Deze methode vertelt of een cocktail al dan niet alcoholisch is. Van zodra één ingrediënt maar een spatje alcohol bevat is het resultaat true!

Tip: Door een andere methode als hulpmethode te gebruiken, kan je je er bij deze oefening heel snel vanaf maken!

De methode `String AlleIngredienten()`

Je retourneert een `String` met daarin alle ingrediënten van de cocktail. We vermelden hierbij telkens de hoeveelheid en de naam van het ingrediënt. Om deze `String` een beetje leesbaar te houden plaats je na elk ingrediënt een puntkomma.

Voor bijvoorbeeld onze Cuba Libre zal de methode `AlleIngredienten()` dus volgende `String`-waarde retourneren: `"100 ml cola; 50 ml rum; 8 ml limoensap; "`.

De methode `Ingredient GeefIngredient(int)`

Als parameter geef je een indexnummer mee.

Deze methode retourneert het ingrediënt dat op die opgegeven positie staat in de `List _ingredienten`. Heb je opgemerkt dat het retourtype van deze methode vandaar dus `Ingredient` is?

Als een ongeldig indexnummer meegegeven wordt (lees: het indexnummer is negatief of te groot), laat je de methode **null** retourneren.

De methode `String GeefNaamIngredient(int)`

Als parameter geef je ook hier een indexnummer mee.

Deze methode *lijkt* heel sterk op de vorige methode `GeefIngredient()`, maar doet toch net iets anders, want hier moet je **de naam** retourneren van het ingrediënt dat op die opgegeven positie staat in de `List _ingredienten`. Vandaar dat je waarschijnlijk al opgemerkt hebt dat het retourtype van deze methode **`String`** is?

Als het indexnummer ongeldig is laat je de methode **de tekst "#Fout"** retourneren.

De methode `void VerwijderIngredient(int)`

Als parameter geef je een indexnummer mee.

Deze methode laat het element dat op die positie in de `List _ingredienten` staat, uit de lijst verwijderen.

Bouw opnieuw zelf een controle in zodat enkel in de situatie waar het indexnummer geldig is het betreffende element uit de lijst gewist wordt.

De methode `void MaakAlcoholvrij(decimal)`

Bij deze methode wis je alle *alcoholische* ingrediënten uit de ingrediëntenlijst van de cocktail.

Tip:

- Je zal, om de alcoholische ingrediënten te identificeren, sowieso met een lusje alle elementen in de lijst moeten overlopen. Terwijl je de lijst doorloopt, zal je de alcoholische elementen eruit moeten verwijderen.

  Opgelet: In zo'n situatie werkt de klassieke foreach-lus NIET(!) en moet je creatief met de for-lus aan de slag!

De `decimal`-parameter bij deze methode is de *nieuwe* prijs voor de cocktail. Een cocktail wordt natuurlijk een pak goedkoper als we er de (duurdere) alcoholische ingrediënten niet meer in mengen.

We gaan bij deze methode ook ineens de naam van de cocktail aanpassen. Om de klanten goed te informeren, plakken we hiervoor aan de huidige naam van de cocktail, het tekstje " - alcoholvrij" .

De methode `double Alcoholpercentage()`

Deze methode berekent (en retourneert) het alcoholpercentage van de cocktail.

Denk eens goed na hoe je uit alle verschillende ingrediënten (elk met een gegeven hoeveelheid en een bepaald alcoholpercentage) het alcoholpercentage van de *volledige* cocktail kan berekenen.

6.4 Een uitsmijter

We hebben nog één, iets moeilijker, voorbeeldje voorzien rond het werken met lijsten. We keren hiervoor even terug naar ons `Reizen`-project.

In de klasse `JaarboekReizen` houden we naast het jaartal (veld `_jaar`) een lijst bij van alle gemaakte reizen (het veld `_reizen` van het type `List<Reis>`). `Reis` is dan op zijn beurt de basisklasse die een afzonderlijke reis (land, aantal dagen en kostprijs) beschrijft.

De klasse `JaarboekReizen` bevat al heel wat functionaliteit. Er werd een ruime waaier aan methoden gedefinieerd waarmee we reizen kunnen toevoegen, kunnen tellen hoeveel reizen er gemaakt zijn, kunnen nagaan welk budget we aan onze reizen gespendeerd hebben, …

We willen aan de klasse `JaarboekReizen` nog een laatste methode toevoegen. Bedoeling is dat we kunnen opvragen **welke** *grote* reizen er gemaakt zijn. Een grote reis definiëren we als een reis die langer duurde dan 7 dagen en die meer kostte dan € 1000.

Het is zeker mogelijk dat iemand meerdere grote reizen maakt in een jaar. Als antwoord zal deze nieuwe methode in deze situatie dus meerdere `Reis`-objecten opleveren. Dit kan enkel lukken als het retourtype van de methode een `List` van `Reis`-objecten is.

We kunnen de methode `List<Reis> GroteReizen()` als volgt definiëren in de klasse `JaarboekReizen`.

```
public List<Reis> GroteReizen()
{
    List<Reis> groteReizen = new List<Reis>();

    foreach (Reis reis in _reizen)
    {
        if (reis.AantalDagen > 7 && reis.Kostprijs > 1000)
        {
            groteReizen.Add(reis);
        }
    }

    return groteReizen;
}
```

Het speciale aan deze methode is dat we een lijst moeten retourneren en vooral dat we die lijst hiervoor eerst zelf nog moeten aanmaken en opvullen.

De lijst waarin we de 'grote' reizen zullen stoppen noemen we `groteReizen`. Deze (hulp)-variabele moeten we dus declareren (en initialiseren) van het type `List<Reis>`. Enkel zo kunnen we in deze variabele meerdere `Reis`-objecten opslaan.

De betreffende instructie hiervoor:

```
List<Reis> groteReizen = new List<Reis>();
```

Om alle grote reizen te vinden, overlopen we de lijst met de reizen (of dus het veld `_reizen`). Dat klinkt weer als foreach-time. Met een bijkomende If gaan we de grote reizen identificeren.

```
foreach (Reis reis in _reizen)
{
    if (reis.AantalDagen > 7 && reis.Kostprijs > 1000)
    {
    }
}
```

Als een reis aan onze criteria voldoet, stoppen we die reis in de nieuwe lijst met:

```
foreach (Reis reis in _reizen)
{
    if (reis.AantalDagen > 7 && reis.Kostprijs > 1000)
    {
        groteReizen.Add(reis);
    }
}
```

De `Reis`-objecten die je op deze manier aan de `List groteReizen` toevoegt, blijven natuurlijk ook in de oorspronkelijke lijst `_reizen` staan. Het is voor een object geen enkel probleem om zich in meerdere lijsten te bevinden.

Op het einde van de methode laten we dan de nieuw samengestelde lijst retourneren:

```
return groteReizen;
```

Bij deze uitsmijter hoort ook een oefening.

Het principe waar je een methode een `List` laat retourneren, die je er eerst zelf moet aanmaken en dan laten opvullen, kan je zo nog eens in een andere context toepassen.

Oefening 6.2 CocktailBar - In de solution `CocktailBar` hebben we bij 'Oefening 6.1' (zie pagina 200) al heel wat `List`-programmacode geschreven.

De `List` waarmee we werken zit in de klasse `Cocktail` en houdt alle ingrediënten bij om de cocktail samen te stellen. Deze lijst werd in deze klasse `Cocktail` gedeclareerd als '`private List<Ingredient> _ingredienten;`'.

De elementen in deze lijst `_ingredienten` zijn dus objecten van de klasse `Ingredient`.

De methode `List<Ingredient> AlcoholischeIngredienten()`

We willen te weten komen welke ingrediënten van een cocktail er alcoholisch zijn. De nieuwe methode `AlcoholischeIngredienten()` in de klasse `Cocktail` moet dit voor ons regelen.

Omdat een cocktail meerdere alcoholische ingrediënten kan bevatten, laten we de methode `AlcoholischeIngredienten()` daarom een lijst van ingrediënten retourneren. Het retourtype moet dus wel `List<Ingredient>` zijn.

Volgende stappen heb je in deze methode `AlcoholischeIngredienten()` nodig:

- Declareer en initialiseer een nieuwe `List` voor `Ingredient`-objecten.
- Doorzoek alle ingredienten van de cocktail. De *alcoholische* onderdelen voeg je toe aan je nieuwe `List`.
- Laat de `List` retourneren.

De methode `List<Ingredient> NietAlcoholischeIngredienten()`

We geven de stoere methode `AlcoholischeIngredienten()` in de klasse `Cocktail` een klein lief zusje.

Met de nieuwe methode `NietAlcoholischeIngredienten()` laat je een lijst met de NIET-alcoholische ingrediënten teruggeven.

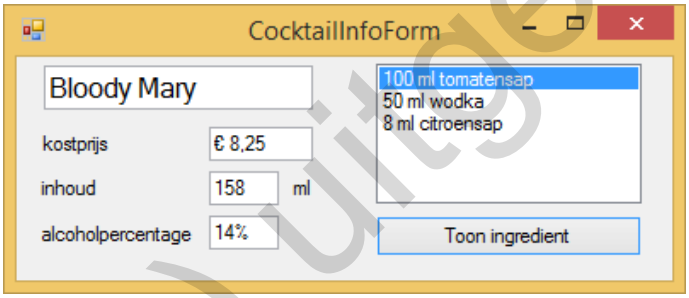
7 Collecties in de presentatielaag

In het vorige hoofdstuk '6 Collecties in C#' hebben jullie eigenlijk niet zo bijster veel nieuws bijgeleerd! We hebben jullie enkele voorbeelden gepresenteerd van hoe je met een `List` kan werken in C#. En geef toe, buiten enkele syntaxverschillen was dit eigenlijk één grote herhaling van de `ArrayList` uit Java (zie de cursus BlueJ).

Dit hoofdstuk zal veel verrassender zijn, want we gaan jullie tonen hoe je **collecties** (bij ons dus de `List`) kan verwerken in de **presentatielaag** van een toepassing. Zoals je weet was er in BlueJ nog geen sprake van een Presentation Layer.

Omdat een `List` intrinsiek al complexer van structuur is, zijn er toch enkele specifieke kneepjes nodig om de objecten uit zo'n collectie op een propere manier op een formulier te presenteren en om die objecten daar correct te laten manipuleren.

We zullen straks o.a. toewerken naar het volgende formulier:



We bouwen m.a.w. een Presentation Layer uit voor de CocktailBar-toepassing waarvoor jullie in hoofdstuk 6 in een lange oefening met veel zweet, bloed en tranen de Business Layer geprogrammeerd hebben.

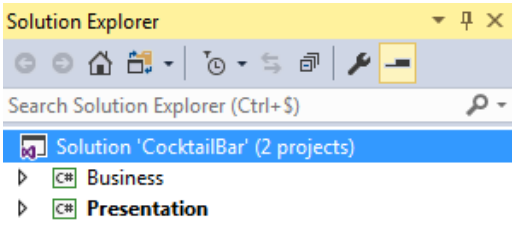
Een cocktail heeft één naam, één prijs, … maar kan **meerdere** ingrediënten bevatten. Dat laatste vertaalt zich in het bovenstaande formulier in een `ListBox` waar alle ingrediënten opgesomd worden.

De `ListBox` neemt in dit hoofdstuk bijgevolg een prominente plaats in.

7.1 De CocktailBar-toepassing

We willen jullie eerst nog eens kort de beginsituatie van de CocktailBar-toepassing voorstellen zoals je deze terugvindt bij de brongegevens van hoofdstuk 7.

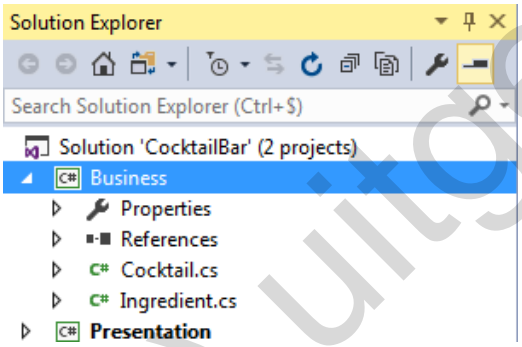
Omdat we bij deze toepassing twee businessklassen en drie formulieren zullen laten samenwerken, hebben we er weer voor geopteerd om de Business - en de Presentation Layer in afzonderlijke projecten onder te brengen.



Om jullie hiervan zo weinig mogelijk 'hinder' te laten ondervinden, hebben wij in het project `Presentation` al een **reference** gelegd naar het project `Business`. Alle nodige using-statements hebben wij al (in elke formulierklasse) ingetikt.

7.1.1 De Business Layer

De Business Layer - of dus het project `Business` - bevat de twee klassen `Cocktail` en `Ingredient`:



Bij 'Oefening 6.1' (vanaf pagina 206) hebben jullie op basis van onze uitvoerige instructies deze beide klassen geprogrammeerd. Omdat we hier niet nog eens zes bladzijden willen reserveren om alle businesscode te printen, beperken we ons tot de velden van deze klassen.

Via de basisklasse `Ingredient` houden we voor elk ingrediënt volgende velden bij:

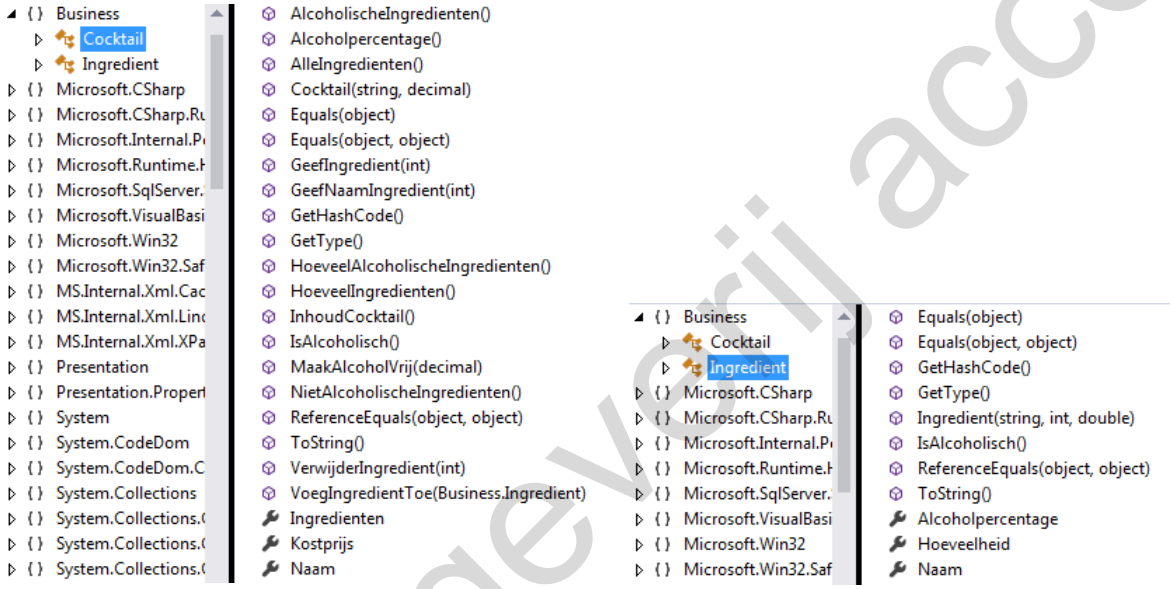
```
public class Ingredient
{
    private String _naam;
    private int _hoeveelheid;
    private double _alcoholpercentage;
}
```

De klasse `Cocktail` beschikt over de volgende velden:

```
public class Cocktail
{
    private String _naam;
    private decimal _kostprijs;
    private List<Ingredient> _ingredienten;
}
```

Een `Cocktail`-object bevat dus naast de `_naam` en de `_kostprijs` nog een veld `_ingredienten` die een **lijst** van `Ingredient`-objecten kan bijhouden!

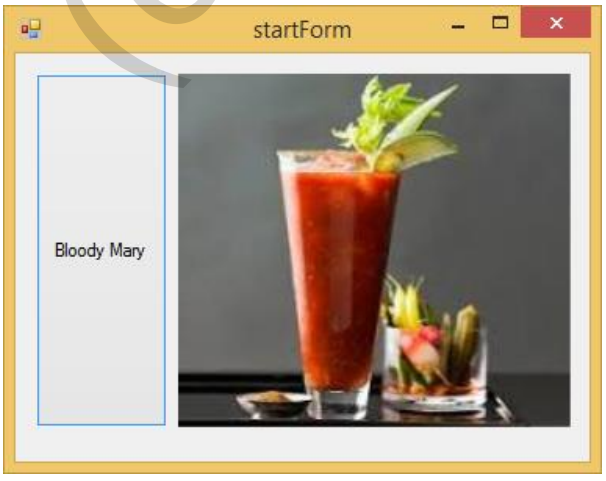
De Object Browser levert trouwens een mooi overzicht van de Properties en methoden die beschikbaar zijn in de klasse `Cocktail` en `Ingredient`:



We verwijzen naar de bestanden **Cocktail.cs** en **Ingredient.cs** voor de volledige klassen–definitie. Wil je wat meer uitleg over een individuele methode moet je maar bladeren naar de betreffende opgave bij 'Oefening 6.1' (vanaf pagina 206).

7.1.2 Het formulier `StartForm`

Het startformulier van het project `Presentation` ziet er als volgt uit:



Belangrijker is de code achter de knop op het formulier:

```
private void bloodyMaryButton_Click(object sender, EventArgs e)
{
    Cocktail cocktail = new Cocktail("Bloody Mary", (decimal)8.25);

    Ingredient i1 = new Ingredient("tomatensap", 100, 0);
    cocktail.VoegIngredientToe(i1);

    Ingredient i2 = new Ingredient("wodka", 50, 0.45);
    cocktail.VoegIngredientToe(i2);

    Ingredient i3 = new Ingredient("citroensap", 8, 0);
    cocktail.VoegIngredientToe(i3);

    CocktailInfoForm formulier = new CocktailInfoForm(cocktail);
    formulier.Show();
}
```

In deze code laten we eerst een nieuw `Cocktail`-object aanmaken. Het vraagt toch zo'n zevental instructies alvorens ons object `cocktail` een volwaardige Bloody Mary geworden is. We moeten immers ook elk van de drie ingrediënten declareren en initialiseren.

Vervolgens laten we een nieuw `CocktailInfoForm`-formulier aanmaken en openen. Merk op dat we bij het aanroepen van de constructor van `CocktailInfoForm` onze "Bloody Mary"-cocktail meegeven als parameter:

```
CocktailInfoForm formulier = new CocktailInfoForm(cocktail);
```

We maken hier gebruik van de techniek op het einde van hoofdstuk 5, waar we een **object** als parameter laten meegeven aan de constructor van een formulier. De constructor van de klasse `CocktailInfoForm` neemt dus een parameter van het type `Cocktail`.

Opmerking:

- De *zeven* instructies om de Bloody Mary samen te stellen, zou je kunnen inkorten tot de vier volgende lijntjes code:

```
Cocktail cocktail = new Cocktail("Bloody Mary", (decimal)8.25);

cocktail.VoegIngredientToe(new Ingredient("tomatensap", 100, 0));
cocktail.VoegIngredientToe(new Ingredient("wodka", 50, 0.45));
cocktail.VoegIngredientToe(new Ingredient("citroensap", 8, 0));
```

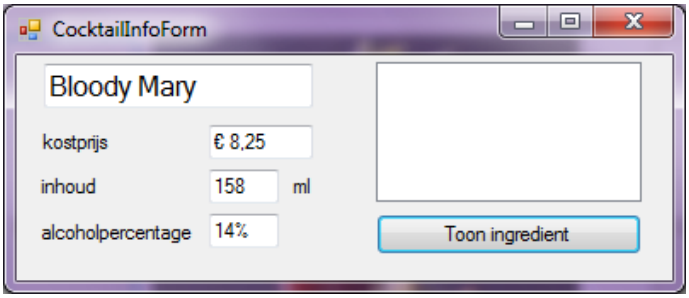
Sleep het formulier `StartForm` eens wat breder … Je merkt dat er op dit formulier nog een knop en een afbeelding van een Cuba Libre staan.

Jullie moeten zorgen dat het formulier ook voor de Cuba Libre werkt. Dit vraagt twee stappen:

1. Laat achter het betreffende knopje een `Cocktail`-object aanmaken, dat je volledig instelt als een 'Cuba Libre'-cocktail (zie pagina 200).
2. Laat er dan een nieuw `CocktailInfoForm`-formulier aanmaken (en tonen), waarbij je het 'Cuba Libre'-object meegeeft als parameter.

7.1.3 Het formulier `CocktailInfoForm`

Open je `CocktailInfoForm` via het knopje 'Bloody Mary' op `StartForm` dan krijg je volgende gegevens te zien:



In het formulier `CocktailInfoForm` tonen we m.a.w. alle info over een bepaalde cocktail.

Het is dus helemaal niet vreemd dat we in dit formulier een businessobject van de klasse `Cocktail` gebruiken. We noemen dit betreffende **veld** `_cocktail`:

```
public partial class CocktailInfoForm : Form
{
    private Cocktail _cocktail;
}
```

De **constructor** van `CocktailInfoForm` wordt aangestuurd door volgende code:

```
public CocktailInfoForm(Cocktail cocktail)
{
    InitializeComponent();

    _cocktail = cocktail;

    naamTextBox.Text = _cocktail.Naam;
    kostprijsTextBox.Text = _cocktail.Kostprijs.ToString("C");
    inhoudTextBox.Text = _cocktail.InhoudCocktail().ToString();
    alcoholpercentageTextBox.Text = _cocktail.Alcoholpercentage().ToString("P0");
}
```

In de constructor laten we als eerste ons veld `_cocktail` initialiseren.

Een belangrijk punt is dat we **via een parameter** het `Cocktail`-object meegeven dat we willen gebruiken in het formulier.

We hoeven deze keer in de constructor van het formulier dus geen nieuw `Cocktail`-object meer aan te maken, maar we laten ons `_cocktail`-veld wijzen naar het object dat we meekrijgen via de parameter.

```
public CocktailInfoForm(Cocktail cocktail)
{
    InitializeComponent();

    _cocktail = cocktail;

    ...
}
```

Door in ons startformulier `startForm`, bij het klikken op de eerste knop, een Bloody Mary samen te stellen en deze mee te geven als parameter aan ons `CocktailInfoForm`-formulier, zal het veld `_cocktail` dus onmiddellijk ingesteld zijn op een 'Bloody Mary'-cocktail.

Zo konden we het formulier `CoctailInfoForm` heel flexibel maken! Je kan in dit formulier om het even welke cocktail presenteren! Je hoeft de gewenste cocktail gewoon mee te geven als parameter aan het formulier!

In de constructor zorgen we met onderstaande instructies dat we meteen de juiste gegevens in de tekstvakken tonen:

```
naamTextBox.Text = _cocktail.Naam;
kostprijsTextBox.Text = _cocktail.Kostprijs.ToString("C");
inhoudTextBox.Text = _cocktail.InhoudCocktail().ToString();
alcoholpercentageTextBox.Text = _cocktail.Alcoholpercentage().ToString("P0");
```

Via de eigenschappen (`Naam` en `Kostprijs`) en de methoden (`InhoudCocktail()` en `Alcoholpercentage()`) vragen we hiervoor aan ons businessobject `_cocktail` de nodige informatie op.

Opmerking:

- Via `ToString("C")` en `ToString("P0")` zetten we de kostprijs om naar Valuta-notatie en laten we het alcoholpercentage weergeven als een procent zonder cijfers na de komma.

De `ListBox` op het formulier `CocktailInfoForm` blijft nu nog leeg. Verder in dit hoofdstuk pakken we dit aan.

7.1.4 Het formulier `IngredientForm`

Als het formulier `CocktailInfoForm` dient om de gegevens van een bepaald `Cocktail`-object weer te geven, dan kunnen we met `IngredientForm` ditzelfde doen voor een `Ingredient`-object.

Het formulier `IngredientForm` werd hiervoor als volgt geprogrammeerd:

```
public partial class IngredientForm : Form
{
    private Ingredient _ingredient;

    public IngredientForm(Ingredient ingredient)
    {
        InitializeComponent();

        _ingredient = ingredient;

        naamTextBox.Text = _ingredient.Naam;
        hoeveelheidTextBox.Text = _ingredient.Hoeveelheid.ToString();
        alcoholpercentageTextBox.Text =
                                    _ingredient.Alcoholpercentage.ToString("P0");
    }
}
```

In deze code onderscheiden jullie moeiteloos volgende stappen:

- We geven aan de constructor van het formulier een `Ingredient`-parameter mee.
- We gebruiken deze parameter om het businessobject `_ingredient` te initialiseren.
- We vragen aan het businessobject `_ingredient` één en ander op om dit, met het juiste formaat, weer te geven in de tekstvakken op het formulier.

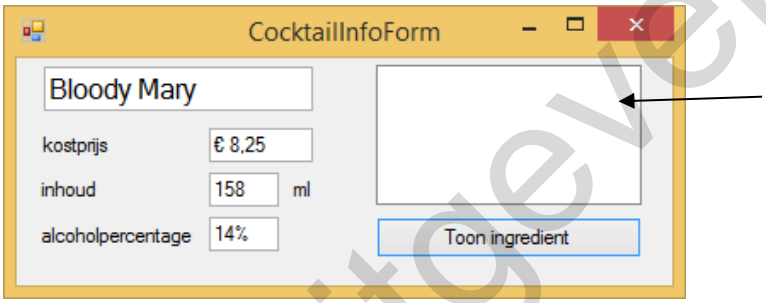
Omdat de CocktailBar-toepassing in deze fase nog niet helemaal af is, kan je het formulier `IngredientForm` voorlopig nog niet aan het werk zien. We wilden bovenstaande code wel al eens bespreken, zodat je je kan voorstellen wat dit formulier gaat doet.

7.2 `List`-bewerkingen in de presentatielaag

Nu jullie een beetje wegwijs gemaakt zijn in de CocktailBar-toepassingen, gaan we samen onderzoeken op welke wijze we lijsten (een `List`) kunnen verwerken in formulieren.

7.2.1 Een collectie toewijzen aan een `ListBox`

In `CocktailInfoForm` blijft het lijstje met de ingrediënten voorlopig nog leeg:



Bedoeling is dat we in dit vak alle ingrediënten van de cocktail opsommen. Dit zijn dus de ingrediënten die we bijhouden in een `List`.

De eigenschap `DataSource` van een `ListBox`

Het betreffende vak rechts bovenaan op het formulier is een `ListBox`. Wij sleepten deze `ListBox` vanuit de Toolbox op het formulier. In het Properties venster kreeg dit element van ons de naam `ingredientenListBox`.

Zo'n `ListBox` is een besturingselement waarmee je **meerdere objecten** kan voorstellen. Een `ListBox` toont geen afzonderlijk *getalletje* of *tekstje*, nee een `ListBox` dient om de elementen uit een collectie te presenteren. Daarom dat je aan een `ListBox` rechtstreeks een collectie kan toewijzen. Dit gebeurt via de eigenschap `DataSource`.

We krijgen dus een instructie van de vorm:

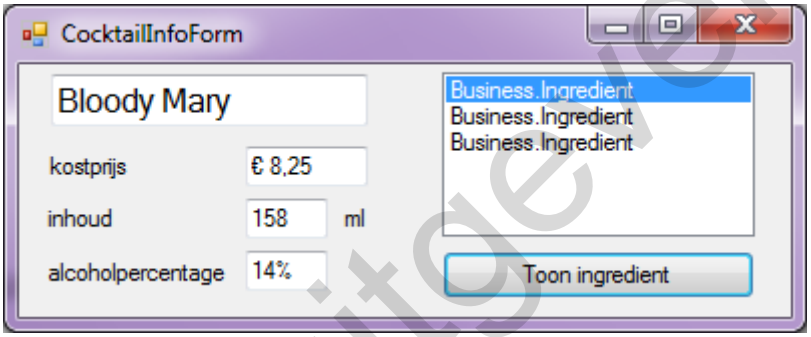
```
listbox.DataSource = collectie;
```

Via de Property `Ingredienten` kunnen we aan een `Cocktail`-object de volledige lijst met ingrediënten opvragen. `_cocktail.Ingredienten` is m.a.w. de collectie die we in onze `ListBox` aan de gebruiker willen presenteren. Merk dus op dat `Ingredienten` een Property is van het type `List<Ingredient>`.

We besluiten daarom om aan de constructor van ons formulier `CocktailInfoForm` onderstaande vette instructie toe te voegen:

```
public CocktailInfoForm(Cocktail cocktail)
{
    InitializeComponent();

    _cocktail = cocktail;

    naamTextBox.Text = _cocktail.Naam;
    kostprijsTextBox.Text = _cocktail.Kostprijs.ToString("C");
    inhoudTextBox.Text = _cocktail.InhoudCocktail().ToString();
    alcoholpercentageTextBox.Text = _cocktail.Alcoholpercentage().ToString("P0");

    ingredientenListBox.DataSource = _cocktail.Ingredienten;

}
```

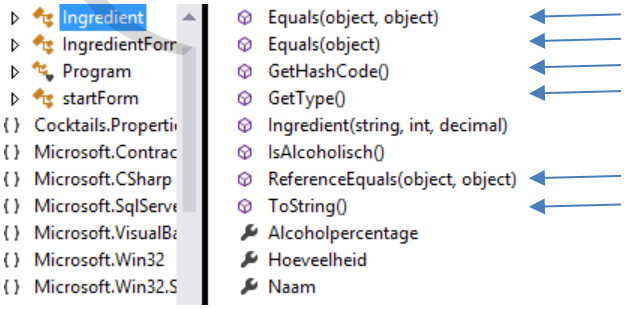
Hoopvol starten we het project opnieuw op, maar onderstaande screenshot toont dat we er nog niet zijn. In de `ListBox` wordt helemaal geen nuttige informatie geprint:



Er ontbreekt nog een puzzelstukje ...

De methode `ToString()`

Is het jullie al opgevallen dat je in de Object Browser bij bijvoorbeeld de klasse `Ingredient` een aantal methoden ziet staan die wij niet zelf gedefinieerd hebben in de klassendefinitie:



Dit zijn methoden die **elke klasse** automatisch **overerft** van de superklasse `Object` (**overerving** kwam in de cursus BlueJ aan bod in hoofdstuk 7). Elke klasse in C# beschikt op die

manier **automatisch** over een methode `ToString()` die bepaalt hoe het object omgezet moet worden naar een `String`.

Visual Studio kan uit zichzelf niet weten hoe van een `Ingredient`-object een `String` moet gemaakt worden, dus wordt van ons verwacht dat wij dit zelf programmeren.

We gaan aan de slag in de klasse `Ingredient` en voegen er volgende methode `ToString()` toe:

```
public override string ToString()
{
    return _hoeveelheid + " ml " + _naam;
}
```

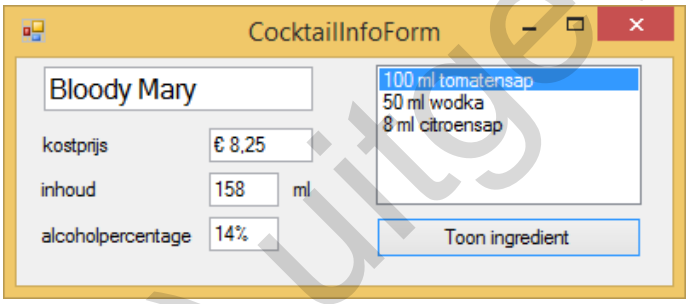
Omdat `ToString()` in de klasse `Ingredient` een methode is met een speciale status (een overgeërfde methode), moeten we bij de header het sleutelwoord **override** vermelden. We geven hiermee aan dat we de reeds *bestaande* methode `ToString()` gaan **overschrijven**.

```
public override string ToString()
```

Met de return regel leggen we vast dat de methode `ToString()` een tekst retourneert van de vorm "xxx ml yyy" waarbij xxx de hoeveelheid en yyy de naam van het ingrediënt voorstelt.

```
return _hoeveelheid + " ml " + _naam;
```

Dan moet je nu de Bloody Mary eens in het formulier `CocktailsInfoForm` openen:



Hip hip hoera, daar zijn de ingrediënten!

Onthoud 7.1 Je kan in een `ListBox` de elementen uit **een collectie** laten weergeven. Dit gebeurt via een instructie van de vorm:

`listbox``.DataSource = `*`collectie`*`;`

Met hierbij:

- *`listbox`* -> de `ListBox` waarin je de objecten wilt tonen;
- *`collectie`* -> de collectie (bij ons een `List`) waarvan je de objecten wilt tonen.

In de `ListBox` zal elk afzonderlijk object dan weergegeven worden met de retourwaarde die de methode `ToString()` oplevert.

En hieronder nog eens samenvatten waarop je moet letten als je zelf de methode `ToString()` gaat programmeren (of dus *overschrijven*):

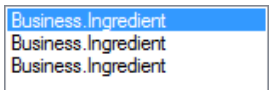
Onthoud 7.2 Elke klasse beschikt door overerving over een methode `ToString()`. Als programmeur kan jij via deze methode `ToString()` vastleggen hoe een object van die klasse omgezet moet worden naar een tekst.

Omdat `ToString()` in een klasse dus een reeds *bestaande* methode is, waarvan jij de functionaliteit wilt **overschrijven**, moet je bij de header het sleutelwoord **override** vermelden:

```
public override string ToString() {}
```

Opmerkingen:

- Toen we de methode `ToString()` nog niet opgenomen hadden in de klasse `Ingredient`, kregen we in onze `ListBox` volgende uitvoer te zien:

  Daaruit kunnen we concluderen dat de methode `ToString()` standaard **het type** van het object retourneert. Dat type is de namespace `Business` en de klasse `Ingredient`.

  Als je de methode `ToString()` dus niet **overschrijft**, werkt ze wel (het formulier loopt niet vast), maar wat geretourneerd wordt, is voor ons niet echt *nuttige* informatie.

  Het is geen slechte gewoonte om in eigen klassen *altijd* een eigen versie van de methode `ToString()` op te nemen. Als onze objecten dan ooit in een `ListBox` terechtkomen (of één van de vele andere besturingselementen waaraan je collecties kan toewijzen), worden ze er alvast al *verzorgd* weergegeven.

- Als je een eigen klasse van een methode `ToString()` voorziet, doe je hiermee in het **venster Immediate** ook profijt.

  We maken ons punt aan de hand van een voorbeeldje.

  In onderstaande screenshot laten we een nieuw `Cocktail`-object instellen en voegen er een drietal ingrediënten aan toe:

  ```
  Immediate Window
  ? Business.Ingredient i1 = new Business.Ingredient("tomatensap", 100, 0);
  ? Business.Ingredient i2 = new Business.Ingredient("wodka", 50, 0.40);
  ? Business.Ingredient i3 = new Business.Ingredient("citroensap", 8, 0);
  ? Business.Cocktail c = new Business.Cocktail("Bloody Mary", (decimal) 8.25);
  ? c.VoegIngredientToe(i1);
  ? c.VoegIngredientToe(i2);
  ? c.VoegIngredientToe(i3);
  ```

  Met de Property `Ingredienten` kunnen we (ook in het venster Immediate) de ingrediëntenlijst opvragen aan ons `Cocktail`-object `c`.

Vooraleer de methode `ToString()` aan de klasse `Ingredient` toegevoegd werd, zou deze lijst met ingrediënten er als volgt uitzien:

```
? c.Ingredienten
Count = 3
    [0]: {Business.Ingredient}
    [1]: {Business.Ingredient}
    [2]: {Business.Ingredient}
```

Met de uitgewerkte methode `ToString()` verandert dit naar:

```
? c.Ingredienten
Count = 3
    [0]: {100 ml tomatensap}
    [1]: {50 ml wodka}
    [2]: {8 ml citroensap}
```

De tekst die tussen de accolades geprint wordt, is m.a.w. ook de retourwaarde van de methode `ToString()`.

Door de methode `ToString()` te overschrijven, heb je voortaan dus zelf in de hand welke tekst er daar getoond wordt. Op deze manier kan je natuurlijk veel beter opvolgen wat er in de `List` gebeurt!

Zelfs bij de instructie waarmee je een object declareert en initialiseert, merk je in het venster Immediate al een verschil!

Zonder de methode `ToString()`:

```
? Business.Ingredient i4 = new Business.Ingredient("cola", 100, 0);
{Business.Ingredient}
    _alcoholpercentage: 0.0
    hoeveelheid: 100
```

Met onze methode `ToString()`:

```
? Business.Ingredient i4 = new Business.Ingredient("cola", 100, 0);
{100 ml cola}
    _alcoholpercentage: 0.0
    hoeveelheid: 100
```

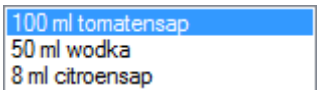
7.2.2 Het geselecteerde object in de `ListBox`

De uitleg hierboven leerde ons dat we in een `ListBox` in feite `ToString()`-tekstjes *te zien* krijgen. De essentie die je niet uit het oog mag verliezen, is echter dat een `ListBox` opgevuld wordt met *volwaardige* **objecten**!

Wij lieten met de onderstaande instructie (zie de constructor in `CocktailInfoForm`) alle objecten die in de ingrediëntenlijst van de cocktail zitten, toevoegen aan de `ListBox` `ingredientenListBox`:

```
ingredientenListBox.DataSource = _cocktail.Ingredienten;
```

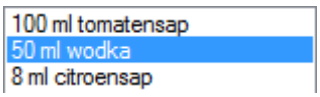
Het resultaat in ons formulier was dat hiermee in `ingredientenListBox` een drietal `Ingredient`-objecten geduwd worden.



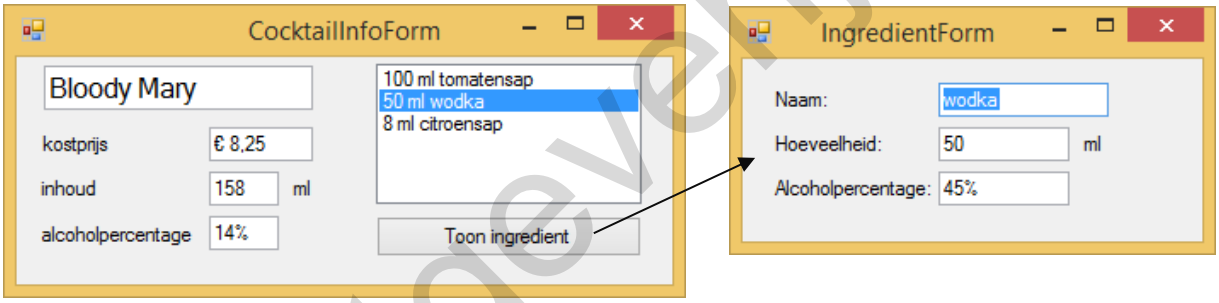
Als je in de `ListBox` een **item selecteert**, moet je beseffen dat je hiermee dus **een volledig `Ingredient`-object** aanduidt (en niet slechts een tekstje)!

Bij een `ListBox` kan je met de eigenschap `SelectedItem` opvragen welk element geselecteerd is. De Property `SelectedItem` levert je dus een **object** op.

Bij onderstaand voorbeeldje spreken we met `ingredientenListBox.SelectedItem` niet het tekstje "50 ml wodka" aan, maar het volledige `Ingredient`-object!



Achter de knop 'Toon ingredient' willen we nu code schrijven waarmee we het geselecteerde ingrediënt laten weergeven in ons formulier `IngredientForm`.



Herinner dat je aan `IngredientForm` als parameter een `Ingredient`-object moet meegeven. Van dit `Ingredient`-object ga je dan de gegevens te zien krijgen in de tekstvakken.

Achter de knop 'Toon ingredient' plaats je hiervoor volgende code:

```
private void toonButton_Click(object sender, EventArgs e)
{
    Ingredient ingredient = (Ingredient) ingredientenListBox.SelectedItem;

    IngredientForm formulier = new IngredientForm(ingredient);
    formulier.Show();
}
```

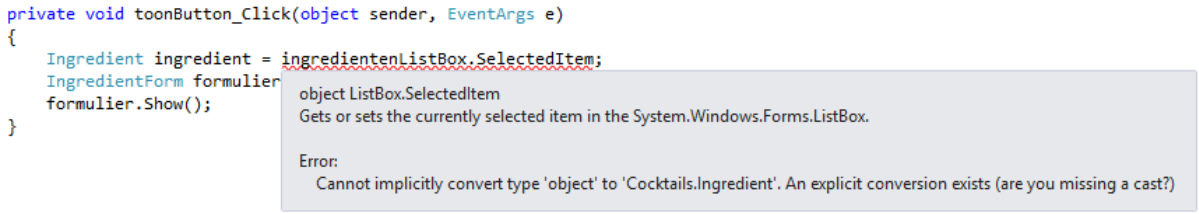
Dit vraagt weer een woordje uitleg.

Met de expressie `ingredientenListBox.SelectedItem` duid je, zoals je ondertussen weet, het object aan dat geselecteerd werd in de `ListBox ingredientenListBox`.

Wij willen een nieuwe hulp-variabele laten wijzen naar dit geselecteerde object. We dachten hierbij *eerst* aan de instructie:

```
Ingredient ingredient = ingredientenListBox.SelectedItem;
```

Er volgt echter protest van Visual Studio:



Wat is er aan de hand:

- Als je met de eigenschap `SelectedItem` het geselecteerde item in een `ListBox` opvraagt, dan krijg je blijkbaar een object terug van het type `Object`. De eigenschap `SelectedItem` weet m.a.w. niet dat het hier om een `Ingredient` gaat.
- Aan de linkerkant van het =-teken declareerden we echter een variabele van het type `Ingredient`.

Visual Studio redeneert dat het NIET mogelijk is om iets van het type `Object` toe te wijzen aan een variabele van het type `Ingredient`. We zitten m.a.w. weer met een conversie-probleem.

We lossen de patstelling op door het geselecteerde item in de `ListBox` expliciet om te zetten van het type `Object` náár het type `Ingredient` en dat kan gelukkig met de korte ()-notatie.

De instructie wordt dus:

```
Ingredient ingredient = (Ingredient) ingredientenListBox.SelectedItem;
```

Het resultaat is dat onze variabele `ingredient` nu wijst naar het geselecteerde ingrediënt in de `ListBox`.

Daarna laten we een nieuw `IngredientForm`-formulier maken en geven hierbij de variabele `ingredient` (of dus het geselecteerde ingrediënt) mee als parameter.

```
IngredientForm formulier = new IngredientForm(ingredient);
formulier.Show();
```

Onthoud 7.3 Met de Property `SelectedItem` duid je het geselecteerde object in een `ListBox` aan!

Opgelet: `SelectedItem` levert je altijd een resultaat op van het type `Object`! `SelectedItem` weet dus niet wat het oorspronkelijke type is van de elementen die je in de `ListBox` stopte.

Hierdoor is de kans groot dat je nog een extra conversie zal moeten uitvoeren! Dit kan met de korte ()-notatie.

Opmerking:

- De code waarmee we het geselecteerde ingrediënt laten weergeven in het ingrediënten–formulier kan iets korter door er geen gebruik meer te maken van een aparte `Ingredient`-hulpvariabele:

```
IngredientForm formulier =
          new IngredientForm((Ingredient) ingredientenListBox.SelectedItem);
formulier.Show();
```

Merk op dat we het geselecteerde item in de `ListBox` nog steeds moeten converteren naar het type `Ingredient`.

En omdat we nu toch bezig zijn … eigenlijk kan dit alles met één lijntje code als we ook onze `IngredientForm` variabele `formulier` niet meer expliciet gaan declareren:

```
(new IngredientForm((Ingredient) ingredientenListBox.SelectedItem)).Show();
```

Oordeel zeker zelf, maar voor ons wordt het zo toch wat onoverzichtelijk.

7.2.3 Een `ListBox` vernieuwen

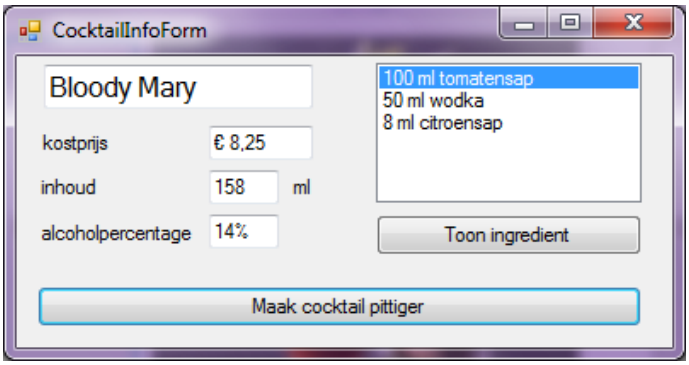
Aan de klasse `Cocktail` hebben wij nog een extra methode `void MaakPittiger()` toegevoegd. In onderstaande oefening onderzoeken jullie wat deze methode precies doet.

Oefening 7.1 Stel onderstaande methode `void MaakPittiger()` in de klasse `Cocktail`.

```
public void MaakPittiger()
{
    foreach (Ingredient ingredient in _ingredienten)
    {
        if (ingredient.IsAlcoholisch() == true)
        {
           ingredient.Hoeveelheid = (int) ((double) ingredient.Hoeveelheid * 1.10);
        }
        else
        {
           ingredient.Hoeveelheid = (int) ((double) ingredient.Hoeveelheid * 0.90);
        }
    }
}
```

Leg in woorden uit wat deze methode precies doet:

Als je het formulier `CocktailInfoForm` iets hoger maakt, merk je dat we onderaan een knop 'Maak cocktail pittiger' verborgen hadden:



Met deze knop willen we de weergegeven cocktail wat *pittiger* maken. Het is maar logisch dat we voor deze bewerking onze nieuwe methode `MaakPittiger()` zullen bovenhalen.

Voeg aan het formulier een methode toe die reageert op de gebeurtenis `Click` van het knopje `pittigerButton`.

De cocktail die weergegeven wordt op het formulier, houden we bij met het businessobject `_cocktail`. We passen de methode `MaakPittiger()` dus toe op dit object `_cocktail`:

```
private void pittigerButton_Click(object sender, EventArgs e)
{
    _cocktail.MaakPittiger();
}
```

Klikken op de knop laat nu wel ons businessobject `_cocktail` bijwerken (we passen er de verhoudingen van de ingrediënten aan), MAAR … op het formulier zelf zal de gebruiker, enkel met deze instructie, nog niets merken!

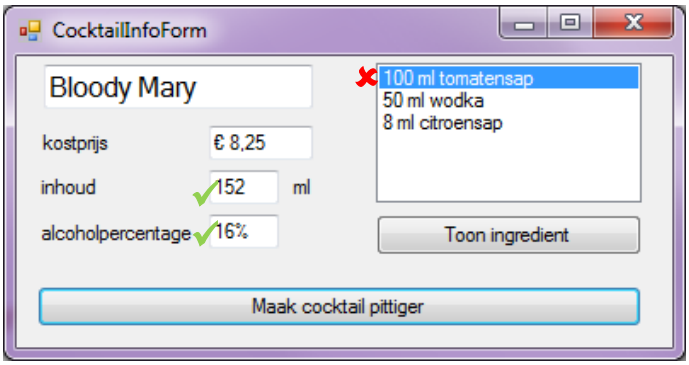
We zien pas een effect als we de ingrediënten in de `ListBox` laten vernieuwen. Omdat we bij het pittiger maken van de cocktail wat met de verhoudingen van de ingrediënten gerom–meld hebben, is het werk pas helemaal af als we ook de tekstvakken met de inhoud en het alcoholpercentage laten bijwerken.

We schrijven hiervoor extra code bij het klikken op het knopje `pittigerButton`:

```
private void pittigerButton_Click(object sender, EventArgs e)
{
    _cocktail.MaakPittiger();

    inhoudTextBox.Text = _cocktail.InhoudCocktail().ToString();
    alcoholpercentageTextBox.Text = _cocktail.Alcoholpercentage().ToString("P0");

    ingredientenListBox.DataSource = _cocktail.Ingredienten;
}
```

In de vette instructies vragen we aan ons `_cocktail`-object de nieuwe inhoud, het nieuwe alcoholpercentage en de lijst van alle (bijgewerkte) ingrediënten nog eens op. We ver–nieuwen de betreffende tekstvakken en de `ListBox` met deze nieuwe waarden.

Als je het formulier `CocktailInfoForm` eens uittest met deze extra code, merk je een probleem! De knop 'Maak cocktail pittiger' laat de tekstvakken wel vernieuwen, maar in de `ListBox` blijven de hoeveelheden per ingrediënt ongewijzigd ;-(.



Je kon het niet weten, maar om een `ListBox` te laten vernieuwen, heb je nog een extra instructie nodig:

```
private void pittigerButton_Click(object sender, EventArgs e)
{
    _cocktail.MaakPittiger();

    inhoudTextBox.Text = _cocktail.InhoudCocktail().ToString();
    alcoholpercentageTextBox.Text = _cocktail.Alcoholpercentage().ToString("P0");

    ingredientenListBox.DataSource = null;
    ingredientenListBox.DataSource = _cocktail.Ingredienten;
}
```

Je wist reeds dat je via de eigenschap `DataSource` een collectie kan toekennen aan een `Listbox`. Het resultaat is dat alle objecten uit de collectie (bij ons is dit een `List`) in de `ListBox` weergegeven worden.

Wil je een `ListBox` **vernieuwen** is het echter **niet voldoende** om de eigenschap `DataSource` opnieuw in te stellen op de nieuwe collectie (bij ons: de bijgewerkte lijst met ingrediënten). Je moet blijkbaar éérst de `ListBox` eens leeg maken(!) en dit kan door de `DataSource` in te stellen op **null**:

```
ingredientenListBox.DataSource = null;
```

We geven toe dat het vervelend is dat het op deze manier moet. Er loert zeker gevaar om de hoek om deze instructie te vergeten, waardoor de `ListBox` niet goed zal werken!

Niemand verbiedt je om dit stukje cursus met fluor in te kleuren en op deze pagina enkele grote rode gevarendriehoeken te tekenen.

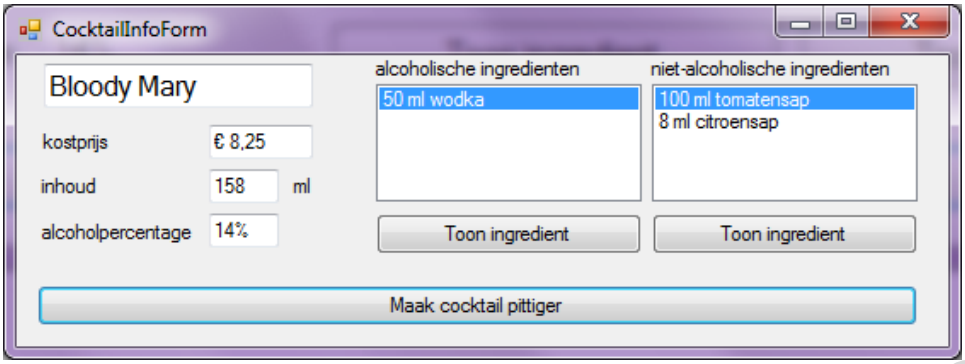
Onthoud 7.4 Alvorens je de inhoud van een `ListBox` kan vernieuwen, moet je eerst deze `ListBox` eens leeg maken.

Vandaar dat onderstaande instructies vaak een koppeltje zullen vormen:

```
listbox.DataSource = null;
listbox.DataSource = collectie;
```

7.3 Even oefenen

Stel dat we op `CocktailInfoForm` de alcoholische en niet-alcoholische ingrediënten van elkaar willen scheiden:



Met deze kleine uitbreiding zullen jullie de belangrijkste `ListBox`-bewerkingen die in dit hoofdstuk aan bod kwamen, nog eens herhalen.

Omdat het bij deze opdracht voornamelijk zaak is om de code die al in het formulier staat, nog eens te klonen, hebben we er alle vertrouwen in dat het niet nodig is om de oplossing hier nog eens te printen.

Als je elk van de onderstaande punten een *check* kan geven, zou het formulier in orde moeten zijn.

- Sleep op het formulier een tweede `ListBox` en een tweede 'Toon ingredient'-knopje. Geef deze elementen een logische naam (en pas eventueel de naam van de andere `ListBox` en `Button` aan).

  Ook de twee nieuwe labeltjes boven de `ListBox`-elementen niet vergeten.

- In de constructor hadden we een instructie staan waar we aan ons `Cocktail`-businessobject **alle** ingrediënten opvroegen via de Property `Ingredienten` om die lijst dan in de `ListBox` te tonen.

  In de ene `ListBox` mogen voortaan **enkel** de alcoholische ingrediënten komen van de cocktail, in de andere **enkel** de niet-alcoholische ingrediënten.

  De klasse `Cocktail` beschikt ook over een methode `AlcoholischeIngredienten()` en een methode `NietAlcoholischeIngredienten()`, waarmee een `List` geretourneerd wordt met enkel die specifieke ingrediënten.

  Lukt het om met beide methoden aan het `Cocktail`-businessobject de lijsten met de betreffende ingrediënten op te vragen om hiermee de twee `ListBox`-elementen op het formulier op te vullen.

- De twee knopjes 'Toon ingredient' zouden aan de `ListBox` erboven moeten opvragen welk ingrediënt er geselecteerd is, om dit ingrediënt vervolgens te tonen in het formulier `IngredientForm`.

❖ De knop 'Maak cocktail pittiger' past de verhouding van alle ingrediënten in de cocktail aan. Na het klikken op dit knopje zou je dus beide `ListBox`-elementen moeten laten updaten.

Niet vergeten dat er twee instructies nodig zijn om een `ListBox` te vernieuwen!

Oefening 7.2 Printburo - Een kopiecentrum staat vaak volgepropt met allerhande draaiende copier-toestellen. In zijn *eenvoudigste* vorm kan je met een copier de volgende taak uitvoeren:

a) je legt er een origineel stapeltje papier op;
b) je geeft aan hoeveel kopies je wilt;
c) je hoopt dat je originele document, in het gevraagde aantal exemplaren, uit één of andere schuif te voorschijn komt rollen.

Moderne copiers kunnen het originele document natuurlijk ook digitiaal inlezen. Je sluit hiervoor bijvoorbeeld fysisch een USB-stick aan waarop het originele pdf-document staat, of je laat het bestand via een pc naar de copier sturen.

Dat er verschillende papierformaten en -diktes zijn, dat je in de praktijk kan kiezen tussen een recto of recto verso druk, … daar gaan we ons in het kader van deze oefening, voor de eenvoud, niet mee bezighouden.

Wat elke copier wel gemeen heeft, is dat er ergens in een hoekje of op één of ander control panel een teller te vinden zal zijn die feilloos bijhoudt hoeveel blaadjes papier er in totaal al geprint zijn.

Voldoende stof om rond dit thema, in de solution `Printburo`, een aardige oefening op te bouwen. In deze solution hebben wij al het één en het ander voorbereid, maar we lieten heel wat gaten over, die jullie mogen opvullen!

Een solution met twee projecten

Onze printburo-toepassing hebben we georganiseerd met een aparte Business - en Presentation Layer. In de Solution Explorer zal je bijgevolg twee afzonderlijke projecten vinden (met de namen `Business` en `Presentation`).

Bij een dergelijke constructie is het zo dat de presentatielaag een beroep zal doen op de businesslaag, of anders geformuleerd: "in de user interface maken we gebruik van de businesslogica". Dit is enkel mogelijk als we in het project met de presentatiecode een link leggen naar het project met de businesscode. Dit is hier nog niet gebeurt.

Jullie eerste opdracht:

❖ Laat in het project `Presentation` een **reference** leggen naar het project `Business`!

De Business Layer

De businessklasse `Printopdracht`

Straks is het de bedoeling om printopdrachten aan een copier door te geven. We stonden daarom eerst even stil bij het concept *printopdracht*.

Elke printopdracht gaat over een origineel document met een gegeven **aantal pagina's** dat een opgegeven **aantal keer geprint** (gekopieerd) moet worden.

We probeerden dit in de klasse `Printopdracht` te modeleren.

Het leverde ons de twee velden `int _paginasOrigineel` en `int _aantalExemplaren` en twee gelijknamige Read only Properties op.

Om een nieuw `Printopdracht`-object aan te maken, geef je aan de constructor via twee parameters, het aantal pagina's van het origineel en het gewenst aantal kopies mee.

In het venster Immediate lieten we als test een printopdracht aanmaken om de eerste 6 hoofdstukken van onze cursus C# (of 206 bladzijden) te kopiëren voor een klas met 20 leerlingen:

```
Immediate Window
? Business.Printopdracht p = new Business.Printopdracht(206,20);
{20 x 206 pagina's}
    _aantalExemplaren: 20
    _paginasOrigineel: 206
    AantalExemplaren: 20
    PaginasOrigineel: 206
```

Opmerking:

- Omdat de solution opgebouwd is uit verschillende projecten/namespaces moesten we in het venster Immediate de namespace `Business` vóór de klasse `Printopdracht` vermelden.

  In de Solution Explorer moet het betreffende project `Business` geselecteerd staan.

De businessklasse `Copier`

Met de klasse `Copier` gaan we de *apparaten* die de printopdrachten zullen verwerken, beschrijven.

We bedenken dat je *meerdere* printopdrachten in serie naar een copier kan sturen. Daarom willen we de printopdrachten in een **een wachtrij** bijgehouden. Vandaar dat we in de klasse `Copier` een veld `_wachtrij` declareerden, en dit als een `List` van `Printopdracht`-objecten.

Het veld `int _teller` houdt bij hoeveel blaadjes reeds **in totaal** uit de copier rolden.

Voor elk van deze velden staat in de klasse `Copier` een gelijknamige Read only Property. Merk dus op dat de Property `Wachtrij` van het type `List<Printopdracht>` moest zijn.

In de constructor bepalen we dat bij een nieuw `Copier`-object de teller op 0 gezet wordt en dat de wachtrij nog leeg is. In de screenshot hieronder zie je dat onze constructor geen parameters neemt:

```
Immediate Window
? Business.Copier c = new Business.Copier();
{Business.Copier}
    _teller: 0
    _wachtrij: Count = 0
    Teller: 0
    Wachtrij: Count = 0
```

In de klasse `Copier` heb je van de twee methoden `VoegPrintopdrachtToe()` en `Print()` al een header staan. De body ontbreekt nog.

Jullie programmeeropdracht in de businesslaag:

❖ De methode `void VoegPrintopdrachtToe(int,int)`:

Met deze methode laten we een *nieuwe* printopdracht (achteraan) op de wachtrij zetten.

Opgelet: we geven bij deze methode GEEN `Printopdracht` mee als parameter, maar vertellen **hoeveel pagina's** ons origineel document bevat (eerste parameter) en in **hoeveel exemplaren** (tweede parameter) we dit document willen kopiëren.

Het is bij deze methode nog NIET de bedoeling om de teller van de copier aan te passen. Documenten *printen* gebeurt immers in een andere methode.

❖ De methode `void Print()`:

Het idee is dat we met deze methode de **eerste** printopdracht uit de wachtrij ophalen, om deze printtaak zogezegd te *verwerken* (=printen).

In de praktijk betekent dit dat je in deze methode twee taken moet uitvoeren:

- De teller (het veld `_teller`) van de copier moet met het juiste aantal gedrukte bladen verhoogd worden. Ga uit van de situatie waar we altijd recto printen.
- Omdat die eerste printopdracht uit de wachtrij nu verwerkt is, mag je dit betreffende `Printopdracht`-object uit de lijst `_wachtrij` verwijderen.

Opgelet: Als je deze methode uitvoert terwijl de wachtrij helemaal **leeg** is, zou je code wel eens een crash kunnen veroorzaken. Daarom is het zeker een goed idee om zelf een controle in te bouwen die nagaat of er in de lijst `_wachtrij` toch minstens één `Printopdracht`-object zit.

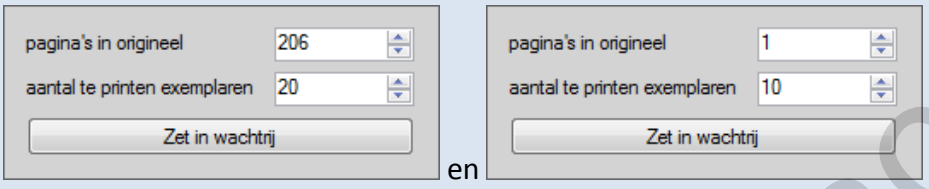
De Presentation Layer

Het formulier `CopierForm`

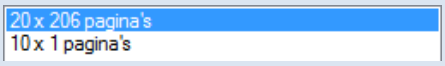
In het formulier `CopierForm` willen we een grafische user interface uitwerken voor een copier-apparaat.

Aan de hand van een voorbeeldje leggen we uit hoe het zou moeten werken:

1. Met de `NumericUpDown`-elementen en de 'Zet in wachtrij'-knop kunnen we print-opdrachten toevoegen aan onze copier. Bijvoorbeeld:

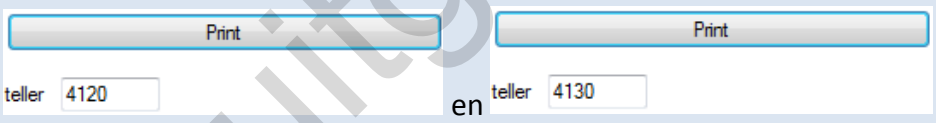


   In de `ListBox` krijg je steeds te zien welke opdrachten (nog) in de wachtrij staan. Na het toevoegen van bovenstaande twee printopdrachten is dit dus:



2. De knop 'Print' laat de eerste printopdracht in de wachtrij verwerken. Het effect is dat dit item uit de wachtrij gehaald wordt en dat we de teller met het aantal geprinte pagina's zien verhogen.

   Klikken we in ons voorbeeld tweemaal op de knop 'Print', dan verspringt de teller respectievelijk naar:



   In deze situatie zal de `ListBox` weer leeg zijn.

Het formulier, of dus de klasse `CopierForm`, moet nog helemaal geprogrammeerd worden.

Hieronder zetten we jullie een beetje op weg. Bij deze uitleg willen we jullie vooral waarschuwen voor enkele adders onder het gras.

❖ Het businessobject in het formulier:

In de businessklasse `Copier` hebben we de logica van onze kopieermachines geprogrammeerd. In ons formulier willen we hierop graag een beroep doen.

We doen dit uiteraard door in de klasse `CopierForm` een veld te declareren van de klasse `Copier`. De meest logische naam voor dit veld is `_copier`.

Het object `_copier` zal dan dienst doen als businessobject in ons formulier en zal dus de toestand van het kopieerapparaat dat we in het formulier *zien*, bijhouden. Met de knoppen op het formulier gaan we dat businessobject `_copier` besturen.

Laat in de constructor dit veld `_copier` initialiseren als een nieuw `Copier`-object.

De adder onder het gras:

- Omdat de klasse `Copier` in een **andere namespace** staat dan het formulier, kan het formulier deze klasse niet zomaar terugvinden (de klasse `Copier` wordt rood onderlijnd).

  Je verhelpt dit als volgt:

  Ofwel vermeld je *elke keer* als je deze klasse `Copier` gebruikt in je formulier, er de namespace `Business` voor (`Business.Copier`).

  Ofwel vermeld je *één maal* bovenaan in de formuliermodule een using naar de namespace `Business` (`using Business`).

- Bij het openen van het formulier:

  Bij het laden van het formulier, zou je in het tekstvak `tellerTextBox` al onmiddellijk de stand van de teller kunnen printen. Dit zal uiteraard nog 0 zijn!

  Dit zou je bij de constructor van het formulier kunnen laten gebeuren.

- Bij de knop 'Zet in wachtrij':

  Met deze knop sturen we zogezegd een nieuwe printopdracht naar onze copier. De informatie over deze printopdracht haal je uit de twee `NumericUpDown`-elementen. De nieuwe printopdracht komt dan in de wachtrij van onze copier terecht.

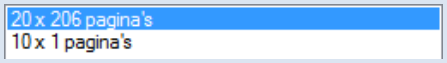
  Je krijgt dit in twee stappen in orde:

  - Het businessobject `_copier` houdt de toestand van ons kopieerapparaat bij! Met de methode `VoegPrintopdrachtToe()` zal je *ten eerste* de printopdracht aan het object `_copier` moeten doorgeven. Kijk zeker nog eens na welke parameters bij deze methode van toepassing zijn.
  - Het businessobject `_copier` laten bijwerken, is echter nog niet voldoende! De nieuwe toestand (de extra printopdracht), zou natuurlijk ook in de `ListBox` met printopdrachten moeten verschijnen.

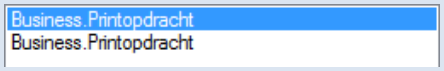
    Aan het businessobject `_copier` kunnen we via de Property `Wachtrij` gemakkelijk de `List` met printopdrachten opvragen. Laat *ten tweede* deze `List` met de juiste instructie in de `ListBox` zetten.

De adders onder het gras:

- In onze voorbeeldscreenshot van de `ListBox` met de wachtrij, zagen de printopdrachten er als volgt uit:

  Zou het kunnen dat jullie nog een genant resultaat krijgen van de volgende vorm:

  Welke methode (bij welke klasse) moeten jullie nog definiëren om de `Printopdracht`-objecten met de juiste notatie in de `ListBox` te krijgen?

  Omdat dit een methode is die een *bijzondere status* inneemt binnen een klasse (je moet de methode namelijk laten **overschrijven**), heb je bij de header nog een speciaal sleutelwoord nodig. Over welk sleutelwoord hebben we het hier?

- Een niet-lege `ListBox` een *nieuwe inhoud* geven, lukt enkel en alleen als je de `ListBox` eerst eens **leeg** laat **maken**! Als in jullie oplossing de eerste printopdracht wel, maar alle volgende NIET in de `ListBox` verschijnen, dan heb je waarschijnlijk tegen dit principe gezondigd.

  Welke instructie gebruik je om de `ListBox` leeg te maken?

❖ Bij de knop 'Print':

Met de knop 'Print' laten we de eerste (bovenste) printopdracht in de wachtrij door onze copier verwerken.

Eigenlijk wordt dit helemaal geregeld via de methode `Print()` die jullie enkele pagina's hiervoor in de businessklasse `Copier` geprogrammeerd hebben. Laat in je code dus gewoon het businessobject `_copier` bijwerken via deze methode `Print()`.

Jullie zijn wel helemaal verantwoordelijk om in het formulier aan de gebruiker de nieuwe toestand te presenteren. Jullie moeten m.a.w. de `Listbox` met de wachtrij en het tekstvak met de teller laten updaten.

Dat er bij het vernieuwen van een `ListBox` een addertje is, hebben we al eens verteld!

Extra - Een printopdracht voorrang geven

Stel nu eens dat terwijl er heel wat (grote) printopdrachten in de wachtrij staan, een ander document heel dringend (zeg maar, liefst onmiddellijk) gekopieerd moet raken. Zoals het formulier `CopierForm` nu geprogrammeerd is, moet die laatste printopdracht ook gewoon op zijn beurt wachten.

Misschien moeten we daarom liever bij onze copiers ook een mogelijkheid inbouwen om een bepaalde printopdracht uit de wachtrij te lichten om deze **prioritair** te printen!

Ons plan is om in de Business Layer de klasse `Copier` wat extra functionaliteit te geven (dus wat slimmer te maken), om dit dan straks nuttig uit te spelen in ons formulier.

❖ Methode `void PrintMetVoorrang(Printopdracht)`

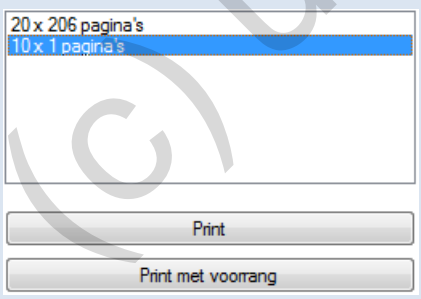
Voeg aan de businessklasse `Copier` deze methode `PrintMetVoorrang()` toe. Als parameter geef je een `Printopdracht`-object (uit de wachtrij) mee.

Deze printopdracht laten we onmiddellijk door de copier verwerken. In deze methode regel je hiervoor twee zaken:

- Laat het veld `_teller` gepast verhogen. Bereken dus hoeveel pagina's gedrukt zullen worden bij de `Printopdracht` die je meekreeg als parameter.
- Zorg dat deze `Printopdracht` uit de lijst `_wachtrij` gehaald wordt. Weet je nog dat we in hoofdstuk 6 bij de `List`-bewerkingen de methode `Remove()` behandeld hebben, waarmee je ineens een opgegeven object uit een `List` verwijdert (als alternatief voor `RemoveAt()`).

❖ Knop 'Print met voorrang':

Op het formulier `CopierForm` voegen we onder de `ListBox` een tweede knop toe:



Met deze nieuwe 'Print met voorrang'-knop laten we niet de eerste opdracht uit de wachtrij printen (zoals bij de 'Print'-knop), maar verwerken we de **geselecteerde** printopdracht in de `Listbox`.

Met bovenstaande screenshot zouden we op die manier dus de geselecteerde 10 blaadjes kunnen laten printen, voordat er vele pakken papier doorgejaagd worden met de kopies van onze C#-cursus.

Wat programmeer je achter de knop 'Print met voorrang':

- In de `ListBox` moeten we identificeren welk element geselecteerd staat. Wij zijn er voorstander van om dit element eerst op te slaan in een hulpvariabele. Via deze hulpvariabele kunnen we dan in het vervolg van onze code gemakkelijk naar het betreffende object verwijzen.

  Declareer dus een `Printopdracht`-hulpvariabele, die je onmiddellijk initialiseert op het geselecteerde element in de `ListBox`.

  Noteer met welke Property je aan een `ListBox` het geselecteerde element opvraagt?

  De adder onder het gras:

  ⌛ Met de Property `SelectedItem` van een `ListBox` krijg je altijd een object terug van het type `Object`? Wij willen dit object echter opslaan in een hulpvariabele van het type `Printopdracht`.

  Dat betekent dat er een conversie tussen gegevenstypen zal moeten gebeuren. Hoe doe je dit?

- Een specifieke printopdracht uitvoeren, hebben we in de businessklasse `Copier` laten regelen met de nieuwe methode `PrintMetVoorrang()`.

  Laat m.a.w. bij ons businessobject `_copier`, via deze methode `PrintMetVoorrang()`, de geselecteerde printopdracht verwerken. Die *geselecteerde printopdracht* zou je ondertussen moeten kunnen aanspreken via een hulpvariabele.

- Tenslotte moeten we in het formulier onze `ListBox` en het tekstvak met de teller nog eens laten vernieuwen.

  Zo krijgt de gebruiker ook het resultaat van zijn/haar prioritaire print op het formulier te zien. De printopdracht die geselecteerd was, zal namelijk verdwenen zijn en de teller van de copier zal weer wat hoger staan.

8 Data Access Layer

De begrippen **Business Layer** en **Presentation Layer** lopen al ruim 200 bladzijden ontegensprekelijk als een rode draad doorheen deze cursus. Hoe je deze twee lagen met elkaar laat samenwerken, kreeg de volle focus.

In de Business Layer programmeren we één (of meerdere) klasse(n) waarmee we de eigenlijke logica van onze toepassing vatten. Om het eens op een alternatieve manier uit te leggen, noemen we dit het *Einstein*-gedeelte van onze code (in de businesslaag zit vaak het echte genie van de toepassing).

Naar analogie kunnen we de Presentation Layer dan als het *Coco Chanel*-gedeelte van de toepassing beschouwen. Deze presentatielaag bestaat enerzijds uit één (of meerdere) formulier(en) met de nodige besturingselementen en anderzijds de bijhorende programmacode (formulieren zijn ook klassen) waarmee we een user interface opbouwen, zodat het formulier op de nodige **Events** (**gebeurtenissen**) reageert.

Coco Chanel zorgt dat alles op het juiste moment (bijvoorbeeld er wordt op een knop geklikt), met het juiste kleurtje, op de juiste plaats (bijvoorbeeld in een tekstvak) op het formulier getoond wordt. Maar telkens als iets *berekend* moet worden, zal *Coco Chanel* dit heel vriendelijk aan *Einstein* delegeren.

In dit hoofdstuk willen we hierboven een **derde laag** introduceren, meer bepaald de **Data Access Layer** (of de **data-accesslaag**).

In de praktijk zal een toepassing vaak met gegevens werken die **blijvend bewaard moeten worden**. Als je een spelletje afsluit dan mag jouw eervolle vermelding op het scorebord niet verloren gaan; als je in je persoonlijke cocktail-app een nieuwe cocktail samenstelt dan wil je die later opnieuw kunnen raadplegen; enzovoort ...

De voor de hand liggende oplossing, die jullie ook zelf konden bedenken, is om deze gegevens in een **databank** te stockeren. De verantwoordelijkheid van de nieuwe Data Access Layer is om een verbinding met zo'n databank te leggen en om de communicatie met de databank te verzorgen.

Dit houdt twee richtingen in, nl. gegevens ophalen **uit** en gegevens wegschrijven **naar de databank**.

Het zal geen verrassing zijn dat we hiervoor het, voor jullie vertrouwde, databanksysteem **MySQL** zullen gebruiken. Jullie kunnen in dit hoofdstuk dus zeker enkele screenshots vanuit **MySQL WorkBench** verwachten.

De toepassingen die we in dit hoofdstuk behandelen, zullen bezijdens de nieuwe Data Access Layer nog steeds een traditionele Business en Presentation Layer bevatten.

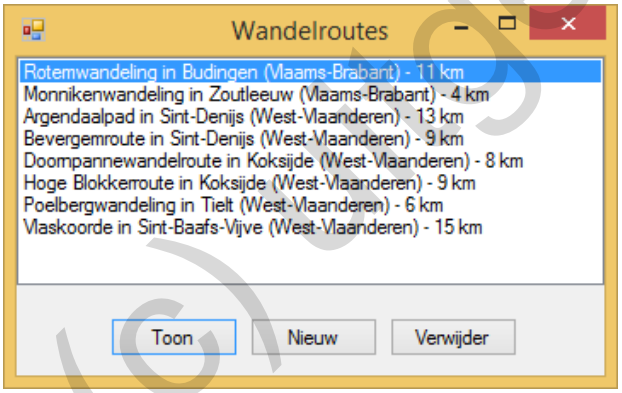
Het tweelagenmodel, dat we tot nu in deze cursus zo hard gepromoot hebben, is op deze manier gepromoveerd naar een **tweelagenmodel++** 😉, of laat ons dit - zoals we het ook deden bij de titel van deze cursus - het **drielagenmodel** noemen.

Omdat de programmacode in een data-accesslaag al voldoende complex is en het wel wat tijd zal vragen om de bijhorende basistechnieken onder de knie te krijgen, zullen we om jullie niet af te leiden, in de komende toepassingen de Business Layer bewust heel eenvoudig houden. Daar plannen we even geen ingewikkelde berekeningen meer.

8.1 De solution `OpStap`

Alvorens we ons volop op de Data Access Layer storten, willen we eerst een nieuwe toepassing introduceren. De solution `OpStap` (**_hoofdstuk_8/_voorbeelden**) is op dit moment nog niet gelinkt aan een databank en hoort daarom (voorlopig nog) helemaal thuis in hoofdstuk 7 van deze cursus.

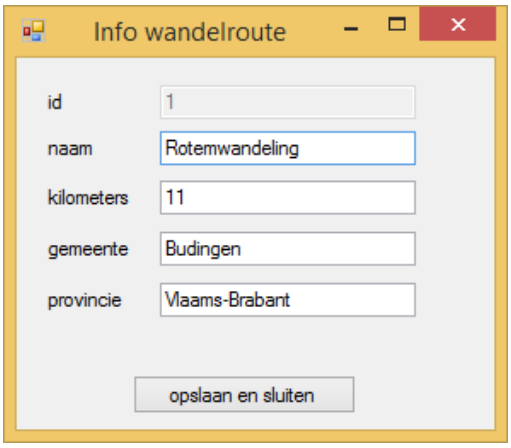
Als je het project opstart, vertelt het startformulier (`WandelroutesForm`) onmiddellijk wat je mag verwachten:



Je krijgt een lijst (`List`) van wandelroutes voorgeschoteld, die verzameld staan in een `ListBox`.

Via de knoppen 'Toon', 'Nieuw' en 'Verwijder' kan je naar wens deze wandelroutes in detail raadplegen (en aanpassen); kan je de lijst uitbreiden met nieuwe wandelroutes en kan je wandelroutes uit het overzicht laten schrappen.

De gegevens van een individuele wandelroute wijzigen of een nieuwe wandelroute aanvullen, gebeurt dan via het formulier `WandelrouteInfoForm`:



Dat er nog geen databank van doen is, merk je aan het feit dat alle wijzigingen die je aan de wandelroutes aanbrengt, verloren gaan bij het afsluiten van het project! Elke keer als je het project opnieuw opstart, is alles weer gereset naar de oorspronkelijke acht wandelroutes.

Straks zullen we de toepassing vertimmeren zodat de wandelroutes wel gelinkt worden aan een databank. De communicatie met de databank in de **Data Access Layer** regelen, vormt het titanenwerk van dit hoofdstuk.

Maar toch eerst eens verkennen hoe de toepassing, nog zonder databank, opgebouwd is …

8.1.1 De klasse `wandelroute`

Een basisbouwsteen in deze toepassing is de klasse `Wandelroute` (zie het project `Business`). `Wandelroute` is de businessklasse die de gegevens van een individuele wandelroute beschrijft.

Elke wandelroute geven we een uniek nummer (veld `int _id`); krijgt een naam (veld `String _naam`); heeft een afstand (veld `int _kilometers`) en is gesitueerd op een bepaalde locatie (veld `String _gemeente` en `String _provincie`).

Bij de klasse `Wandelroute` hoort dit klassendiagram:

Wandelroute

`int _id`
`String _naam`
`int _kilometers`
`String _gemeente`
`String _provincie`

`Wandelroute(int,String,int,String,String)`
Property: `int Id` Property: `String Naam` Property: `int Kilometers` Property: `String Gemeente` Property: `String Provincie` Methode: `string ToString()`

Bij de constructor geven we via vijf parameters de waarden mee die aan de vijf velden van een nieuw `Wandelroute`-object toegekend moeten worden.

Merk ook op dat we voor elk veld in de klasse een eigen Property voorzien hebben.

Misschien toch nog even de aandacht vestigen op de methode `string ToString()`:

```
public override string ToString()
{
    return _naam + " in " + _gemeente + " (" + _provincie + ") - " +
           _kilometers.ToString() + " km";
}
```

Omdat we de wandelroutes op het formulier `WandelroutesForm` in een `ListBox` voorstellen, moesten we via deze methode `ToString()` eerst bepalen hoe we er zo'n individuele wandelroute willen presenteren.

8.1.2 De wandelroutes in een `ListBox`

Bij het openen van het formulier `WandelroutesForm` (zie het project `Presentation`) krijgen we onmiddellijk een achttal vaste wandelroutes te zien. We regelden dit als volgt:

- In het formulier `WandelroutesForm` declareerden we het veld `_wandelroutesLijst` als een `List` van `Wandelroute`-objecten:

  ```
  private List<Wandelroute> _wandelroutesLijst;
  ```

  We mogen gerust stellen dat `_wandelroutesLijst` het belangrijkste (business)object is in ons project. Het veld `_wandelroutesLijst` zal alle wandelroutes bevatten die we voorstellen in de `ListBox`. Als we in het formulier nieuwe wandelroutes willen opnemen of we willen oude wandelroutes verwijderen, dan zullen we dit via het veld `_wandelroutesLijst` moeten regelen.

- Bij het aanmaken van een nieuw `WandelroutesForm`-formulier (of m.a.w. bij de constructor van deze klasse), voeren we enkele acties uit op onze lijst `_wandelroutesLijst`:

  ```
  public WandelroutesForm()
  {
      InitializeComponent();

      // het veld _wandelroutesLijst initialiseren
      _wandelroutesLijst = new List<Wandelroute>();

      // via de hulpmethode LijstOpvullen() laten we het veld _wandelroutesLijst
      // opvullen met enkele wandelroutes
      LijstOpvullen();

      // de lijst met wandelroutes weergeven in de listbox
      wandelroutesListBox.DataSource = _wandelroutesLijst;
  }
  ```

Je herkent hierboven zeker de instructie waar we `_wandelroutesLijst` initialiseren als een `List` van `Wandelroute`-objecten, alsook de instructie waar we via de Property `DataSource` de lijst `_wandelroutesLijst` laten weergeven in de `ListBox`.

De hulpmethode `void LijstOpvullen()` is op zijn beurt verantwoordelijk om de initiële acht wandelroutes in de lijst te stoppen. Dit gebeurt met code van de volgende vorm:

```
private void LijstOpvullen()
{
    // wandelroutes aanmaken en aan de lijst _wandelroutesLijst toevoegen
    Wandelroute w1 = new Wandelroute(1, "Rotemwandeling", 11, "Budingen",
                                     "Vlaams-Brabant");
    Wandelroute w2 = new Wandelroute(2, "Monnikenwandeling", 4, "Zoutleeuw",
                                     "Vlaams-Brabant");
    ...

    _wandelroutesLijst.Add(w1);
    _wandelroutesLijst.Add(w2);
    ...
}
```

8.1.3 De details van een wandelroute tonen

Met de knop 'Toon' krijg je in een nieuw formulier `WandelrouteInfoForm` de details te zien van de wandelroute die geselecteerd stond in de `ListBox`.

De methode die reageert op de gebeurtenis `Click` van deze knop bevat hiervoor volgende code:

```
private void toonButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute in de ListBox opvragen
    Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;

    // het formulier WandelrouteInfoForm openen met de geselecteerde wandelroute
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // misschien heeft gebruiker gegevens van de wandelroute veranderd
    // --> ListBox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Via de eigenschap `SelectedItem` kunnen we een verwijzing opvragen naar de `Wandelroute` die geselecteerd staat in de `ListBox`. In hoofdstuk 7 hebben we gezien dat bij dergelijke instructie een conversie (hier naar het type `Wandelroute`) nodig is:

```
Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;
```

We maken vervolgens een nieuw `WandelrouteInfoForm`-formulier aan. Dit is het formulier waar we de details van een wandelroute zien. Bij de constructor van dit formulier, moeten we als parameter een `Wandelroute`-object meegeven. In onze situatie is dit de wandelroute die geselecteerd stond in de `ListBox`.

```
WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
```

Straks bestuderen we nog welke programmacode het formulier `WandelrouteInfoForm` aanstuurt.

Om het formulier dan op het scherm te brengen, introduceren we een nieuwigheid. We doen dit nl. met de methode `ShowDialog()` (i.p.v. met de klassieke `Show()`):

```
formulier.ShowDialog();
```

De methode `ShowDialog()` zorgt, net zoals de methode `Show()`, dat het formulier op het scherm getoond wordt. Het speciale is echter dat het uitvoeren van het programma hierbij gepauzeerd wordt zolang dit formulier open blijft staan. Of m.a.w. de programmaregels die onder deze `ShowDialog`-instructie staan, worden pas uitgevoerd als dat formulier gesloten wordt.

Een formulier dat je met `ShowDialog()` opent, blijft ook geforceerd op de voorgrond staan in je toepassing en verhindert dat je naar een ander geopend formulier in de toepassing kan gaan.

Onthoud 8.1 Met de **methode `ShowDialog()`** laat je een formulier openen als een **dialoogvenster**.

Wanneer een dialoogvenster open staat, kan op dat moment geen enkel ander formulier in je toepassing de focus meer krijgen. De uitvoer van de code wordt ook *bevroren* bij een `ShowDialog()`-instructie.

De toepassing wordt pas weer vrijgegeven als je het *dialoogvenster* sluit.

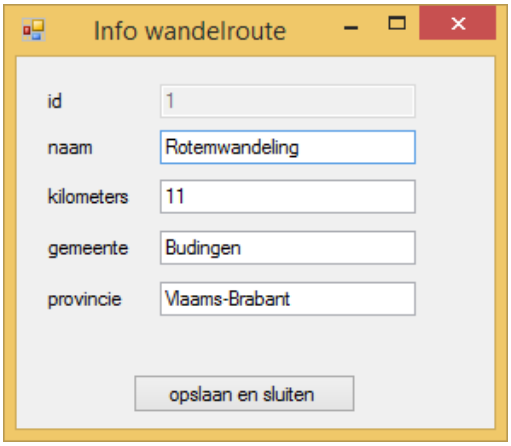
Nu is het mogelijk dat de gebruiker in het formulier `WandelrouteInfoForm` (dat als dialoogvenster geopend werd) ondertussen gegevens van de meegegeven wandelroute gewijzigd heeft. Als dit zo is, willen we uiteraard die wijzigingen in onze `ListBox` te zien krijgen.

We laten daarom de inhoud van de `ListBox` `wandelroutesListBox` vernieuwen door de lijst met alle wandelroutes (waar dus ook de bijgewerkte wandelroute in steekt) nog eens weg te schrijven naar de eigenschap `DataSource` van de `ListBox`. In hoofdstuk 7 leerden jullie dat dit enkel werkt als je hiervóór eerst de `DataSource` volledig leeg maakt (instelt op **null**).

```
wandelroutesListBox.DataSource = null;
wandelroutesListBox.DataSource = _wandelroutesLijst;
```

8.1.4 Een wandelroute wijzigen

Met de knop 'Toon' op `WandelroutesForm` laten we dus in een nieuw formulier (`WandelrouteInfoForm`) de details weergeven van de wandelroute die geselecteerd stond in de `ListBox`:



De achterliggende code hiervoor op `WandelrouteInfoForm`:

```
public partial class WandelrouteInfoForm : Form
{
    private Wandelroute _wandelroute;

    public WandelrouteInfoForm(Wandelroute wandelroute)
    {
        InitializeComponent();

        // laat het business-object _wandelroute wijzen naar de meegekregen
        // parameter
        _wandelroute = wandelroute;

        // gegevens van het business-object _wandelroute in de tekstvakken tonen
        idTextBox.Text = _wandelroute.Id.ToString();
        naamTextBox.Text = _wandelroute.Naam;
        kmTextBox.Text = _wandelroute.Kilometers.ToString();
        gemeenteTextBox.Text = _wandelroute.Gemeente;
        provincieTextBox.Text = _wandelroute.Provincie;
    }
```

Kort samengevat:

- We declareren in dit formulier een businessobject `Wandelroute _wandelroute`.
- Aan de header van de constructor van de klasse `WandelrouteInfoForm` hebben we een `Wandelroute`-parameter toegevoegd. Als er een nieuw `WandelrouteInfoForm`-formulier aangemaakt wordt, moet dus verplicht een wandelroute meegegeven worden.
- We laten in de constructor ons businessobject `_wandelroute` initialiseren op de wandelroute die we via de parameter binnenkrijgen.
- We zorgen tenslotte bij de constructor dat we alle gegevens van ons businessobject `_wandelroute` in de verschillende tekstvakken weergeven.

De gebruiker zou nu in de tekstvakken van dit formulier allerhande gegevens van de getoonde wandelroute kunnen veranderen (behalve in het Id-tekstvak dat we via de eigenschap `Enabled` lieten beschermen tegen wijzigingen).

Met de knop 'Opslaan en Sluiten' zorgen we dat alle wijzigingen die aangebracht zijn in de tekstvakken op hun beurt terug weggeschreven worden naar het business-object `_wandelroute`. We laten met andere woorden het object `_wandelroute` updaten naar de wijzigingen die de gebruiker gemaakt heeft. Daarna laten we het formulier zichzelf sluiten met de methode `Close()`. Dit allemaal met volgende code:

```
private void opslaanSluitenButton_Click(object sender, EventArgs e)
{
    // inhoud van de tekstvakken opslaan in het business-object _wandelroute
    _wandelroute.Naam = naamTextBox.Text;
    _wandelroute.Kilometers = Convert.ToInt32(kmTextBox.Text);
    _wandelroute.Gemeente = gemeenteTextBox.Text;
    _wandelroute.Provincie = provincieTextBox.Text;

    // laat het formulier sluiten
    Close();
}
```

Opmerking:

- De code die we bij het onderdeel "8.1.3 De details van een wandelroute tonen" en "8.1.4 Een wandelroute wijzigen" uit de doeken deden, zorgt er nu samen voor dat we de wandelroutes die we in de `ListBox` op `WandelroutesForm` zien, effectief kunnen wijzigen.

  Het geheim van de chef zit hem hieronder in het woordje '**verwijzing**'.

  - Met het knopje 'Toon' op `WandelroutesForm` geven we aan `WandelrouteInfoForm` een **verwijzing** door naar een wandelroute die in onze lijst met wandelroutes staat.
  - In `WandelrouteInfoForm` laten we met het knopje 'Opslaan en Sluiten' via deze **verwijzing** de wijzigingen dus effectief rechtstreeks aanbrengen aan die wandelroute in onze lijst.

8.1.5 Een nieuwe wandelroute toevoegen

De code om een nieuwe wandelroute toe te voegen (via de knop 'Nieuw') is verwant aan de code achter de knop 'Toon' (waarmee we een wandelroute konden wijzigen), maar bevat toch een beetje meer 'body':

```
private void nieuwButton_Click(object sender, EventArgs e)
{
    // wat is het hoogste ID van een wandelroute in de lijst _wandelroutesLijst
    int maxId = _wandelroutesLijst.Max(x => x.ID);

    // nieuw Wandelroute-object aanmaken
    Wandelroute wandelroute = new Wandelroute(maxId+1, "", 0, "", "");
```

```
    // nieuwe wandelroute toevoegen aan lijst wandelroutes
    _wandelroutesLijst.Add(wandelroute);

    // het formulier WandelrouteInfoForm openen
    // we geven de nieuwe wandelroute mee als parameter
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // de listbox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

We willen dat elke wandelroute in onze lijst een uniek ID bezit. Om dit zo te houden, moeten we natuurlijk zorgen dat we aan nieuwe wandelroutes ook een *nieuw* ID toekennen.

Om het 'volgende' unieke ID te kunnen bepalen, laten we eerst opzoeken wat het hoogste ID is dat al in de lijst met wandelroutes voorkomt.

```
int maxId = _wandelroutesLijst.Max(x => x.ID);
```

Opmerking:

- Je mag bovenstaande instructie als een *hocus pocus* oplossing zien. Pogen deze uit de doeken te doen zou ons veel te ver leiden. Wie interesse heeft moet maar eens op internet zoeken wat de mogelijkheden zijn van Lambda-functies bij een `List` in C# (en je zal begrijpen waarom we er hier verder strategisch het zwijgen toe doen).

We maken vervolgens een nieuw `Wandelroute`-object aan. De constructor bij de klasse `Wandelroute` vereist dat we al een ID, een naam, een afstand, een gemeente en een provincie opgeven.

```
Wandelroute wandelroute = new Wandelroute(maxId+1, "", 0, "", "");
```

Over de parameters die we bij de constructor meegeven, kunnen we volgende opmerkingen maken:

- Om ons ID uniek te maken, zetten we de nieuwe wandelroute op het hoogste ID dat in de lijst stond (`maxId`), vermeerderd met 1.
- Omdat we de gebruiker straks zelf de naam, de afstand, de gemeente en de provincie laten bepalen, zetten we voor de nieuwe wandelroute deze velden voorlopig nog op een 'lege' waarde (of een 0 voor de getallen en een "" voor de `String`-velden).

Heel belangrijk: de nieuwe (nog 'lege') wandelroute voegen we al toe aan onze lijst `_wandelroutesLijst`, zodat deze straks ook mee opgenomen wordt in de `ListBox`:

```
_wandelroutesLijst.Add(wandelroute);
```

Met de instructies die dan volgen, laten we de nieuwe wandelroute weergeven in het `WandelrouteInfoForm`-formulier.

```
WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
formulier.ShowDialog();
```

Dit zal concreet in het volgende formulier resulteren:

Omdat we dit `WandelrouteInfoForm`-formulier hier als een dialoogvenster openen (cfr. `ShowDialog()`), forceren we de gebruiker om in dit venster nú de gegevens van de nieuwe wandelroute verder aan te vullen.

Met het knopje 'Opslaan en Sluiten' laten we de wijzigingen opslaan in het nieuwe `Wandelroute`-object en wordt dit dialoogvenster gesloten, zodat we weer in het formulier met alle wandelroutes terecht komen.

Door middel van de laatste instructies waarmee we de `ListBox` terug laten vernieuwen, zorgen we dat de nieuw aangevulde wandelroute er achteraan in het lijstje bijkomt.

```
wandelroutesListBox.DataSource = null;
wandelroutesListBox.DataSource = _wandelroutesLijst;
```

8.1.6 Een wandelroute verwijderen

Om de cirkel helemaal rond te maken, kijken we ook nog even naar de code waarmee we een wandelroute uit onze lijst (en bijgevolg uit de `ListBox`) verwijderen.

Hieronder een print van de code achter de knop 'Verwijder':

```
private void verwijderButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute opvragen
    Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;

     // de geselecteerde wandelroute uit de lijst laten verwijderen
    _wandelroutesLijst.Remove(wandelroute);

    // De listbox laten vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Met de eigenschap `SelectedItem` gaan we eerst weer de wandelroute identificeren die geselecteerd staat in de `ListBox`.

```
Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;
```

Met volgende instructie laten we deze geselecteerde wandelroute verwijderen uit onze lijst:

```
_wandelroutesLijst.Remove(wandelroute);
```

Opmerking:

- In hoofdstuk 6 bij 'Onthoud 6.8' (zie pagina 198) hebben we jullie de methode `Remove()` voorgesteld waarmee je een gegeven **object** uit een `List` kan wissen. Het element dat je uit de `List` wil verwijderen, geef je mee als parameter bij deze methode.

Omdat er nu opnieuw een wijziging is aangebracht aan de lijst met wandelroutes (er is namelijk een object uit verdwenen), moeten we met de twee gekende `DataSource`-instructies de `ListBox` terug laten vernieuwen.

```
wandelroutesListBox.DataSource = null;
wandelroutesListBox.DataSource = _wandelroutesLijst;
```

8.1.7 Wat ontbreekt nog …

Ondertussen weten jullie dat onze solution `OpStap` dient om een lijst met wandelroutes te raadplegen, aan te passen, uit te breiden of terug in te krimpen. Jullie hebben ook even onder de motorkap kunnen kijken en jullie zijn dus op de hoogte hoe alles in de toepassing precies uitgewerkt (geprogrammeerd) werd.

Zolang de toepassing (in **run-time**) draait, zijn er geen klachten, want alle updates die we aanbrengen aan de wandelroutes worden perfect bijgehouden (in de variabele `_lijstWandelroutes`). Zodra we het project stoppen (we gaan naar **design-time**), loopt er een wiel van de wagen, want op dat moment gaat al ons werk verloren. Telkens we het project opnieuw opstarten, worden de wandelroutes immers terug op hun initiële beginsituatie gezet.

In het vervolg van dit hoofdstuk focussen we ons nog maar op één ding: "Hoe zorgen we ervoor dat de wijzigingen die we aanbrengen aan de lijst met wandelroutes, blijvend bewaard worden!"

We zijn dus genoodzaakt om de gegevens van de wandelroutes ergens extern (buiten het project) op te slaan. Een **databank** is hiervoor natuurlijk het aangewezen middel.

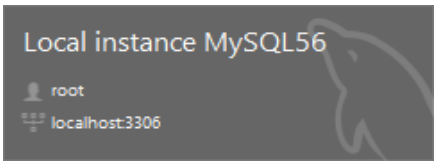
Omdat jullie eerder in de lessen informatica al met **MySql** leerden werken, kunnen we maar zo gemakkelijk zijn om hier nog eens voor dit databanksysteem te opteren.

8.2 De databank `wandelroutes`

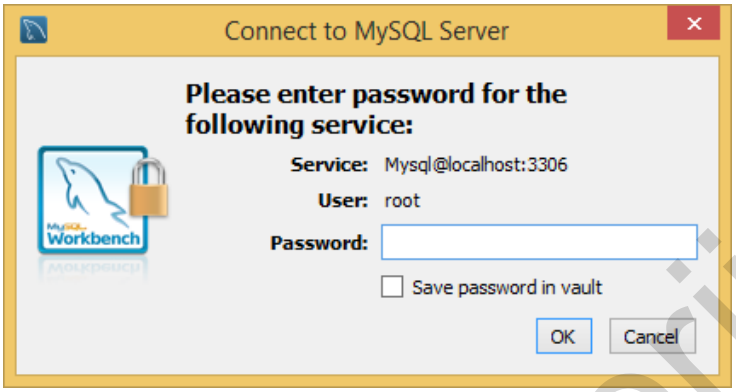
We hebben als beginsituatie al een kleine databank samengesteld met enkele wandelroutes. Dit is het bestand **Dump_wandelroutes.sql** (zie **_hoofdstuk_8/_voorbeelden/_databanken**).

Via onderstaande stappen laten jullie dit bestand inlezen in je MySQL-server:

1. Open **MySQL Workbench** op je computer.
2. Via het panel **'Local Instance MySQL56'** kan je je aanmelden op de MySQL-server.

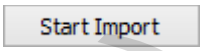


Op school hebben wij als wachtwoord **Leerling123** ingesteld.

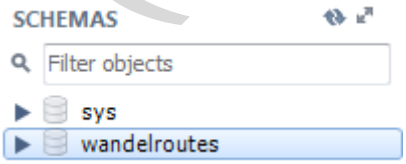


3. Met de knop 'Data Import/Restore' (Data Import/Restore) ga je naar de weergave om een databank te importeren.
4. Blader bij de optie 'Import from Self-Contained File' naar het bronbestand **Dump_wandelroutes.sql**:

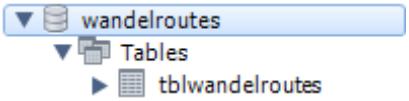
5. Met het knopje 'Start Import' laat je de import-actie effectief uitvoeren.



Het panel **Schemas** links onderaan bij WorkBench bevat alle databanken die ingeladen zijn op de MySQL-server. Misschien moet je er eerst nog eens het knopje vernieuwen ( ) aanklikken, maar dan staat de databank `wandelroutes` er zeker bij.



Door de databank open te klappen, merk je dat deze één tabel (`tblwandelroutes`) bevat:

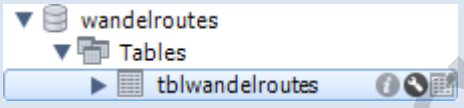


De gegevens in `tblwandelroutes` kan je bekijken door met de muis op `tblwandel-routes` te gaan staan en het derde icoontje () aan te klikken:

Er werden blijkbaar al 10 wandelroutes ingevoerd in de tabel:

id	naam	kilometers	gemeente	provincie
1	Rotemwandeling	12	Budingen	Vlaams-Brabant
2	Monnikenwandeling	4	Zoutleeuw	Vlaams-Brabant
3	Argendaalpad	13	Sint-Denijs	West-Vlaanderen
4	Bevergemroute	9	Sint-Denijs	West-Vlaanderen
5	Doornpannewandelroute	8	Koksijde	West-Vlaanderen
6	Hoge Blekkerroute	9	Koksijde	West-Vlaanderen
7	Poelbergwandeling	6	Tielt	West-Vlaanderen
8	Vlaskoorde	15	Sint-Baafs-Vijve	West-Vlaanderen
9	Zwalmpad	8	Brakel	Oost-Vlaanderen
10	Ooidonkroute	7	Deinze	Oost-Vlaanderen
NULL	NULL	NULL	NULL	NULL

Oefening 8.1 Om onderstaande vraagjes te beantwoorden, zal je de ontwerpweergave van `tblwandelroutes` moeten raadplegen. Deze vraag je het gemakkelijkst op via het tweede icoontje () bij de tabel:



- Wat is het **gegevenstype** van de verschillende velden?

  `id`:
  `naam`:
  `kilometers`:
  `gemeente`:
  `provincie`:

- Welk veld doet dienst als **Primary Key** in de tabel?

- Is het veld `id` ingesteld als een **autonummeringsveld**?

8.3 De Data Access Layer - voorbereiding

We weten nu dat onze wandelroutes blijvend gestockeerd worden in een tabel `tblwandelroutes` in een databank `wandelroutes`. Deze databank draait op de MySQL-server op je computer.

De volgende grote stap is om deze tabel met wandelroutes te linken aan onze bestaande OpStap-toepassing!

Omdat een databank benaderen een heel specifieke taak is, beschouwen we het geheel van de code waarmee we dit regelen als een **afzonderlijke laag** in een toepassing!

We noemen deze laag de **Data Access Layer**. Naast de **Business Layer** en de **Presentation Layer** (die samen het tweelagenmodel vormgeven), introduceren we met de Data Access Layer dus een nieuwe **derde** laag!

> **Onthoud 8.2** Als in je toepassing gebruik gemaakt wordt van gegevensopslag in een **externe databank**, dan noemen we de code waarmee we de toegang tot die databank regelen de **Data Access Layer**.
>
> Dit omvat code om de gegevens uit de databank te lezen en code om gegevens weg te schrijven naar de databank.

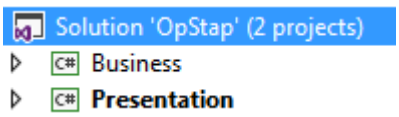
Alvorens we ons op de specifieke programmacode kunnen richten in de Data Access Layer, moeten we eerst enkele voorbereidingen treffen in onze OpStap-toepassing.

We zullen een apart project voorzien in onze solution om er de data-accesscode onder te brengen en leggen er enkele noodzakelijke references. We tonen hoe je in een nieuw Visual Studio venster (de Server Explorer) een databank kan inspecteren.

8.3.1 Elke Layer in een apart project

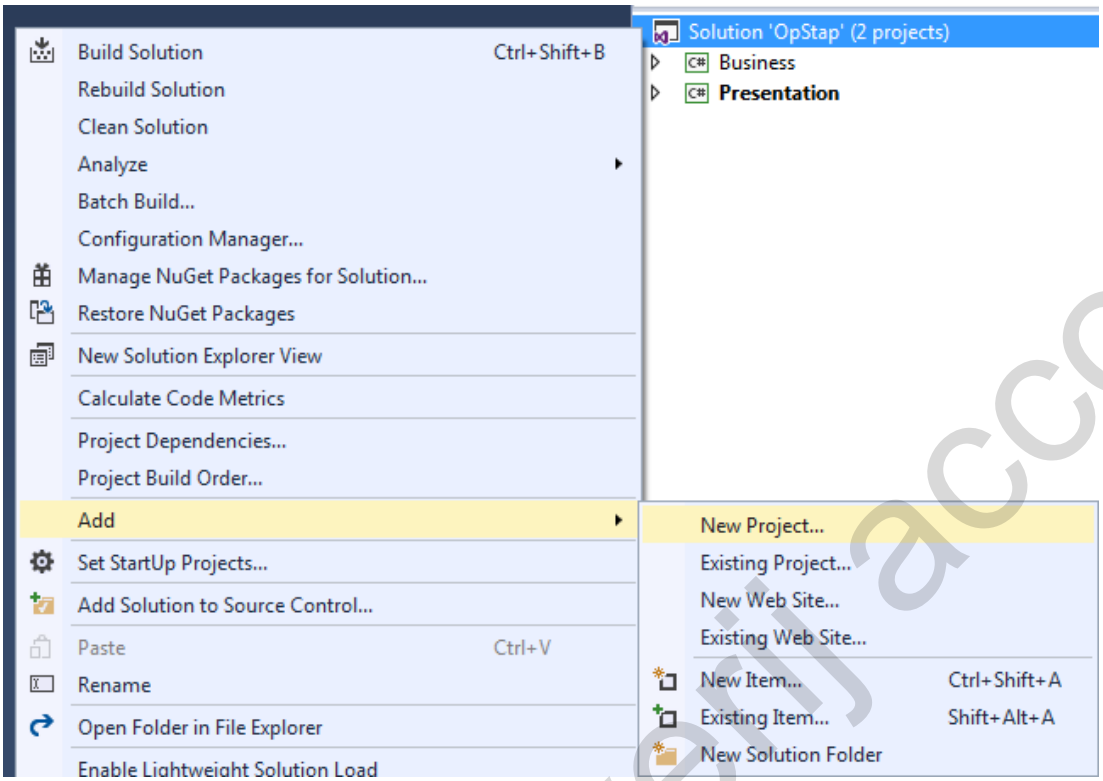
We hebben ondertussen de goede gewoonte gekweekt om bij 'grotere' toepassingen de business- en de presentatiecode onder te brengen in aparte projecten.

Bij onze OpStap-toepassing zien we in de **Solution Explorer** een solution waaronder de Business - en de Presentation Layer elk over een eigen project beschikken. Zo houden we het overzichelijk.



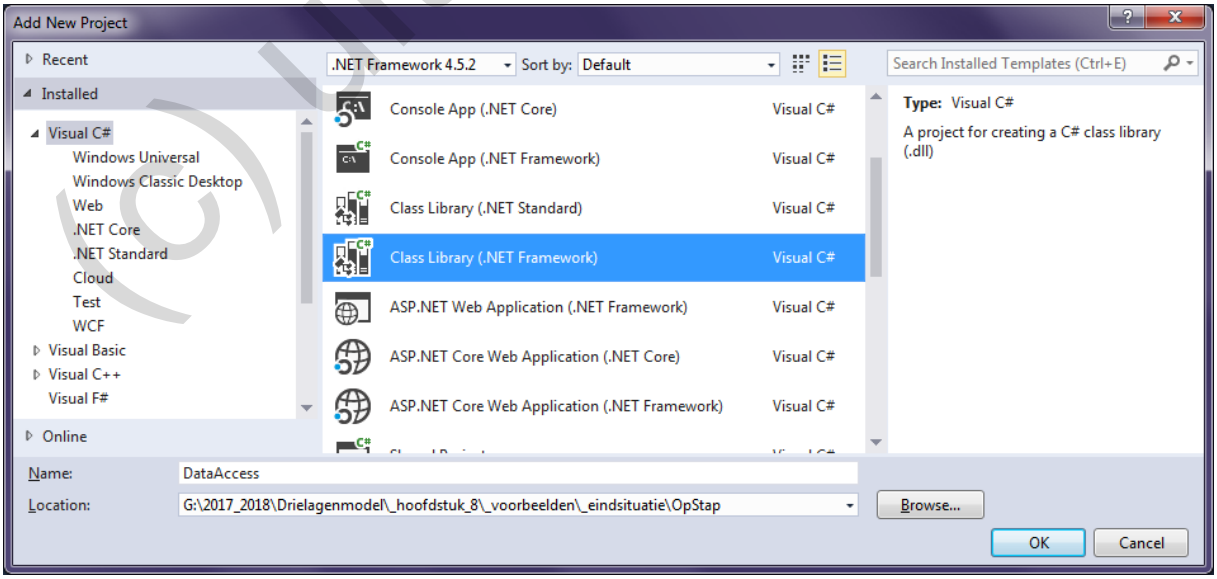
Nu we met een derde Layer (de Data Access Layer) op de proppen komen, is het maar logisch dat we die ook een eigen project geven in de solution!

Om dit nieuwe project aan onze solution toe te voegen, klik je in de Solution Explorer rechts op de solution `OpStap` en volg je: Add | New Project …

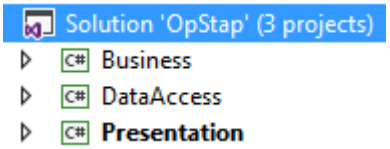


In de Data Access Layer zullen we nooit formulieren onderbrengen (dit is immers gereserveerd voor de Presentation Layer). In het 'Data Access'-project komen straks *gewone* klassen waarmee we de toegang tot de databank regelen. Daarom mag je voor dit nieuwe project het type '**Class Library (.NET Framework)**' selecteren.

We geven het nieuwe project, naar analogie met de andere projecten, de naam **DataAccess**:



In de Solution Explorer hebben we nu een solution met drie projecten:



Het project `Presentation` staat nog steeds in het vet. Dat hoort zo, want `Presentation` blijft het **Startup**-project. Dit project bevat immers het formulier dat geopend moet worden bij het starten van de toepassing!

8.3.2 References

References tussen projecten

Tussen projecten in een Visual Studio solution worden standaard dikke muren opgetrokken. In principe zijn projecten onzichtbaar voor elkaar.

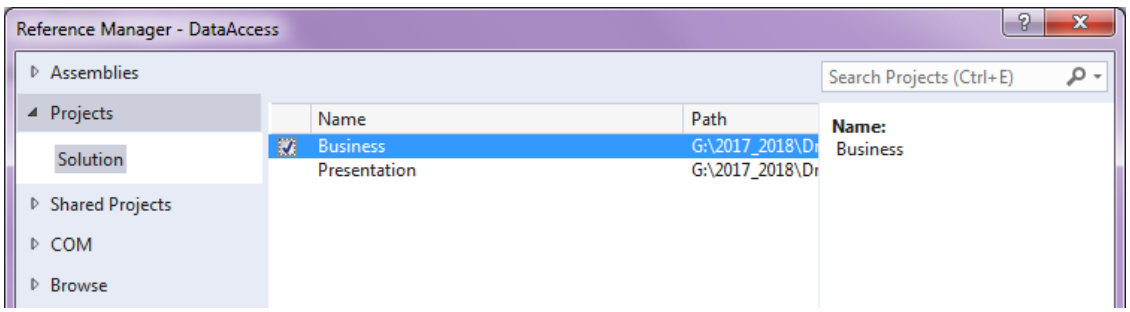
Dat is dan ook de reden waarom we in de solution handmatig in het project `Presentation` een **reference** moesten toevoegen naar het project `Business`. Zonder die reference is het niet mogelijk om in de presentatielaag gebruik te maken van de klassen die gedefinieerd werden in de businesslaag. Omdat we in een formulier (presentatielaag) gewoonlijk een businessobject declareren, moeten de businessklassen er net wél bereikbaar zijn!

Nu we een derde project (het project `DataAccess`) in onze solution hebben, zullen we nog wat meer muren moeten slopen, of m.a.w. nog enkele extra references moeten toevoegen.

Pas als we later aan het programmeren slaan, zal duidelijk worden in welke klassen we gebruik maken van code uit andere projecten. Om straks echter niet opgehouden te worden door ontbrekende verwijzingen, willen we nu al opsommen welke references we waar nodig zullen hebben en laten jullie deze references onmiddellijk *preventief* toevoegen:

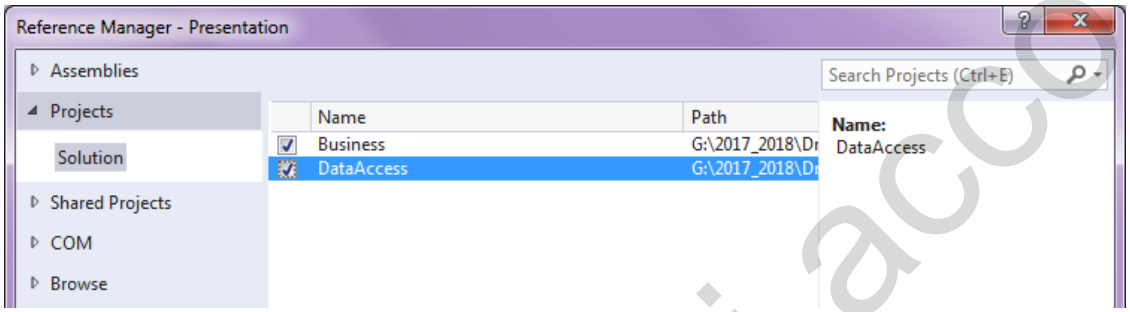
- In het project `DataAccess` zullen we werken met objecten van de businessklasse `Wandelroute`. Aan het project `DataAccess` voegen we daarom een reference toe naar het project `Business`.

  Klik in de Solution Explorer rechts op het project `DataAccess` en volg er Add | Reference… Vink het project `Business` aan.



- Vanuit het formulier `WandelroutesForm` zullen we de taak opstarten om wandelroutes op te halen uit de databank en zullen we instructies uitvoeren om wandelroutes op te slaan in de databank. `WandelrouteForm` zal hiervoor beroep doen op de vele methoden die we nog in de Data Access Layer moeten definiëren. Het project `Presentation` heeft daarvoor een reference nodig naar het - voorlopig nog lege - project `DataAccess`.

  Klik rechts op het project `Presentation` en volg er Add | Reference… Vink daar naast het project `Business` nu ook het project `DataAccess` aan.



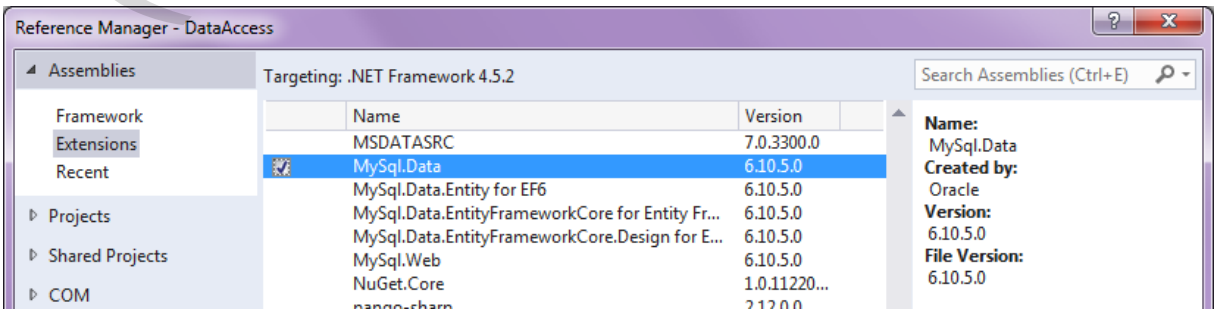
Reference naar MySql

Vanuit Visual Studio kan je niet 'zomaar' een connectie leggen met een MySQL-databank. De klassen waarmee de toegang tot zo'n databank opgezet kan worden, zijn standaard NIET beschikbaar in Visual Studio.

Als je MySQL met de nodige componenten (**Connector/NET**) op je pc geïnstalleerd hebt (!), staat ergens op je computer een **dll-bestand** dat de noodzakelijke MySQL-klassen bevat (onderverdeeld in enkele namespaces).

Jij moet expliciet een verwijzing (**reference**) leggen naar dit dll-bestand om deze klassen te kunnen gebruiken. Omdat we in onze toepassing alle code betreffende de MySQL-databank verzamelen in de Data Access Layer, zullen we deze reference toevoegen aan het project `DataAccess`.

Klik in de **Solution Explorer** rechts op de projectnaam `DataAccess` en volg er Add | Reference... De referentie die wij nodig hebben, vind je in de lijst bij **Assemblies** | **Extensions**, waar je het item `MySql.Data` aanvinkt.



Onthoud 8.3 Voeg aan het project waar je de connectie met een MySQL-databank wilt opzetten een verwijzing (**reference**) toe naar de `MySql.Data`-Assembly. Enkel zo zijn de nodige klassen waarmee je een MySQL-databank kan benaderen beschikbaar in dit project.

Deze Assembly werd via de **Connector/NET**-component van MySQL op je computer geïnstalleerd.

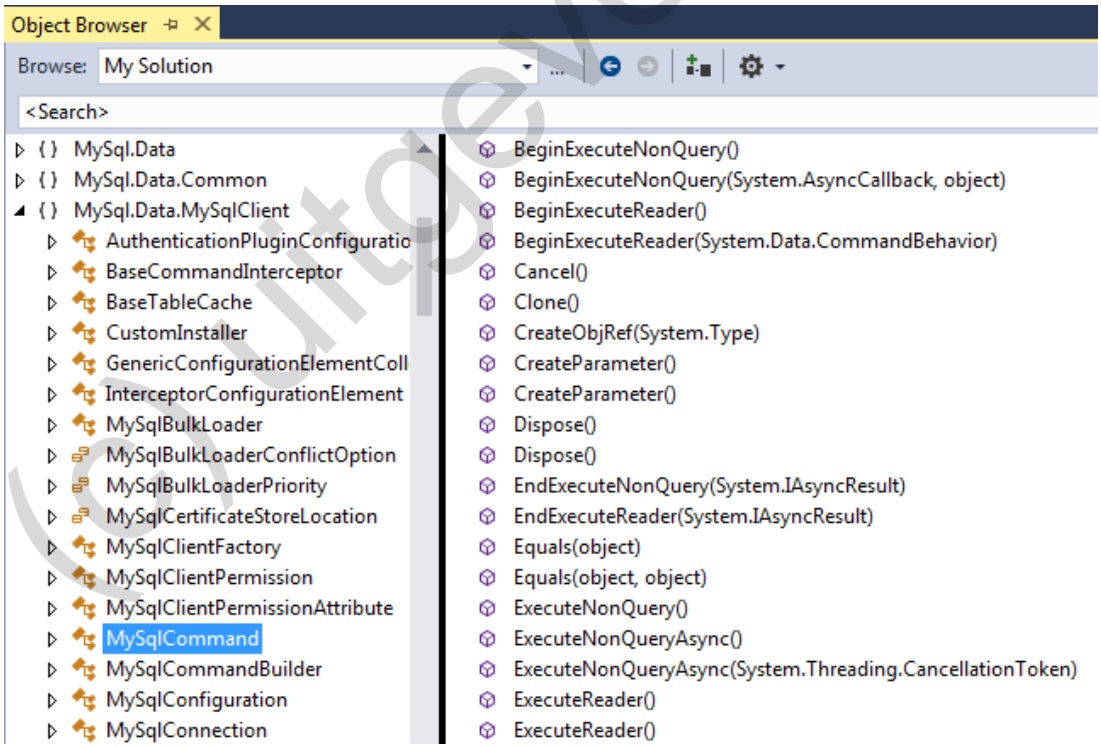
Opmerkingen:

- Afhankelijk van de geïnstalleerde versie van Connector/NET zal bij de `MySQL.Data`-reference een ander versienummer vermeld staan.

  Mogelijk moet je in dit project deze reference naar `MySQL.Data` herstellen als je de solution opent op een computer waar een andere versie van Connector/NET geïnstalleerd werd.

- Als je dit wilt, kan je de vele MySQL-klassen die via deze reference geïmporteerd werden, bekijken in de **Object Browser**. De betreffende namespaces starten allemaal met `MySql.Data`.

  De hieronder aangeduide klasse `MySQLCommand` is er bijvoorbeeld eentje die verder in dit hoofdstuk nog regelmatig aan bod zal komen:



8.3.3 De Server Explorer

Als je straks code schrijft waarmee je gegevens uit de tabel `tblwandelroutes` gaat inlezen, zal een overzichtje met de veldnamen van deze tabel zeker handig blijken. Door in **MySQL WorkBench** de tabel `tblwandelroutes` open te zetten, heb je deze veldnamen onmiddellijk bij de hand.

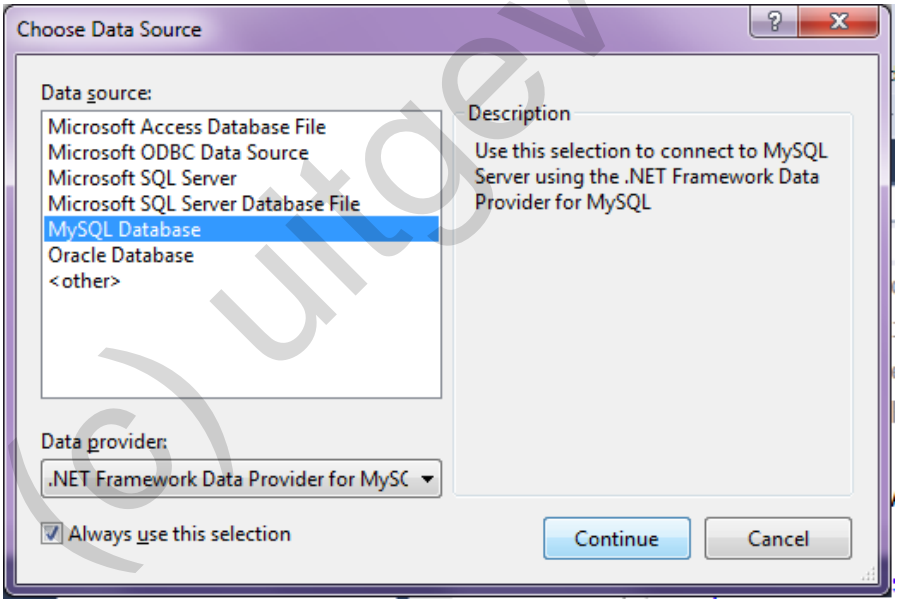
Visual Studio ontwikkelaars maken echter graag gebruik van de mogelijkheid om rechtstreeks vanuit Visual Studio de externe databanken te kunnen inspecteren. Als je bij MySQL de component **MySQL for Visual Studio** liet installeren (!), dan werd er aan Visual Studio een extra module toegevoegd waarmee je een MySQL-databank kan binnentrekken in een Visual Studio venster.

We demonstreren hieronder de stappen om de databank `wandelroutes` (die we reeds eerder op onze MySQL-server geïmporteerd hebben) beschikbaar te maken in een Visual Studio venster.

1. Open in Visual Studio de **Server Explorer** (via View | Server Explorer).

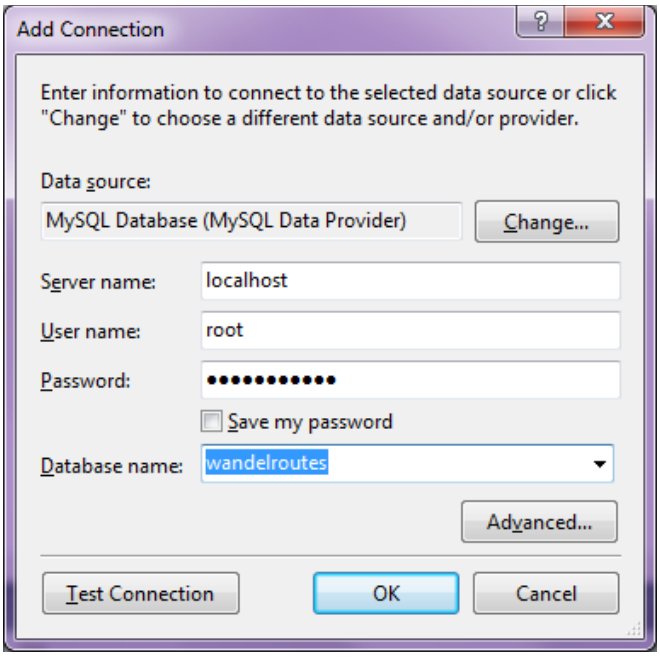
2. Via de knop 'Connect to Database' () kunnen we een connectie naar een externe databank toevoegen.

   We specificeren dat het om een **MySQL Database** gaat:



   Als je de optie '**Always use this selection**' aanvinkt, krijg je dit venster nooit meer te zien en zal de wizard sowieso van een MySQL Database uitgaan.

3. We moeten de nodige credentials verschaffen om toegang te krijgen tot de databank.



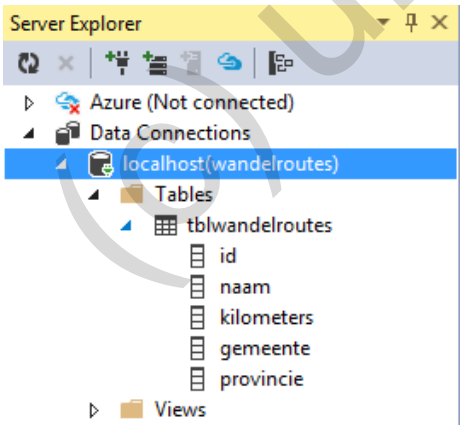
Server name: localhost (*localhost verwijst naar de MySQL-server die lokaal op jouw computer draait*);

User name: root (*root is een gebruiker die de juiste rechten bezit op de MySQL-server*);

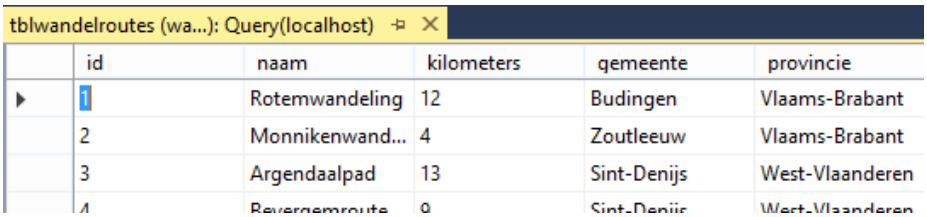
Password: Leerling123 (*stelden wij zo in bij onze installatie van MySQL*);

Database name: wandelroutes.

In de Server Explorer kan je nu door de juiste items open te klikken, de velden van `tblwandelroutes` bekijken.



Klik je in de Server Explorer in het snelmenu (rechtermuisknop) van `tblwandelroutes` de optie 'Retrieve Data' aan, krijg je ook **de records** in deze tabel te zien:



id	naam	kilometers	gemeente	provincie
1	Rotemwandeling	12	Budingen	Vlaams-Brabant
2	Monnikenwand...	4	Zoutleeuw	Vlaams-Brabant
3	Argendaalpad	13	Sint-Denijs	West-Vlaanderen
4	Bevergemroute	9	Sint-Denijs	West-Vlaanderen

Onthoud 8.4 Als bij de installatie van je MySQL-server de component **MySQL for Visual Studio** opgenomen werd, kan je **in Visual Studio** via de **Server Explorer** je MySQL-databanken bekijken.

Je kan zo rechtstreeks vanuit Visual Studio het ontwerp (welke velden) en de inhoud (welke records) van de tabellen in je MySQL-databanken raadplegen.

8.4 De Data Access Layer - programmacode

8.4.1 De klasse `WandelrouteDA`

Alle code waarmee we vanuit ons project de tabel `tblwandelroutes` benaderen, zullen we verzamelen in **één klasse**. Deze klasse hoort uiteraard thuis in het project `DataAccess`.

Klik in de Solution Explorer rechts op het project `DataAccess` en volg er Add | Class... We noemen deze nieuwe klasse `wandelrouteDA` (waarbij `DA` de afkorting is van **Data Access**). Visual Studio vergeet steeds om een klasse public te maken, maar wij wisten het nog wel:

```
public class WandelrouteDA
{
}
```

Het is een goede praktijk om voor elke (MySQL-)tabel die je vanuit je project benadert een aparte data-accessklasse te programmeren. In onze toepassing hebben we maar één externe tabel (`tblwandelroutes`), of onze Data Access Layer zal volstaan met de klasse `WandelrouteDA`.

Je maakt het je straks wat makkelijker door de namespace `MySQL.Data.MySqlClient` bij de **`using`**-statements op te nemen. Zo voorkomen we dat we telkens als we er een vermelding maken van een klasse uit deze namespace (en dit zullen we meerdere keren doen), we er '`MySql.Data.MySqlClient`' bij moeten tikken.

`MySQL.Data.MySqlClient` is één van de namespaces die we in het `DataAccess`-project geïmporteerd hebben door de reference toe te voegen naar de `MySql.Data`-Assembly (zie pagina 250).

Omdat we straks in onze data-accessklasse `WandelrouteDA` met objecten van de klasse `Wandelroute` gaan werken en die klasse `Wandelroute` in het businessproject `Business` gedefinieerd werd, doen we er ook een using naar de namespace `Business` bij:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;
using Business;
```

Om onze data-accessklasse op te zetten, hebben we nog enkele heel specifieke instructies nodig. Laat je niet overdonderen, diezelfde standaardinstructies komen in elke data-access-klasse terug.

In de klasse `WandelrouteDA` declareren we twee velden:

```
public class WandelrouteDA
{
    private String _connString;
    private MySqlConnection _mySqlConnection;
}
```

De data-accessklasse moet weten tot **welke** externe databank hij zich moet richten. Vragen als: "Op welke MySQL-server staat deze databank?"; "Wat is de naam van de databank?" en "Met welke gebruikersnaam en wachtwoord meld je je aan op de MySQL server?" moeten hierbij beantwoord worden.

Deze informatie wordt standaard voorgesteld met een zogeheten **connectiestring**. We zullen deze connectiestring bijhouden in een veld `_connString`. Omdat dit nog over 'tekst'-informatie gaat, declareren we dit veld `_connString` van het type `String`.

Om lees- of schrijfacties in een databank uit te voeren heb je een fysieke **connectie** met deze databank nodig. Zo'n connectie is een object van de klasse `MySqlConnection`. In onze data-accessklasse `WandelrouteDA` zullen we het veld `_mySqlConnection` gebruiken om de connectie naar de databank `wandelroutes` bij te houden.

Zoals we dit al vanaf les één in BlueJ deden: "De velden van een klasse moet je in de constructor **initialiseren**". In de klasse `WandelrouteDA` voorzien we volgende constructor om onze twee velden de juiste startwaarden te geven:

```
public class WandelrouteDA
{
    private String _connString;
    private MySqlConnection _mySqlConnection;

    public WandelrouteDA()
    {
        // connectiestring voor de MySQL-databank wandelroutes
        _connString = "server=localhost;user id=root;Password=Leerling123;
                       database=wandelroutes";
        // initialiseer de connectie op basis van de connectiestring
        _mySqlConnection = new MySqlConnection(_connString);
    }
}
```

We stellen het veld `_connString` in op de connectiestring voor onze databank `wandelroutes`.

```
_connString = "server=localhost;user id=root;Password=Leerling123;
               database=wandelroutes";
```

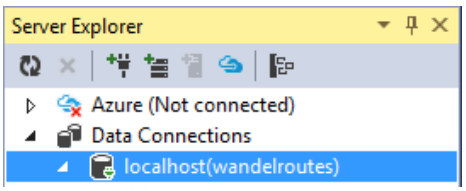
Deze connectiestring beschrijft, in het vereiste formaat, ontegensprekelijk welke databank we bedoelen, namelijk de databank **`wandelroutes`** op de MySQL-server **localhost** waarop we als de gebruiker **root** met het wachtwoord **Leerling123** inloggen.

Dit zijn dezelfde credentials die we eerder invulden om deze databank binnen te nemen in de **Server Explorer** (zie pagina 253).

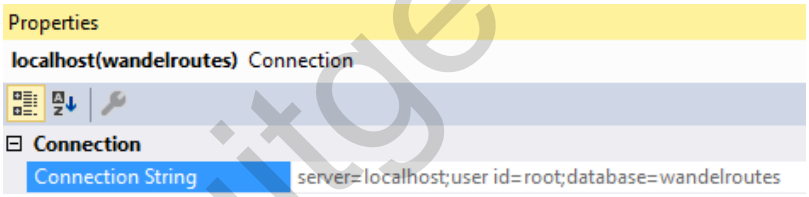
Opmerking:

- Met een trucje kan je trouwens vanuit de Server Explorer deze connectiestring recupereren:

  1. Selecteer in de Server Explorer de connectie naar de databank `wandelroutes`:



  2. Ga in het Properties venster naar de eigenschap `Connection String`:



  Je kan dus *bijna* de volledige connectiestring vanuit deze Property kopiëren en zo in je code plakken. Het wachtwoord (`Password=Leerling123`) zal je er zelf moeten bijtikken, want blijkbaar wordt die regel niet bijgehouden bij de Property.

Daarna laten we het veld `_mySqlConnection` initialiseren als een nieuw `MySqlConnection`-object. Je merkt dat we bij deze constructor als parameter de connectiestring naar de databank moeten meegeven. Of dus:

```
_mySqlConnection = new MySqlConnection(_connString);
```

Opmerking:

- Het **using**-statement van de namespace `MySql.Data.MySqlClient` bewijst zijn nut. De klasse `MySqlConnection` is er namelijk eentje uit deze namespace. Zonder de using hadden we deze namespace voluit moeten uittikken bij elk gebruik van deze klasse:

```
private MySql.Data.MySqlClient.MySqlConnection _mySqlConnection;
```

en

```
_mySqlConnection = new MySql.Data.MySqlClient.MySqlConnection(_connString);
```

8.4.2 Databankgegevens lezen in de klasse `WandelrouteDA`

De volgende stap is om in de data-accessklasse `WandelrouteDA` **methoden** te voorzien waarmee we databankbewerkingen op de tabel `tblwandelroutes` kunnen uitvoeren en de meest voor de hand liggende databankbewerking, dat is het **uitlezen** van (alle) wandelroutes in de tabel.

We laten dit werk opknappen in een nieuwe methode `ReadTable()`. Het retourtype van deze methode stellen we in op `List<Wandelroute>`, want deze methode zal als resultaat een **lijst** retourneren met alle wandelroutes die in de tabel `tblwandelroutes` staan.

Hieronder geven we al eens een volledige print van de methode `ReadTable()`.

Deze methode zal waarschijnlijk het meest onbegrijpelijke stuk code lijken dat we jullie al voorgeschoteld hebben, maar GEEN PANIEK ... Als je straks de betekenis van elk van de instructies snapt, wordt het al heel wat concreter. We verwachten ook niet dat jullie deze code *gesloten boek* uit een hoge hoed kunnen toveren.

```
public List<Wandelroute> ReadTable()
{
    List<Wandelroute> lijst = new List<Wandelroute>();

    // SQL-statement om alle wandelroutes, alfabetisch gerangschikt, op te vragen
    String sql = "SELECT * FROM tblwandelroutes ORDER BY naam;";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // met ExecuteReader laat je een leescommando opstarten
    // de ingelezen informatie komt in mySqlDataReader terecht
    MySqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

    // lusje om alle records in mySqlDataReader te overlopen
    while (mySqlDataReader.Read()==true)
    {
        // nieuwe wandelroute maken met de actieve record in mySqlDataReader
        Wandelroute wandelroute =
            new Wandelroute((int)(mySqlDataReader["id"]),
                            mySqlDataReader["naam"].ToString(),
                            (int)(mySqlDataReader["kilometers"]),
                            mySqlDataReader["gemeente"].ToString(),
                            mySqlDataReader["provincie"].ToString());

        // voeg de wandelroute toe aan de lijst
        lijst.Add(wandelroute);
    }

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();

    // lijst met alle wandelroutes retourneren
    return lijst;
}
```

De essentie van deze methode `ReadTable()` is dat we er een lijst (`List`) zullen opbouwen waarin we alle wandelroutes stoppen die in de tabel `tblwandelroutes` staan.

Hiermee begrijpen jullie dus al waarom we onderstaande declaratie en initialisatie nodig hebben:

```
List<Wandelroute> lijst = new List<Wandelroute>();
```

En dan al eens vooruit lopen. Een wandelroute toevoegen aan deze lijst, gebeurt verder in de methode met:

```
lijst.Add(wandelroute);
```

Om deze lijst uiteindelijk te laten retourneren, zullen we de methode afsluiten met:

```
return lijst;
```

De universele taal om een databank aan te spreken is **SQL**. En hupsakee, we kunnen nu dus ook op kousenvoeten de leerstof van de cursus MySQL binnensmokkelen in deze cursus programmeren. Of ... SQL-instructies leren schrijven, was geen verloren moeite 😉!

Verwacht voorlopig nog geen ingewikkeld **SQL-statement**, want de instructie om alle wandelroutes in te lezen (en deze in alfabetische volgorde te plaatsen) kan met:

```
SELECT      *
FROM        tblwandelroutes
ORDER BY    naam
```

In de methode `ReadTable()` laten we ditzelfde SQL-statement als volgt in een `String`-variabele opslaan:

```
String sql = "SELECT * FROM tblwandelroutes ORDER BY naam;";
```

Onze variabele `sql` is *maar* een *tekstje*. Als je effectief een SQL-statement wilt uitvoeren op een databank, heb je een object van de klasse `MySqlCommand` nodig. Wij declareren en initialiseren daarom in de methode `ReadTable()` zo'n nieuw object (en noemen dit object `mySqlCommand`).

Bij de constructor van deze klasse `MySqlCommand`, moet je **twee parameters** aanleveren, namelijk het SQL-statement in tekstformaat (of dus onze variabele `sql`) en de connectie met de databank waarop je deze SQL-instructie wilt toepassen (of dus het veld `_mySqlConnection`):

```
MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

Je kan een `MySqlCommand` enkel en alleen toepassen op een databank waarvoor je een connectie geopend hebt! Onze connectie met de databank houden wij bij in het veld `_mySqlConnection`. Deze connectie zetten we als volgt open:

```
_mySqlConnection.Open();
```

Omdat een open connectie heel wat systeembronnen van je computer vraagt, laten we *op het einde* van de methode de connectie terug sluiten. Een connectie niet sluiten, kan je toepassing ernstige schade toebrengen (laten crashen):

```
_mySqlConnection.Close();
```

Als je een `MySqlCommand` klaar hebt (wij noemden dit object `mySqlCommand`) en de connectie met de databank staat open, dan ben je helemaal klaar om effectief dit `MySqlCommand` uit te voeren.

Omdat ons `MySQLCommand` een **lees-operatie** is die **meer dan één waarde** oplevert (het resultaat bevat meerdere velden en/of records), starten we het command op via de methode `ExecuteReader()`.

De velden/records die deze methode `ExecuteReader()` ophaalt, worden opgeslagen in een object van de klasse `MySqlDataReader`.

```
MySqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
```

Onthoud 8.5 Een `MySqlCommand`-object omvat een SQL-statement en is in staat om dit SQL-statement effectief uit te voeren op een databank waarvoor een connectie geopend werd.

Als dit SQL-statement een **lees-operatie** (een **SELECT**) is, die **meer dan één waarde** oplevert (meerdere velden en/of records), dan moet je hiervoor de methode `ExecuteReader()` toepassen op het `MySqlCommand`-object.

De gegevens die dit command inleest, worden opgeslagen in een variabele van het type `MySqlDataReader`.

De situatie is nu dat we in onze methode `ReadTable()` het object `mySqlDataReader` alfabetisch lieten opvullen met alle wandelroutes uit de MySQL-tabel `tblwandelroutes`.

Wij moeten deze wandelroutes nu echter nog in onze eigen lijst krijgen (zie de variabele `List<Wandelroute> lijst`)! Om dit voor elkaar te krijgen, gaan we alle records uit `mySqlDataReader` één per één overlopen en hierbij telkens de actieve rij aan onze lijst toevoegen.

Het ware *leuk* geweest, hadden we de records in een `MySqlDataReader` met een foreach-lusje kunnen doorlopen. De foreach is immers de lus-structuur die in jullie curriculum al het vaakst voorbij kwam.

Helaas pindakaas, een `MySqlDataReader` eist een bijzondere programmeertruc om zijn inhoud vrij te geven en daarbij hoort een **while**-lus.

Het lusje ziet er als volgt uit:

```
// lusje om alle records in mySqlDataReader te overlopen
while (mySqlDataReader.Read()==true)
{
    // nieuwe wandelroute maken met de actieve record in mySqlDataReader
    Wandelroute wandelroute =
         new Wandelroute((int)(mySqlDataReader["id"]),
                         mySqlDataReader["naam"].ToString(),
                         (int)(mySqlDataReader["kilometers"]),
                         mySqlDataReader["gemeente"].ToString(),
                         mySqlDataReader["provincie"].ToString());

    // voeg de wandelroute toe aan de lijst
    lijst.Add(wandelroute);
}
```

De expressie "`mySqlDataReader.Read()`" die we in de while-voorwaarde gebruiken, betekent dat de *volgende* record in `mySqlDataReader` **actief** gemaakt wordt. Deze expressie geeft als retourwaarde **false** terug mochten we al op de laatste record staan (want dan is er geen *volgende* record meer om actief te maken); anders is de retourwaarde **true**.

Met de while-voorwaarde '`(mySqlDataReader.Read()==true)`' verkrijgen we dus een lus die zich herhaalt tot alle records in de `mySqlDataReader` doorlopen zijn. Elke record in `mySqlDataReader` heeft hierbij de eer gehad om eens de **actieve record** te mogen spelen.

Bij een `MySqlDataReader`-object kan je met de volgende expressie aan de **actieve record** de waarde opvragen van een bepaald **veld**:

```
object["veld"]
```

Om dus bijvoorbeeld aan ons object `mySqlDataReader` de afstand (of het veld `kilometers`) op te vragen bij de **actieve record**, noteer je:

```
mySqlDataReader["kilometers"]
```

De laatste puzzelstukjes vallen in elkaar:

Binnen de lus maken we telkens een nieuw `Wandelroute`-object aan en we initialiseren dit object met de gegevens die we aan de **actieve record** in `mySqlDataReader` opvragen.

De constructor van de klasse `Wandelroute` verlangt hiervoor vijf parameters. We zullen uit ons `mySqlDataReader`-object dus vijf maal (voor het id, de naam, de afstand, de gemeente en de provincie) de waarde van een bepaald veld ophalen.

Waren er in C# GEEN conversies tussen gegevenstypen nodig geweest, dan hadden we er ons met volgende instructie vanaf kunnen maken:

```
Wandelroute wandelroute =
        new Wandelroute(mySqlDataReader["id"], mySqlDataReader["naam"],
                        mySqlDataReader["kilometers"], mySqlDataReader["gemeente"],
                        mySqlDataReader["provincie"]);
```

Maar zoveel geluk hebben wij nooit … Elk gegeven dat je uit een `mySqlReader` ophaalt, zal je expliciet naar het juiste type moeten omzetten!

Als je moet converteren naar een numeriek type, dan kan dit met de korte ()-notatie. Casten naar het type `String` doe je met de `ToString()`-methode.

De constructor bij de klasse `Wandelroute` heeft twee `int`-parameters (`id` en `kilometers`) en drie `String`-parameters (`naam`, `gemeente` en `provincie`). We implemteren dus nog verplicht onderstaande vijf conversies bij de instructie waarmee we het `Wandelroute`-object aanmaken:

```
Wandelroute wandelroute =
        new Wandelroute((int)mySqlDataReader["id"],
                        mySqlDataReader["naam"].ToString(),
                        (int)mySqlDataReader["kilometers"],
                        mySqlDataReader["gemeente"].ToString(),
                        mySqlDataReader["provincie"].ToString());
```

Nog eens herhalen dat we alle nieuwe `Wandelroute`-objecten die we op die manier in de lus samenstellen, laten toevoegen aan onze eigen lijst:

```
lijst.Add(wandelroute);
```

Voor elke wandelroute in de MySQL-tabel `tblwandelroutes`, zal er dus één overeenkomstig object in die lijst komen. Op het einde van de methode retourneren we die lijst met alle `Wandelroute`-objecten.

```
return lijst;
```

De methode `ReadTable()` toepassen

Het was een hele trip om de methode `ReadTable()` van naaldje tot draadje te analyseren, maar we weten nu tenminste dat we in de data-accessklasse `wandelrouteDA` een methode voorhanden hebben om de gegevens uit de MySQL-tabel `wandelroute` te lezen. Deze wandelroutes worden hierbij als een lijst (`List`) van `Wandelroute`-objecten aangeleverd.

De hoogste tijd om eens te kijken hoe we deze methode in ons `WandelroutesForm`-formulier kunnen integreren, zodat we er in de `ListBox` voortaan de wandelroutes **uit de databank** te zien krijgen! In `WandelroutesForm` wordt momenteel nog volgende code gebruikt om het formulier op te starten:

```
public partial class WandelroutesForm : Form
{
    private List<Wandelroute> _wandelroutesLijst;

    public WandelroutesForm()
    {
        InitializeComponent();

        // het veld _wandelroutesLijst initialiseren
        _wandelroutesLijst = new List<Wandelroute>();

        // via de hulpmethode LijstOpvullen() laten we het veld _wandelroutesLijst
        // opvullen met enkele wandelroutes
        LijstOpvullen();

        // de lijst met wandelroutes weergeven in de listbox
        wandelroutesListBox.DataSource = _wandelroutesLijst;
    }
```

We declareren en initialiseren in dit formulier m.a.w. een lijst van `Wandelroute`-objecten. We laten bij de constructor van het formulier deze lijst met wandelroutes weergeven in de `ListBox wandelroutesListBox` (via de Property `DataSource`).

Om enkele wandelroutes in de lijst te krijgen, roepen we de hulpmethode `LijstOpvullen()` op, waar we handmatig enkele `Wandelroute`-objecten aanmaken om die dan in de lijst te stoppen:

```
private void LijstOpvullen()
{
    // wandelroutes aanmaken en aan de lijst _wandelroutesLijst toevoegen
    Wandelroute w1 = new Wandelroute(1, "Rotemwandeling", 11, "Budingen",
                                     "Vlaams-Brabant");
    Wandelroute w2 = new Wandelroute(2, "Monnikenwandeling", 4, "Zoutleeuw",
                                     "Vlaams-Brabant");
    ...

    _wandelroutesLijst.Add(w1);
    _wandelroutesLijst.Add(w2);
    ...
}
```

Wij willen nu zo snel mogelijk van deze methode `LijstOpvullen()` af!

Onze lijst `_wandelroutesLijst` zouden we voortaan immers gaan bevolken met de wandelroutes uit de tabel `tblwandelroutes` in de MySQL-databank `wandelroutes`! Met de methode `ReadTable()` van onze data-accessklasse `WandelrouteDA` moet dit ons lukken.

Omdat we in het formulier `WandelroutesForm` (project `Presentation`) hiervoor functionaliteit zullen gebruiken die geïmplementeerd werd in het project `DataAccess`, laten we in het formulier `WandelroutesForm` eerst en vooral een using toevoegen naar de namespace `DataAccess`:

```
using DataAccess;
```

We wijzigen de code in `WandelroutesForm` als volgt:

```
public partial class WandelroutesForm : Form
{
    private List<Wandelroute> _wandelroutesLijst;
    private WandelrouteDA _wandelrouteDA;

    public WandelroutesForm()
    {
        InitializeComponent();

        // het veld _wandelroutesLijst initialiseren
        _wandelroutesLijst = new List<Wandelroute>();

        // het veld _wandelrouteDA initialiseren
        _wandelrouteDA = new WandelrouteDA ();

        // via de hulpmethode LijstOpvullen() laten we het veld _wandelroutesLijst
        // opvullen met enkele wandelroutes
        LijstOpvullen();

        // de lijst met wandelroutes opvullen via ons data-accessobject
        _wandelroutesLijst = _wandelrouteDA.ReadTable();
```

```
        // de lijst met wandelroutes weergeven in de listbox
        wandelroutesListBox.DataSource = _wandelroutesLijst;
    }
```

We nemen even de tijd om de nieuwe (vette) instructies te duiden.

Om onze krachtige methode `ReadTable()` te kunnen toepassen, hebben we een object nodig van de data-accesklasse `WandelrouteDA`. `ReadTable()` is immers een methode die in die klasse `WandelrouteDA` gedefinieerd werd.

Daarom laten we in de formulierklasse `WandelroutesForm` zo'n `WandelrouteDA`-object declareren. Het formulier `WandelroutesForm` kreeg er zo het veld `_wandelrouteDA` bij:

```
private WandelrouteDA _wandelrouteDA;
```

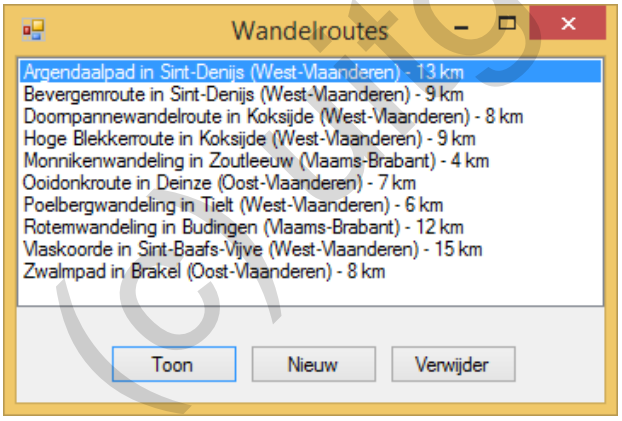
We zeggen dan dat het veld `_wandelrouteDA` dienst doet als **data-accessobject** in het formulier. In de constructor van `WandelroutesForm` laten we dit veld natuurlijk ook initialiseren:

```
_wandelrouteDA = new WandelrouteDA ();
```

De methode `ReadTable()` levert als resultaat een lijst op van `Wandelroute`-objecten (met erin alle wandelroutes uit de databank). Met volgende eenvoudige instructie laten we die lijst van `Wandelroute`-objecten terechtkomen in ons veld `_wandelroutesLijst`:

```
_wandelroutesLijst = _wandelrouteDA.ReadTable();
```

Omdat we tenslotte de lijst `_wandelroutesLijst` gebruiken om de `ListBox` op te vullen, zal je nu - als je de toepassing opstart - volgend formulier te zien krijgen:



Bravo, de 10 wandelroutes die hier gepresenteerd worden, komen nu rechtstreeks uit onze MySQL-tabel `tblwandelroutes`.

Het is nu niet zo moeilijk om een test te bedenken die bewijst dat we hier niet aan het bluffen zijn. Open in **MySQL WorkBench** de databank `wandelroutes` en daarin de tabel `tblwandelroutes`. Breng een wijziging aan bij één of andere wandelroute (en laat deze wijziging toepassen in de MySQL-tabel). Als je daarna de solution `OpStap` opnieuw laat lopen, zou je in de `ListBox` de gewijzigde wandelroute moeten zien!

De hulpmethode `LijstOpvullen()` - waarmee we voorheen enkele voorbeeld-wandelroutes in de lijst stopten - is hiermee overbodig geworden en kan bij het oud ijzer. Je mag de methode `LijstOpvullen()` bijgevolg wissen uit de formulierklasse.

8.4.3 Databankgegevens updaten in de klasse `WandelrouteDA`

We zijn er nu dus al in geslaagd om onze OpStap-toepassing te openen met de wandelroutes uit de MySQL-databank. Zolang het project draait, kunnen we ook zonder problemen wijzigingen aanbrengen aan de wandelroutes en de bijgewerkte routes worden steeds mooi in de `ListBox` op het formulier getoond. Als we de toepassing echter sluiten en opnieuw opstarten, blijkt al ons werk verloren.

Het probleem is dat de wijzigingen die we aanbrengen wel bijgehouden worden in **de lijst** `_wandelroutesLijst`, maar dat deze nog niet doorgeschreven worden naar **de databank**!

Als we dit mogelijk willen maken, zullen we onze data-accessklasse `WandelrouteDA`, die de communicatie met de achterliggende databank regelt, moeten uitbreiden met extra *features*. We stomen een nieuwe methode `void UpdateRecord(Wandelroute)` klaar voor deze taak.

Bij deze methode `Updaterecord()` geven we als parameter een (gewijzigde) wandelroute mee (dus een parameter van het type `Wandelroute`). Deze methode zal in de achterliggende tabel `tblwandelroutes` deze specifieke wandelroute gaan updaten.

De methode `UpdateRecord()` *leest* dus geen gegevens, maar krijgt de kennis om in de databank gegevens te *schrijven*.

Laat ons voor de korte pijn gaan en jullie onmiddellijk de volledige code voorschotelen:

```
public void UpdateRecord(Wandelroute wandelroute)
{
    // SQL-statement om een wandelroute te updaten
    String sql = "UPDATE tblwandelroutes SET naam = @Naam,
                  kilometers = @Kilometers, gemeente = @Gemeente,
                  provincie = @Provincie WHERE (id = @ID);";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@Naam", wandelroute.Naam);
    mySqlCommand.Parameters.AddWithValue("@Kilometers", wandelroute.Kilometers);
    mySqlCommand.Parameters.AddWithValue("@Gemeente", wandelroute.Gemeente);
    mySqlCommand.Parameters.AddWithValue("@Provincie", wandelroute.Provincie);
    mySqlCommand.Parameters.AddWithValue("@ID", wandelroute.Id);

    // de connectie met de databank openen
    _mySqlConnection.Open();

     // ExecuteNonQuery om een MySqlCommand te starten dat GEEN gegevens leest
    mySqlCommand.ExecuteNonQuery();

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();
}
```

"Een databankbewerking uitvoeren", dat betekent dat we opnieuw zullen starten met een **SQL-statement**. We willen gegevens wijzigen in een tabel, dus deze keer geen SELECT-, maar een **UPDATE**-statement.

Om de UPDATE nog eens op te frissen, beginnen we als opwarmer met een SQL-oefening. In `tblwandelroutes` vind je als eerste record de 'Rotemwandeling' terug:

id	naam	kilometers	gemeente	provincie
1	Rotemwandeling	12	Budingen	Vlaams-Brabant

Stel dat de cultuurraad van Budingen deze wandeling hertekent, waardoor de afstand 13 kilometer wordt. Ze willen deze wandeling trouwens voortaan onder de naam 'Zilveren Rotemtocht' kenbaar maken aan het grote publiek.

Een mogelijk SQL-statement om deze wijziging aan te brengen in de databank is:

```
UPDATE      tblwandelroutes
SET         naam = 'Zilveren Rotemtocht', kilometers = 13
WHERE       id = 1;
```

In programmacode zou dit SQL-statement er dan als volgt uitzien:

```
String sql =
          "UPDATE tblwandelroutes SET naam = 'Zilveren Rotemtocht', kilometers = 13
           WHERE ID = 1;";
```

Merk op dat de WHERE-clausule hier aangeeft **welke** wandelroute bijgewerkt moet worden. Zeker niet vergeten (!), want zonder deze WHERE-regel zou je **alle** wandelroutes in de tabel op 13 kilometers zetten en noem je ze **allemaal** 'Zilveren Rotemtocht'.

Met dit, veel te specifieke, SQL-statement zijn we er natuurlijk nog niet in onze methode `UpdateRecord()`. Onze methode moet meer kunnen dan **enkel** de Budingen-wandeling op 13 kilometer te zetten en een meer populaire naam te geven.

Deze denkoefening leverde ons al wel een blauwdruk op van het UPDATE-statement om onze `String sql` variabele in te stellen. Nu nog een nieuwe techniek introduceren, zodat het SQL-statement *elke mogelijke wijziging* bij een wandelroute aankan.

De innovatie waarmee we op de proppen komen, zijn de **@parameters** (spreek uit als: *'at-parameter'*).

In de eigenlijke instructie die we in de methode `UpdateRecord()` gebruiken, staan er vijf zulke @parameters verwerkt:

```
String sql = "UPDATE tblwandelroutes SET naam = @Naam,
                    kilometers = @Kilometers, gemeente = @Gemeente,
                    provincie = @Provincie WHERE (id = @ID);";
```

Door een @parameter in een SQL-statement te zetten, geef je deze impliciet **een naam**. De namen van de @parameters zijn hier: `@Naam`, `@Kilometers`, `@Gemeente`, `@Provincie` en `@ID`.

Een @parameter mag je beschouwen als een *onbekende* of een *variabele* in een SQL-statement. De bedoeling is om in een latere fase deze onbekenden te vervangen door een specifieke waarde.

Je zou bovenstaande SQL-instructie dus kunnen lezen als:

"Update in `tblwandelroutes` de wandelroute waarvan we het `id` op dit moment nog niet kennen en zet daarbij de kolom `naam` op een waarde die nu voor ons nog onbekend is; zet de kolom `kilometers` op een afstand waar we nu nog het raden naar hebben; we zullen ook de kolommen `gemeente` en `provincie` instellen, maar waarop: 'nog geen idee'."

Vooraleer we tonen hoe we de @parameters hun waarde geven, laten we eerst ons SQL-statement, dat nog enkel maar in *tekstformaat* bestaat (variabele `sql`), omzetten naar een `MySQLCommand`. De instructie hiervoor zijn we vroeger al eens tegengekomen:

```
MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

En hiermee zitten de vijf @parameters uit de `String sql` nu ergens in ons `MySqlCommand`-object `mySqlCommand` verwerkt. Weet dat ons commando `mySqlCommand` pas uitgevoerd zal kunnen worden als we aan elke @parameter een concrete waarde hebben gegeven!

We maken eerst nog even het tussenstapje naar onze inleidende 'Rotemwandeltocht' opwarmingsoefening.

Daar moesten we bij de wandelroute met het ID **1** de afstand instellen op **13** kilometer en de naam wijzigen naar **Rotem Zilvertocht**. Dat zouden dus drie **concrete** waarden zijn die op de plaats van drie onbekenden (de @parameters, nl. `@ID`, `@Kilometers` en `@Naam`) in het SQL-statement, moeten komen.

De programmacode om dit te regelen, zou volgende zijn:

```
mySqlCommand.Parameters.AddWithValue("@Naam", "Rotem Zilvertocht");
mySqlCommand.Parameters.AddWithValue("@Kilometers", 13);
mySqlCommand.Parameters.AddWithValue("@ID", 1);
```

De '`MySqlCommand.Parameters.AddWithValue()`'-expressie beschrijft dus **welke @parameter** je laat vervangen door **welke waarde**, bijvoorbeeld de @parameter `@Kilometers` wordt de waarde 13.

Als we de drie aangegeven subsituties uitvoeren in het originele SQL-statement (met de @parameters), dan bekomen we volgende SQL-code.

```
"UPDATE tblwandelroutes SET naam = 'Rotem Zilvertocht',
                kilometers = 13, gemeente = @Gemeente,
                provincie = @Provincie WHERE (id = 1);";
```

Als je elke @parameter een eigenlijke waarde geeft, dan bekom je een geldig (concreet) UPDATE-statement, waar de MySQL-server raad mee weet.

Onthoud 8.6 De instructie om een **@parameter** uit een MySqlCommand-object **te vervangen** door een **specifieke waarde**, is van de vorm:

`object.Parameters.AddWithValue("@parameternaam", waarde)`

Met hierbij:

- `object` --> het `MySqlCommand`-object waarvan je een @parameter een waarde wilt geven;
- `@parameternaam` --> welke @parameter wil je een waarde geven;
- `waarde` --> de waarde die je wilt toekennen aan de @parameter.

Nu je goed weet wat @parameters zijn en snapt hoe je deze @parameters een concrete waarde geeft, kunnen we nog eens kijken naar onze definitieve constructie in de methode `UpdateRecord()`.

Heel belangrijk is dat we bij de header van de methode `UpdateRecord()` een parameter vragen van het type `Wandelroute` De naam van deze parameter is `wandelroute`.

```
public void UpdateRecord(Wandelroute wandelroute) {}
```

De wandelroute die we via deze parameter binnenkrijgen, is net de wandelroute die we in `tblwandelroutes` moeten updaten!

In het `mySqlCommand` zullen we daarom de @parameters laten vervangen door de juiste gegevens die we aan deze parameter `wandelroute` opvragen:

```
mySqlCommand.Parameters.AddWithValue("@Naam", wandelroute.Naam);
mySqlCommand.Parameters.AddWithValue("@Kilometers", wandelroute.Kilometers);
mySqlCommand.Parameters.AddWithValue("@Gemeente", wandelroute.Gemeente);
mySqlCommand.Parameters.AddWithValue("@Provincie", wandelroute.Provincie);
mySqlCommand.Parameters.AddWithValue("@ID", wandelroute.Id);
```

Bijvoorbeeld de laatste instructie hierboven betekent dat we in ons SQL-statement op de plaats van de @parameter met de naam `@ID`, de waarde van de Property `Id` van de parameter `wandelroute` zullen zetten. Hiermee leggen we in het UPDATE-statement m.a.w. vast **welke** wandelroute er in `tblWandelroutes` geüpdatet moet worden.

Met de vier andere `AddWithValue`-instructies zorgen we dat we in het SQL-statement de @parameters instellen die bepalen welke nieuwe waarden de velden `naam`, `kilometers`, `gemeente` en `provincie` krijgen. De Properties `Naam`, `Kilometers`, `Gemeente` en `Provincie` van de parameter `wandelroute` vertellen wat die concrete nieuwe waarden moeten zijn.

We finaliseren het SQL-statement dus door onderstaande vet cursieve items te injecteren op de plaats van de @parameters:

```
"UPDATE tblwandelroutes SET naam = 'wandelroute.Naam',
          kilometers = wandelroute.Kilometers, gemeente = 'wandelroute.Gemeente',
          provincie = 'wandelroute.Provincie' WHERE (id = wandelroute.Id);";
```

Nu het SQL-statement in `mySqlCommand` helemaal op punt staat, volgen de laatste instructies als vanzelf.

Om effectief het `MySqlCommand` te kunnen opstarten (en dus de update in de tabel uit te voeren), moeten we eerst nog de connectie openen:

```
_mySqlConnection.Open();
```

Een `MySQLCommand` waarmee je geen gegevens leest, maar **waarmee je een INSERT-, UPDATE- of DELETE-statement uitvoert**, start je op via de methode `ExecuteNonQuery()`:

```
mySqlCommand.ExecuteNonQuery();
```

Onthoud 8.7 Een `MySqlCommand`-object omvat een SQL-statement en is in staat om dit SQL-statement uit te voeren op een databank waarvoor een connectie geopend werd.

Als dit SQL-statement geen lees-operatie (SELECT) is, maar op één of andere manier **gegevens wijzigt** in de databank (**UPDATE**, **INSERT** of **DELETE**), dan moet je hiervoor de methode `ExecuteNonQuery()` toepassen het `MySqlCommand`-object.

De connectie terug laten sluiten is de allerlaatste actie die we in de methode `UpdateRecord()` laten uitvoeren:

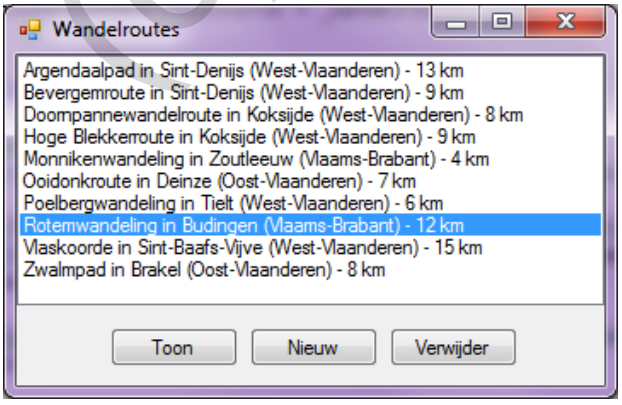
```
_mySqlConnection.Close();
```

De methode `UpdateRecord()` toepassen

Onze data-accessklasse `WandelrouteDA` is nu dus uitgebreid met de extra methode `UpdateRecord()` waarmee we een wandelroute in de tabel `tblwandelroutes` kunnen updaten. De wandelroute zoals we die geüpdatet willen zien, moeten we hiervoor gewoon meegeven als parameter bij deze methode `UpdateRecord()`.

Nu eens kijken om deze methode te integreren in onze wandelroute-formulieren, zodat wandelroutes die we wijzigen nu effectief ook in de databank aangepast worden.

Hoe werkte `WandelroutesForm` ook al weer?



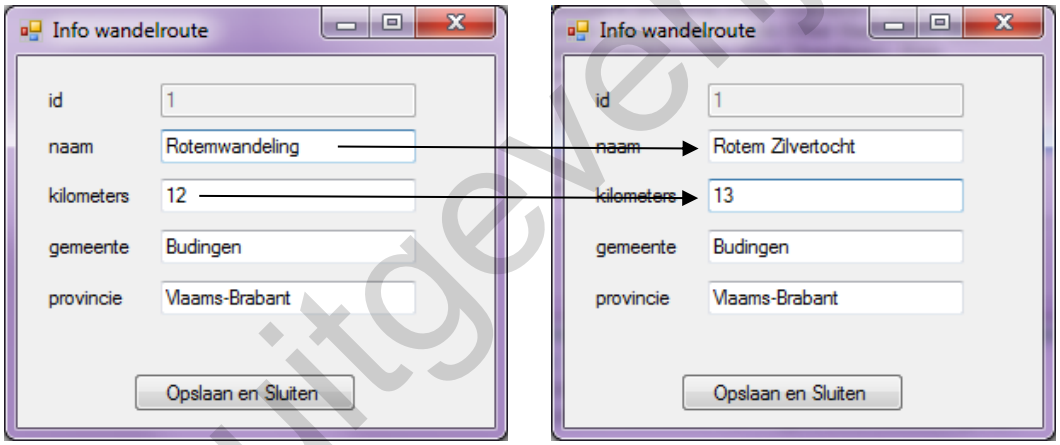
Achter het knopje 'Toon' op `WandelroutesForm` staat de code die voor ons interessant is:

```
private void toonButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute in de ListBox opvragen
    Wandelroute wandelroute = (Wandelroute)wandelroutesListBox.SelectedItem;

    // het formulier WandelrouteInfoForm openen met de geselecteerde wandelroute
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // misschien heeft gebruiker gegevens van de wandelroute veranderd
    // --> ListBox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

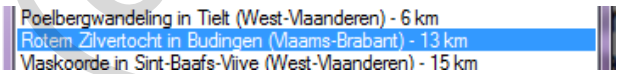
Bovenstaand stukje code, kunnen we in volgende belangrijke stappen opdelen:

- We vragen eerst op welke wandelroute in de `ListBox` geselecteerd staat.
- De gevonden wandelroute geven we door aan het formulier `WandelrouteInfoForm` dat we als een dialoogvenster laten openen.

  De gebruiker kan in `WandelrouteInfoForm` de wandelroute wijzigen en laten opslaan.



- Eens het formuliertje gesloten wordt, loopt de code in de methode `toonButton_Click` weer verder. We laten er de `ListBox` vernieuwen, zodat de bijgewerkte wandelroute daar ook geüpdatet wordt:



Opgelet: In de `ListBox` wordt de 'vernieuwde' Rotem-wandeling correct getoond (de wandelroute in de `ListBox` is immers bijgewerkt), maar deze wijziging is nog **NIET** doorgeschreven naar de databank.

De toepassing sluiten en openen, maakt dat we terug de originele Rotem-wandeling krijgen. Ergens zullen we dus nog de opdracht moeten geven om de wandelroute ook te laten **updaten in de databank**!

Het wijzigen van de wandelroute gebeurt in `WandelrouteInfoForm`, dus eens dit formulier gesloten werd, heeft de gebruiker zijn aanpassingen al aangebracht. We laten daarom de wandelroute in de databank updaten ná de `ShowDialog()`-instructie.

De nieuwe, toe te voegen, instructie staat hieronder in het vet:

```
private void toonButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute in de ListBox opvragen
    Wandelroute wandelroute = (Wandelroute)wandelroutesListBox.SelectedItem;

    // het formulier WandelrouteInfoForm openen met de geselecteerde wandelroute
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // de (gewijzigde) wandelroute wegschrijven naar de databank
    _wandelrouteDA.UpdateRecord(wandelroute);

    // misschien heeft gebruiker gegevens van de wandelroute veranderd
    // --> ListBox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Veel extra code hebben we dus niet nodig om hier de koppeling met de databank in orde te krijgen.

`_wandelrouteDA` is het **data-accessobject** dat we reeds eerder in dit formulier gedeclareerd en geïnitialiseerd hadden. Via dit object kunnen we de databank-acties oproepen die we in de klasse `WandelrouteDA` programmeerden.

We hoeven in ons formulier dus enkel maar de methode `UpdateRecord()` op te roepen bij dit data-accessobject `_wandelrouteDA` en geven hierbij die (gewijzigde) wandelroute mee als argument.

```
_wandelrouteDA.UpdateRecord(wandelroute);
```

8.4.4 Databankgegevens toevoegen in de klasse `WandelrouteDA`

De volgende databankbewerking die we willen aanpakken in de data-accessklasse `WandelrouteDA` is het toevoegen van nieuwe wandelroutes aan `tblwandelroutes`.

Wedden dat onderstaande code bij de nieuwe methode `void CreateRecord(Wandelroute)` al niet meer zo vreemd zal aanvoelen:

```
public void CreateRecord(Wandelroute wandelroute)
{
    // SQL-statement om een wandelroute toe te voegen
    String sql =
             "INSERT INTO tblwandelroutes(naam, kilometers, gemeente, provincie)
              VALUES(@Naam, @Kilometers, @Gemeente, @Provincie);";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

```
    // parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@Naam", wandelroute.Naam);
    mySqlCommand.Parameters.AddWithValue("@Kilometers", wandelroute.Kilometers);
    mySqlCommand.Parameters.AddWithValue("@Gemeente", wandelroute.Gemeente);
    mySqlCommand.Parameters.AddWithValue("@Provincie", wandelroute.Provincie);

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // ExecuteNonQuery om een MySqlCommand te starten dat geen gegevens inleest
    mySqlCommand.ExecuteNonQuery();

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();
}
```

De kern van elke data-accessmethode is het juiste **SQL-statement**. Omdat we met deze methode een **nieuwe record toevoegen**, zullen we werken met een **INSERT**-statement.

Eerst een voorbeeldje:

De Muziekbosroute zou één van de mooiste uitgepijlde wandelroutes in Vlaanderen zijn. Deze route is 11 km lang en start in Louise Marie (Oost-Vlaanderen).

Het SQL-statement om deze route aan `tblwandelroutes` toe te voegen:

```
INSERT INTO  tblwandelroutes(naam, kilometers, gemeente, provincie)
VALUES       ('Muziekbosroute', 11, 'Louise Marie',
             'Oost-Vlaanderen')
```

Merk op dat we in dit statement het `id`-veld volledig negeren. `id` is immers een **auto-nummeringsveld** dat onze MySQL-server automatisch een unieke waarde zal geven.

In programmacode zouden we dit SQL-statement, als volgt verwerken:

```
String sql =
          "INSERT INTO tblwandelroutes(naam, kilometers, gemeente, provincie)
           VALUES ('Muziekbosroute', 11, 'Louise Marie', 'Oost-Vlaanderen')";
```

Maar de taak van onze methode `CreateRecord()` is niet om 'de Muziekbosroute' toe te voegen aan `tblwandelroutes`. We moeten de methode `CreateRecord()` zo opbouwen, zodat we ermee **om het even welke** wandelroute in `tblwandelroutes` krijgen.

De truc is opnieuw om in deze fase **@parameters** in het SQL-statement op te nemen.

```
String sql = "INSERT INTO tblwandelroutes(naam, kilometers, gemeente, provincie)
              VALUES(@Naam, @Kilometers, @Gemeente, @Provincie);";
```

Je leest bovenstaand SQL-statement als:

"Voeg aan `tblwandelroutes` een nieuwe record toe, maar welke waarden je hierbij in de kolommen `naam`, `kilometers`, `gemeente` en `provincie` moet schrijven, is nog onbekend."

In onze code vormen we het SQL-statement (in tekstformaat) eerst opnieuw om naar een `MySqlCommand`-object:

```
MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

Alvorens het `MySqlCommand` effectief uit te voeren, moeten we de onbekenden (de @parameters) hun waarde geven.

Om nog even naar ons voorbeeldje met de 'Muziekbosroute' terug te keren. Daarvoor zouden we de @parameters als volgt moeten substitueren:

```
mySqlCommand.Parameters.AddWithValue("@Naam", "Muziekbosroute");
mySqlCommand.Parameters.AddWithValue("@Kilometers", 11);
mySqlCommand.Parameters.AddWithValue("@Gemeente", "Louise Marie");
mySqlCommand.Parameters.AddWithValue("@Provincie", "Oost-Vlaanderen");
```

De bedoeling van de methode `CreateRecord()` is echter om de wandelroute toe te voegen die via de parameter `wandelroute` aangeleverd wordt! Zie de header van de methode:

```
public void CreateRecord(Wandelroute wandelroute) {}
```

We gebruiken bijgevolg de nodige gegevens van die parameter `wandelroute` om de @parameters in `mySqlCommand` te vervangen:

```
mySqlCommand.Parameters.AddWithValue("@Naam", wandelroute.Naam);
mySqlCommand.Parameters.AddWithValue("@Kilometers", wandelroute.Kilometers);
mySqlCommand.Parameters.AddWithValue("@Gemeente", wandelroute.Gemeente);
mySqlCommand.Parameters.AddWithValue("@Provincie", wandelroute.Provincie);
```

De Properties van de parameter `wandelroute` worden dus in feite als volgt in ons SQL-statement geïnjecteerd:

```
"INSERT INTO tblwandelroutes(naam, kilometers, gemeente, provincie)
 VALUES ('wandelroute.Naam', wandelroute.Kilometers, 'wandelroute.Gemeente',
         'wandelroute.Provincie');";
```

Nu `mySqlCommand` klaar staat, rollen we de laatste instructies uit.

We openen de connectie:

```
_mySqlConnection.Open();
```

We laten `mySqlCommand` opstarten. Een **INSERT**-statement (net als een UPDATE en DELETE), voer je uit via de methode `executeNonQuery()`:

```
mySqlCommand.ExecuteNonQuery();
```

Het is dus deze laatste instructie die de nieuwe wandelroute effectief in de tabel laat zetten.

Tenslotte, om alles mooi op te ruimen, laten we de connectie nog sluiten:

```
_mySqlConnection.Close();
```

De methode `CreateRecord()` toepassen

Er tekent zich ondertussen een duidelijk stramien af. We gaan weer op zoek waar we in onze OpStap-toepassing de nieuwe `CreateRecord()`-methode moeten oproepen, zodat de nieuwe wandelroutes niet enkel in de `ListBox`, maar ook in de **databank** terecht komen.

Om een nieuwe wandelroute toe te voegen, gebruiken we in `WandelroutesForm` het knopje 'Nieuw'.

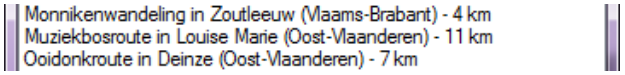
Hieronder de betreffende programmacode bij deze knop 'Nieuw':

```
private void nieuwButton_Click(object sender, EventArgs e)
{
    // wat is het hoogste ID van een wandelroute in de lijst _wandelroutesLijst
    int maxId = _wandelroutesLijst.Max(x => x.Id);

    // nieuw Wandelroute-object aanmaken
    Wandelroute wandelroute = new Wandelroute(maxId+1, "", 0, "", "");

    // nieuwe wandelroute toevoegen aan lijst wandelroutes
    _wandelroutesLijst.Add(wandelroute);

    // het formulier WandelrouteInfoForm openen
    // we geven de nieuwe wandelroute mee als parameter
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // de listbox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

We onderscheiden de volgende belangrijke stappen:

- We laten een nieuw (leeg) `Wandelroute`-object aanmaken, dat we onmiddellijk toevoegen aan de lijst met wandelroutes (`_wandelroutesLijst`). Daarvóór hadden we opgezocht welk (uniek volgend) id we aan deze wandelroute moeten toekennen.
- De nieuwe (nog lege) wandelroute wordt doorgegeven aan het formulier `WandelrouteInfoForm`, zodat de gebruiker de kans krijgt om de gegevens in te tikken.

- Tenslotte laten we de inhoud van de `ListBox` op het formulier `WandelroutesForm` vernieuwen, zodat de nieuwe wandelroute ook in het lijstje staat.



Om er nu voor te zorgen dat de nieuwe wandelroute ook **in de databank** terechtkomt, bouwen we de code achter het knopje 'Nieuw' een beetje om. 't Wordt net iets meer werk dan op het eerste gezicht gedacht:

```
private void nieuwButton_Click(object sender, EventArgs e)
{
    // wat is het hoogste ID van een wandelroute in de lijst _wandelroutesLijst
    int maxId = _wandelroutesLijst.Max(x => x.Id);

    // nieuw Wandelroute-object aanmaken
    Wandelroute wandelroute = new Wandelroute(0, "", 0, "", "");

    // nieuwe wandelroute toevoegen aan lijst wandelroutes
    _wandelroutesLijst.Add(wandelroute);

    // het formulier WandelrouteInfoForm openen
    // we geven de nieuwe wandelroute mee als parameter
    WandelrouteInfoForm formulier = new WandelrouteInfoForm(wandelroute);
    formulier.ShowDialog();

    // de (nieuwe) wandelroute toevoegen aan de tabel
    _wandelrouteDA.CreateRecord(wandelroute);

    // de lijst met wandelroutes terug inlezen uit de databank
    _wandelroutesLijst = _wandelrouteDA.ReadTable();

    // de listbox vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Onderstaande instructie voert het belangrijkste werk uit:

```
_wandelrouteDA.CreateRecord(wandelroute);
```

`_wandelrouteDA` is het **data-accessobject** in het formulier. Via dit object kunnen we de databank-acties oproepen die we in de klasse `WandelrouteDA` programmeerden.

We roepen de methode `CreateRecord()` op bij dit data-accessobject `_wandelrouteDA` om de nieuwe wandelroute in de databank op te slaan. Die nieuwe wandelroute geven we hiervoor mee als parameter bij deze methode.

Deze instructie plaatsten we na de `ShowDialog()`-regel, omdat de gebruiker dan al de kans gekregen heeft om de gegevens van de nieuwe wandelroute aan te vullen in het formulier `WandelrouteInfoForm`.

Dat we nog wat meer moesten sleutelen aan de code achter de knop 'Nieuw', heeft alles te maken met het **`Id`-veld** van de nieuwe wandelroute! We proberen hieronder deze zaak stap voor stap te kaderen:

In de tabel `tblwandelroutes` is de kolom `id` een **autonummeringsveld**. Wij zijn dus niet meer zelf verantwoordelijk om een wandelroute een uniek id te geven, maar laten dit nu over aan MySQL.

Vandaar dat de instructie waar wij zelf het volgende id opzochten, geschrapt mag worden (is niet meer *ons* werk):

```
int maxId = _wandelroutesLijst.Max(x => x.Id);
```

Als we een nieuw `Wandelroute`-object aanmaken, moeten we echter nog steeds de id-parameter opgeven (dit is de eerste parameter bij de constructor van de klasse `Wandelroute`). De parameters bij een constructor zijn immers verplicht in te vullen! We zetten daar maar een *voorlopige* 0:

```
Wandelroute wandelroute = new Wandelroute(0, "", 0, "", "");
```

Op het moment dat de wandelroute **in de databank** ingevoerd wordt, d.m.v. de `CreateRecord()`-instructie, krijgt de wandelroute er zijn definitieve id toegekend! Die *voorlopige* 0 die we bij het `Wandelroute`-object ingesteld hebben, wordt in `CreateRecord()` straal genegeerd (kijk er het SQL-statement maar eens na).

Let op … dit laatste gebeurt **énkel** in de tabel `tblwandelroutes` op onze MySQL-server. In ons Visual Studio project - waar de nieuwe wandelroute ook opgenomen is in de lijst `_wandelroutesLijst` - weten we niet welk id in de tabel gekozen is. In het `Wandelroute`-object in de lijst staat het id dus nog steeds op de waarde 0!

De enige manier die we kunnen bedenken om ook in de lijst `_wandelroutesLijst` het juiste id bij de nieuwe wandelroute te zetten, is door er de tabel `tblwandelroutes` uit de databank **nog eens in te lezen**. Dit kan met de methode `ReadTable()`.

Daarom dat we aan de methode `CreateRecord()` volgende extra instructie lieten toevoegen om de lijst `_wandelroutesLijst` nog eens helemaal te synchroniseren met de tabel `tblwandelroutes` in de databank.

```
_wandelroutesLijst = _wandelrouteDA.ReadTable();
```

Omdat we op deze manier nog eens alle wandelroutes, inclusief de nieuwe wandelroute, uit de databank inlezen (om deze vervolgens in de `ListBox` te tonen), kunnen we in principe onze instructie schrappen waar wij zelf het nieuwe `Wandelroute`-object aan de lijst met de wandelroutes lieten toevoegen:

```
_wandelroutesLijst.Add(wandelroute);
```

8.4.5 Databankgegevens wissen in de klasse `WandelrouteDA`

De volgende basis-databankbewerking die we nog willen ondersteunen via de data-access-klasse `WandelrouteDA` is het **wissen** van wandelroutes.

We pakken dit onderdeeltje aan in de nieuwe methode `void DeleteRecord(int)`. Als parameter geven we deze keer gewoon een `int` mee (en dus niet een volledig `Wandelroute`-object). Deze `int`-parameter geeft het id aan van de wandelroute die we willen wissen.

Nu we al zo ver gekomen zijn in dit 8e hoofdstuk, zou onderstaande code jullie al heel vertrouwd moeten lijken:

```
public void DeleteRecord(int id)
{
    // SQL-statement om een wandelroute te wissen
    String sql = "DELETE FROM tblwandelroutes WHERE (id = @ID);";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@ID", id);

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // ExecuteNonQuery om een MySqlCommand te starten dat geen gegevens inleest
    mySqlCommand.ExecuteNonQuery();

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();
}
```

Het begint weer met het opstellen van het juiste **SQL-statement**. Gegevens wissen uit een tabel gebeurt met een **DELETE**.

Stel bijvoorbeeld dat we de 'Monnikenwandeling' uit `tblwandelroutes` willen wegdoen:

id	naam	kilometers	gemeente	provincie
1	Rotem Zilvertocht	13	Budingen	Vlaams-Brabant
2	Monnikenwandeling	4	Zoutleeuw	Vlaams-Brabant
3	Argendaalpad	13	Sint-Denijs	West-Vlaanderen

Dit zal ons lukken met volgend statement:

```
DELETE FROM  tblWandelroutes
WHERE        (id = 2);
```

Merk op dat de WHERE-clausule bij een DELETE-statement *levensbelangrijk* is! Zonder de WHERE maak je niet duidelijk welke records je wilt wissen, met als resultaat dat de **volledige tabel** leeggemaakt wordt!

In programmacode-formaat schrijven we bovenstaand SQL-statement als:

```
String sql = "DELETE FROM tblwandelroutes WHERE (id = 2);";
```

Maar deze instructie liquideert steeds die éne wandelroute waarvoor het id gelijk is aan 2 in `tblwandelroutes`. Omdat we met de methode `DeleteRecord()` ook andere wandelroutes uit de tabel moet kunnen wissen, voegen we een @parameter toe:

```
String sql = "DELETE FROM tblwandelroutes WHERE (id = @ID);";
```

Lees bovenstaand SQL-statement als:

"Wis uit `tblWandelroutes` de record waarvoor de kolom `id` op een bepaalde waarde staat die we nu nog niet kennen".

Eerst nog eventjes van het SQL-statement uit de `String`-variabele `sql` een echt `MySqlCommand`-object maken:

```
MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

Een `MySqlCommando` kunnen we pas uitvoeren als we alle @parameter (de *onbekenden*) hun eigenlijke waarde gegeven hebben. Ons object `mySqlCommand` bevat één zo'n @parameter, die we `@ID` genoemd hebben.

Als we de 'Monnikenwandeling' kwijt willen in `tblwandelroutes`, zouden we de parameter `@ID` op 2 moeten instellen in het SQL-statement, of dus:

```
mySqlCommand.Parameters.AddWithValue("@ID", 2);
```

Welke specifieke record we 'echt' moeten wissen, wordt in de methode `Delete-Record()` aangegeven met de `int`-parameter `id`. Zie de header van deze methode:

```
public void DeleteRecord(int id) {}
```

Het is dus de waarde die we meekrijgen via deze parameter `id` die de plaats moet innemen van de @parameter `@ID`:

```
mySqlCommand.Parameters.AddWithValue("@ID",id);
```

Niet vergeten om de connectie open te zetten:

```
_mySqlConnection.Open();
```

Ons object `mySqlCommand` bevat deze keer een **DELETE**-statement. Zo'n statement laat je ook opstarten met de methode `ExecuteNonQuery()`.

```
mySqlCommand.ExecuteNonQuery();
```

Pas als bovenstaande instructie uitgevoerd werd, is de DELETE-actie in de databank effectief gebeurd (en werd de gevraagde wandelroute gewist).

Ons werk zit er helemaal op als we de connectie met de databank terug gesloten hebben:

```
_mySqlConnection.Close();
```

De methode `DeleteRecord()` toepassen

We keren nog eens terug naar onze OpStap-toepassing om ook het wissen van wandelroutes de nodige **databank**-ondersteuning te geven.

In het formulier `WandelroutesForm` hebben we via het knopje 'Verwijder'al de mogelijkheid om wandelroutes uit de `ListBox` te wissen.

Deze knop wordt op dit moment aangestuurd via volgende code:

```
private void verwijderButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute opvragen
    Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;

    // de geselecteerde wandelroute uit de lijst laten verwijderen
    _wandelroutesLijst.Remove(wandelroute);

    // de listbox laten vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Je herkent hier volgende bewerkingen:

- We bepalen eerst welke wandelroute in de `ListBox` geselecteerd staat.
- Deze wandelroute wordt uit de lijst `_wandelroutesLijst` verwijderd.
- We laten de `ListBox` op het formulier vernieuwen door de lijst `_wandelroutesLijst` nog eens opnieuw naar de `ListBox` te schrijven.

  Pas zo zal de 'gewiste' wandelroute ook uit de `ListBox` verdwenen zijn.

Om onze **databank** up-to-date te houden, moeten we een wandelroute die we in het formulier `WandelroutesForm` wissen, ook steeds gaan verwijderen uit `tblwandelroutes` op de MySQL-server. Als we dit niet doen, komen deze wandelroutes immers steeds terug na het stoppen en starten van de toepassing.

Het is dus duidelijk dat we hiervoor ons **data-accessobject** `_wandelrouteDA` weer aan het werk moeten zetten, aangezien we via dit object alle databank-bewerkingen regelen.

Het is voldoende om de code achter het knopje 'Verwijder' op `WandelroutesForm` hiervoor één instructie langer maken:

```
private void verwijderButton_Click(object sender, EventArgs e)
{
    // de geselecteerde wandelroute opvragen
    Wandelroute wandelroute = (Wandelroute) wandelroutesListBox.SelectedItem;

    // de geselecteerde wandelroute uit de lijst laten verwijderen
    _wandelroutesLijst.Remove(wandelroute);

    // de wandelroute verwijderen uit de tabel
    _wandelrouteDA.DeleteRecord(wandelroute.Id);

    // de listbox laten vernieuwen
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Met de nieuwe methode `DeleteRecord()` van ons data-accessobject `_wandelrouteDA` kunnen we een rij laten wissen in `tblwandelroutes`, maar dan moeten we aan deze methode wel als parameter het id meegeven van de te verwijderen record.

Daarom dat we aan de wandelroute die we reeds in het formulier uit de lijst gewist hebben (en die we aanspreken met de variabele `wandelroute`), vragen wat zijn id is. We hebben daar meer dan 30 pagina's geleden een Property `Id` voor geprogrammeerd.

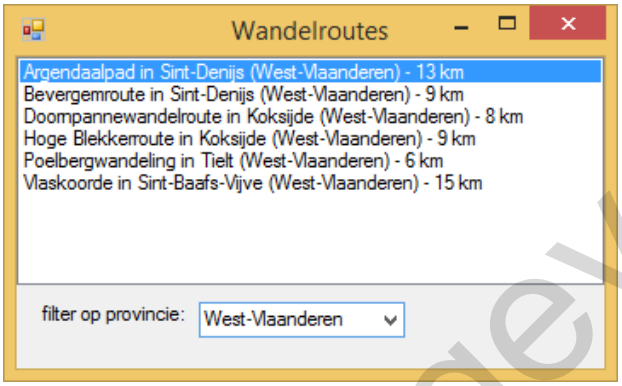
```
_wandelrouteDA.DeleteRecord(wandelroute.Id);
```

Als je nu de 'Budingenwandeling' in het formulier verwijdert, keert die nooit meer terug.

8.4.6 Databankgegevens lezen met een filter

Het principe van de @parameters in een SQL-statement is ondertussen gekend. We hebben immers met lichte dwang al drie voorbeeldjes onder jullie neus geduwd (eens met een UPDATE-, eens met een INSERT- en eens met een DELETE-statement).

Ook bij lees-operaties in een databank (SELECT-statement) kunnen @parameters van pas komen. Om dit met een nuttig voorbeeld te staven, verleggen we onze focus even naar het formulier `WandelroutesFilterForm`.



De bedoeling is dat we in dit formulier de wandelroutes kunnen opvragen **per provincie**. De gebruiker kan in de **keuzelijst** zelf bepalen welke provincie dit moet zijn.

Opmerking:

- Om `WandelroutesFilterForm` straks in de toepassing te kunnen openen, gaan we even moeten *prutsen* in het bestand **program.cs** in het project `Presentation`.

  Laat er het formulier `WandelroutesFilterForm` instellen als startformulier:

```
Application.Run(new WandelroutesFilterForm());
```

In de klasse `WandelroutesDA`

Alle communicatie met de databank gebeurt in de Data Access Layer. Wij gebruiken de data-accessklasse `WandelroutesDA` om alle databankacties met de tabel `tblWandelroutes` mogelijk te maken.

Deze keer zijn we op zoek naar een methode die ons een lijst oplevert van die wandelroutes uit de databank die in een bepaalde provincie liggen. Bij deze methode zou je op één of andere manier moeten kunnen opgeven over welke provincie het dan precies gaat.

Als de methode `ReadTable()` (zie pagina 256) een lijst met **alle** wandelroutes retourneert, hengelen we nu naar een methode die een **gefilterde** lijst van wandelroutes teruggeeft.

We willen jullie eerst de volledige code voorstellen van onze nieuwe methode `ReadTable-FilterProvincie()` in de klasse `WandelroutesDA`, om er daarna enkele interessante fragmentjes uit te lichten. Zo zie je meteen ook dat deze nieuwe methode veel gemeen heeft met de methode `ReadTable()`.

```
public List<Wandelroute> ReadTableFilterProvincie(String provincie)
{
    List<Wandelroute> lijst = new List<Wandelroute>();

    // SQL-statement om de wandelroutes gefilterd op een provincie op te vragen
    String sql = "SELECT * FROM tblwandelroutes WHERE provincie = @Provincie
                  ORDER BY naam;";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // parameter in het SQL-commando een waarde geven
    mySqlCommand.Parameters.AddWithValue("@Provincie", provincie);

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // met ExecuteReader laat je een leescommando opstarten
    // ingelezen informatie komt in mySqlDataReader terecht
    MySqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

    // lusje om alle records in mySqlDataReader te overlopen
    while (mySqlDataReader.Read() == true)
    {
        // nieuwe wandelroute maken met de actieve record in mySqlDataReader
        Wandelroute wandelroute = new Wandelroute((int)(mySqlDataReader["id"]),
                                  mySqlDataReader["naam"].ToString(),
                                  (int)(mySqlDataReader["kilometers"]),
                                  mySqlDataReader["gemeente"].ToString(),
                                  mySqlDataReader["provincie"].ToString());

        // voeg de wandelroute toe aan de lijst
        lijst.Add(wandelroute);
    }

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();

    // lijst met alle wandelroutes teruggeven
    return lijst;
}
```

Eerst eens naar de header van onze nieuwe methode kijken:

```
public List<Wandelroute> ReadTableFilterProvincie(String provincie){}
```

We kunnen hieruit al heel wat afleiden:

- We noemden de methode `ReadTableFilterProvincie`. We lezen immers gegevens uit de tabel `tblwandelroutes` (--*ReadTable*--), maar halen enkel de wandelroutes van een gegeven provincie op (--*FilterProvincie*--).

- Als parameter geven we bij deze methode een `String` mee die de provincie zal aangeven waarvan we de wandelroutes willen selecteren.
- Omdat er bij een provincie meerdere wandelroutes kunnen horen, laten we deze methode een `List` van `Wandelroute`-objecten retourneren.

Stel dat we jullie vragen om het SQL-statement neer te pennen waarmee je alle wandelroutes uit de tabel `tblWandelroutes` selecteert die in de provincie West-Vlaanderen liggen, dit gesorteerd op hun naam. Ongetwijfeld heb je dan ook onderstaande regels op papier gezet:

```
SELECT     *
FROM       tblwandelroutes
WHERE      provincie='West-Vlaanderen'
ORDER BY   naam;
```

Dit is hoe we dit SQL-statement in onze programmacode opslaan in een `String`-variabele:

```
String sql = "SELECT * FROM tblwandelroutes WHERE provincie = 'West-Vlaanderen'
              ORDER BY naam;";
```

Wij moeten er echter rekening mee houden dat onze methode niet enkel de wandelroutes van 'West-Vlaanderen' moet ophalen, maar dat dit SQL-statement voor om het even welke provincie moet werken.

Vandaar dat we in onze programmacode een @parameter verwerken in het SQL-statement:

```
String sql = "SELECT * FROM tblwandelroutes WHERE provincie = @Provincie
              ORDER BY naam;";
```

Lees bovenstaande instructie als:

"Geef alle wandelroutes uit `tblwandelroutes`, gesorteerd op naam, gesitueerd in een bepaalde provincie, maar welke provincie dit dan precies is, dat weten we nog niet."

Alvorens we iets verder in de procedure het `MySqlCommand` effectief uitvoeren, zullen we eerst de @parameter concreet moeten maken.

Op de plaats waar nu deze @parameter `@Provincie` staat in het SQL-statement, moet de eigenlijke provincie komen, waarvoor we de wandelroutes willen selecteren. M.a.w. laten we `@Provincie` vervangen door de parameter die we meegaven bij de methode `ReadTableFilterProvincie()`, want net via die parameter wordt de bewuste provincie doorgegeven.

Vandaar de instructie:

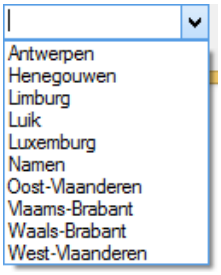
```
mySqlCommand.Parameters.AddWithValue("@Provincie", provincie);
```

Voor meer uitleg over de overige code in deze methode (en dat is er heel wat) verwijzen we jullie graag door naar het onderdeel '8.4.2 Databankgegevens lezen in de klasse `WandelrouteDA`' (zie pagina 256 en volgende).

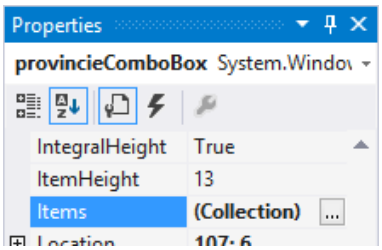
De methode `ReadTableFilterProvincie()` toepassen

Het formulier `WandelrouteFilterForm` vormt de presentatielaag waar de gebruiker de wandelroutes per provincie kan bekijken.

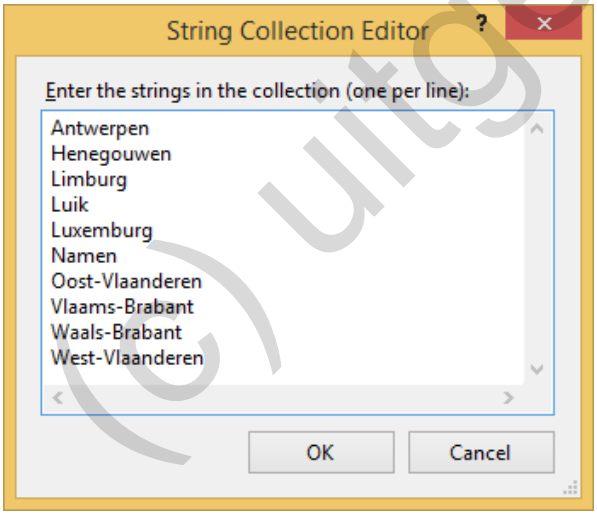
Op dit formulier hebben we voor een keuzelijst (`ComboBox`) gezorgd met alle provincies:



Om waarden aan een keuzelijst toe te voegen, ga je in het vensters Properties naar de eigenschap `Items`:



Je klikt er op het knopje met de drie puntjes en komt in volgend venster terecht, waar alle items uit de keuzelijst opgesomd worden:



Opmerking:

- In bovenstaand venster tikten we gewoon één na één alle provincies in. Een veel beter alternatief zou natuurlijk zijn om ook de provincies uit de databank te lezen! Op dit moment zou ons dit echter iets te ver leiden …

  Een keuzelijst opvullen met databankgegevens komt in de toekomst zeker wel nog eens aan bod in één of andere extra oefening, of toets, of proefwerk.

Bij de programmacode bij `WandelroutesFilterForm` eerst eens nagaan of bovenaan de twee volgende using-statements voorkomen:

```
using Business;
using DataAccess;
```

We zullen in dit formulier immers gebruik maken van de klasse `Wandelroute` (Business Layer) en `WandelrouteDA` (Data Access Layer) en deze klassen zijn ondergebracht in andere projecten! Dit is een prijs die we graag betalen om onze solution te compartimenteren in logische projecten.

Hieronder een eerste blokje code bij de klasse `WandelroutesFilterForm`:

```
public partial class WandelroutesFilterForm : Form
{
    private List<Wandelroute> _wandelroutesLijst;
    private WandelrouteDA _wandelrouteDA;

    public WandelroutesFilterForm()
    {
        InitializeComponent();

        // het veld _wandelroutesLijst initialiseren
        _wandelroutesLijst = new List<Wandelroute>();

        // het veld _wandelrouteDA initialiseren
        _wandelrouteDA = new WandelrouteDA();
    }
```

In bovenstaand stukje code laten we een lijst van `Wandelroute`-objecten declareren en initialiseren. In deze lijst `_wandelroutesLijst` houden we straks de gevonden wandelroutes bij.

Om toegang tot de databank te krijgen, declareren en initialiseren we een `WandelrouteDA`-veld. Dit **data-accessobject** noemen we `_wandelrouteDA`.

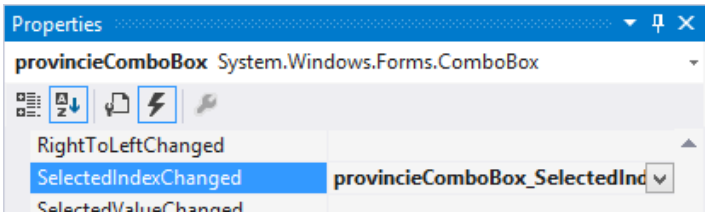
Merk op dat we in de constructor dus wel onze objecten initialiseren, maar nog GEEN databankgegevens ophalen. Bij het openen van het formulier zullen er m.a.w. nog geen wandelroutes te zien zijn in de `ListBox`.

Elke keer als de gebruiker in het keuzelijstje een andere provincie selecteert, zouden we in de `ListBox` de wandelroutes van die provincie moeten tonen. Dit laatste regelen we met volgende methode:

```
private void provincieComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    // de lijst op basis van filter opvullen via ons data-accessobject
    _wandelroutesLijst =
                    _wandelrouteDA.ReadTableFilterProvincie(provincieComboBox.Text);

    // de lijst met wandelroutes vernieuwen in de listbox
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Deze methode reageert op de gebeurtenis (Event) `SelectedIndexChanged` van de keuzelijst `provincieComboBox`.

Om deze methode in te voegen, selecteerden we eerst in de weergave Design van het formulier het keuzelijstje om dan in het venster Properties bij de Events (⚡) op zoek te gaan naar de gebeurtenis `SelectedIndexChanged`. In het vakje ernaast dubbelklikten we om de overeenkomstige methode toe te voegen aan de formulierklasse.



De methode `SelectedIndexChanged` wordt opgeroepen elke keer als de gebruiker een ander element in de `ComboBox` selecteert.

Bij deze methode moeten we eigenlijk maar twee acties uitvoeren:

❖ Met het data-accessobject `_wandelrouteDA` starten we onze databankactie op:

```
_wandelroutesLijst =
                _wandelrouteDA.ReadTableFilterProvincie(provincieComboBox.Text);
```

Met de methode `ReadTableFilterProvincie()` lezen we een lijst van wandelroutes in.

Bij deze methode moeten we als parameter de provincie doorgeven waarvan we de wandelroutes willen selecteren. Dit is meer bepaald de provincie die de gebruiker in de keuzelijst `provincieComboBox` aangeduid heeft.

En hoe vragen we aan de keuzelijst op wat er geselecteerd staat? Uit de code hierboven blijkt dat dit gebeurt met de expressie "`provincieComboBox.Text`". We gebruiken dus dezelfde Propery `Text`, net zoals bij een tekstvak, om de inhoud ervan aan te duiden.

Opmerking:

- We moeten toch eventjes melden dat keuzelijsten in feite *complexer* zijn dan we hier laten blijken. Ook met de methode `SelectedItem` (cfr. de `ListBox`) kan je het geselecteerde item achterhalen.

  Omdat we ons in deze cursus echter beperken tot *eenvoudige* keuzelijsten (de items in de keuzelijst tikten we in als letterlijke waarden), hebben we aan de Property `Text` voldoende.

Onthoud 8.8 Stel je hebt een keuzelijst met de naam `vakComboBox`. De items die in deze keuzelijst voorkomen, tikte je vooraf in het Properties venster in bij het invulvenstertje achter de Property `Items`.

Met de expressie **`vakComboBox.Text`** verwijs je dan naar het **geselecteerde item** van deze keuzelijst. Dit levert je een resultaat op van het type `String`.

❖ Nu we de wandelroutes van de geselecteerde provincie in de lijst `_wandelroutesList` gestopt hebben, kunnen we de laatste actie uitvoeren. We laten namelijk deze lijst weergeven (vernieuwen) in de `ListBox` op het formulier:

```
wandelroutesListBox.DataSource = null;
wandelroutesListBox.DataSource = _wandelroutesLijst;
```

De clou van dit voorbeeldje is dus dat - elke keer als de gebruiker in de keuzelijst een andere provincie kiest - we het data-accessobject `_wandelrouteDA` aan het werk zetten om de lijst met wandelroutes bij die provincie op te halen uit de databank, om dan de `ListBox` te laten vernieuwen met deze lijst.

8.4.7 Eén waarde uit de databank lezen

Bij 'Onthoud 8.5' (zie pagina 258) en bij 'Onthoud 8.7' (zie pagina 267) hadden we het over de verschillende methoden om een `MySqlCommand` op te starten/uit te voeren. We hadden tot nu volgende opdeling:

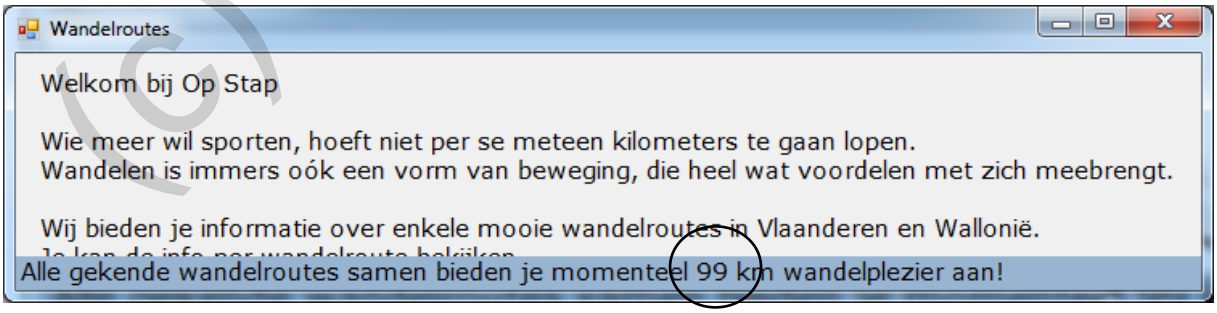
1 Als het SQL-statement een lees-actie (SELECT) betreft waarmee we meerdere gegevens (meerdere rijen en/of kolommen) inlezen, dan lieten we het `MySqlCommand` opstarten met de methode `ExecuteReader()`.

   De ingelezen gegevens kwamen dan in een `MySqlDataReader` terecht.

2 Hadden we te doen met een SQL-statement dat gegevens op één of andere manier wijzigt in de databank (UPDATE, INSERT of DELETE), dan startten we het `MySqlCommand` op met de methode `ExecuteNonQuery()`.

We willen hier een **derde variant** (de methode `ExecuteScalar()`) introduceren waarmee we ook een leesactie opstarten (dus **SELECT**), die deze keer echter maar in **één enkele waarde** resulteert.

We werken in het formulier `WandelroutesWelkomForm` een toepasselijk voorbeeldje uit.



Onderaan in dat formulier tonen we het **totaal aantal kilometer** aan wandelroutes dat we in onze toepassing verzameld hebben. Dit totaal zullen we uiteraard uit onze databank ophalen, zodat we na het toevoegen, verwijderen of wijzigen van wandelroutes hier automatisch een geactualiseerd aantal kilometer te zien krijgen.

Opmerking:

- Om straks dit formulier te kunnen testen, moet je van `WandelroutesWelkomForm` vlug nog even in **program.cs** het startformulier maken:

```
Application.Run(new WandelroutesFilterForm());
```

In de klasse `WandelroutesDA`

Omdat we een nieuwe vraag hebben die we aan onze databank moeten delegeren (namelijk geef eens de totale som van alle afstanden van de wandelroutes in `tblWandelroutes`), zullen we onze data-accessklasse `WandelrouteDA` uitbreiden met een extra methode.

Deze nieuwe methode `ReadAantalKm()` ziet er als volgt uit:

```
public int ReadAantalKm()
{
    int totaal;

    // SQL-statement om alle afstanden in tblwandelroutes op te tellen
    String sql = "SELECT SUM(kilometers) FROM tblwandelroutes;";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // met ExecuteScalar laat je een leescommando opstarten
    // er wordt slechts één waarde ingelezen
    // opgehaalde resultaat nog converteren naar het type int
    totaal = Convert.ToInt32(mySqlCommand.ExecuteScalar());

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();

    // totaal aantal km teruggeven
    return totaal;
}
```

Merk op dat deze methode `ReadAantalKm()` gewoon een `int` retourneert. Het totaal aantal kilometer kunnen we inderdaad in één getalletje vatten. Deze methode hoeft ons deze keer geen lijst van `Wandelroute`-objecten op te leveren (zoals bij onze eerdere Read-methoden).

We hebben een SQL-statement nodig dat de som maakt van alle afstanden in de kolom `kilometers` bij de tabel `tblwandelroutes`. Een som laten berekenen over meerdere records, daarvoor gebruik je de statistische functie SUM() in je SQL-statement.

Had je onderstaand SQL-statement zelf nog kunnen opstellen?

```
SELECT      SUM(kilometers)
FROM        tblWandelroutes;
```

In onze code vertaalt zich dat in volgende instructie:

```
String sql = "SELECT SUM(kilometers) FROM tblwandelroutes;";
```

Zoals steeds laten we het SQL-statement in tekstformaat (de variabele `sql`) omzetten naar een echt `MySqlCommand`:

```
MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

Het is - na het openen van de connectie - dan nog zaak om ons `MySqlCommand` effectief op te starten (uit te voeren):

```
totaal = Convert.ToInt32(mySqlCommand.ExecuteScalar());
```

De nieuwigheid is hier de methode `ExecuteScalar()`!

Vroeger vertelden we dat je met de methode `ExecuteReader()` een `MySqlCommand` kon opstarten, tenminste als het bijhorend SQL-statement een lees-operatie (**SELECT**) bevatte. Omdat een leesactie in een databank **meerdere rijen en/of kolommen** kan opleveren, wordt het resultaat van de methode `ExecuteReader()` opgeslagen in een `MySqlDataReader`-object.

Een `MySqlDataReader`-object is trouwens een *zwaar* object, dat heel wat geheugen-bronnen in beslag neemt en dat - zoals jullie al eerder ondervonden hebben - moeilijk te manipuleren valt.

Omdat je niet altijd een kanon nodig hebt om een vliegje uit de lucht te schieten, heeft men bij de klasse `MySqlCommand` nog een alternatieve manier voorzien om een lees-operatie (SELECT) op te starten. Het gaat over onze bewuste methode `ExecuteScalar()`.

Het grote verschil is dat de methode `ExecuteScalar()` maar **één waarde** zal inlezen! Dit is in regel de waarde van de eerste kolom in de eerste rij van het resultaat dat je SELECT-statement opleverde.

Omdat de methode `ExecuteScalar()` maar één waarde retourneert, is het ook niet meer nodig om deze waarde in de vorm van een onhandige `MySqlDataReader` op te slaan.

Het SQL-statement in ons voorbeeldje berekent het totaal aantal kilometer, of heeft als uitkomst dus ook maar één enkel getal. Vandaar dat we hier probleemloos met de methode `ExecuteScalar()` kunnen werken (en we de grote broer `ExecuteReader()` achterwege laten).

Het retourtype van de methode `ExecuteScalar()` is heel algemeen, namelijk van het type `Object`.

Dit is wel logisch, want toen de programmeurs van MySQL de methode `ExecuteScalar()` ter beschikking stelden, konden ze nooit van tevoren weten of we nu met die methode een getal (bv. totaal km), dan wel een `String` (bv. de naam van de langste wandelroute), dan wel een `double` (de gemiddelde afstand van de wandelroutes), ... zouden ophalen.

Er zal zich dus steeds nog een conversie opdringen!

Onthoud 8.9 Als het SQL-statement van een `MySqlCommand`-object een **lees-operatie** (een **SELECT**) is, die **één waarde** oplevert, dan kan je dit `MySqlCommand` laten uitvoeren met de methode `ExecuteScalar()`.

De (enige) ingelezen waarde die je bekomt met de methode `ExecuteScalar()` zal van het type `Object` zijn. Een conversie dringt zich dus altijd op.

Mocht je lees-operatie toch meerdere rijen en/of kolommen opleveren, zal enkel het eerste veld (=kolom) van de eerste record (=rij) van het opgehaalde resultaat uit de databank geretourneerd worden.

Wij willen de ingelezen waarde (het totaal aantal kilometers) opslaan in een variabele `int totaal`. Er is dus een conversie naar het type `int` nodig:

```
totaal = Convert.ToInt32(mySqlCommand.ExecuteScalar());
```

De korte conversienotatie met de ronde haken "`(int)`" weigerde hier dienst omdat we niet tussen twee numerieke typen aan het converteren zijn. We moesten onze cast dus wel met de iets langere `Convert.ToInt32()`-notatie regelen.

De methode `ReadAantalKm()` sluiten we af met volgende instructie:

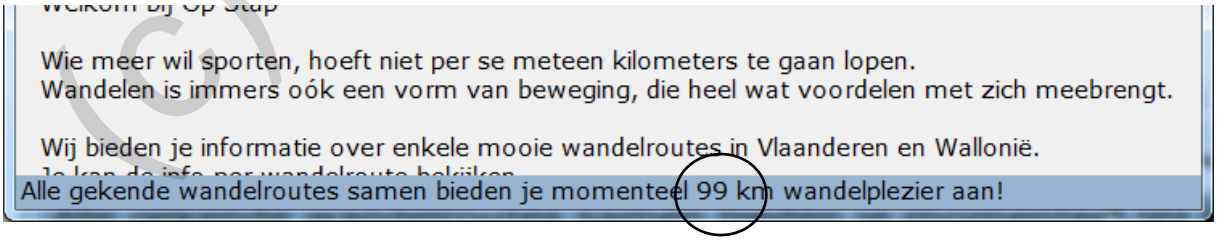
```
return totaal;
```

Onze methode moet immers het totaal aantal kilometer, dat we met succes aan onze databank bevraagd hebben, laten retourneren!

De methode `ReadAantalKm()` toepassen

We hebben onze data-accessklasse `WandelrouteDA` laten uitbreiden met de methode `ReadAantalKm()`. Deze methode berekent en retourneert het totaal aantal kilometer van alle wandelroutes in onze databank.

In het formulier `WandelroutesWelkomForm` willen we dit totaal als volgt presenteren:



Het blauwe labeltje onderaan waarin dit totaal moet komen, heet trouwens `infoLabel`.

In `WandelroutesWelkomForm` nog eens controleren of je volgende usings hebt:

```
using Business;
using DataAccess;
```

Met onderstaande code kruiden we het formulier af:

```
public partial class WandelroutesWelkomForm : Form
{
    private int _aantalKmWandelroutes;
    private WandelrouteDA _wandelrouteDA;

    public WandelroutesWelkomForm()
    {
        InitializeComponent();

        // het veld _wandelrouteDA initialiseren
        _wandelrouteDA = new WandelrouteDA();

        // het veld _aantalKmWandelroutes initialiseren via ons data-accessobject
        _aantalKmWandelroutes = _wandelrouteDA.ReadAantalKm();

        // het aantal km wandelroutes weergeven in het infolabel
        infoLabel.Text = "Alle gekende wandelroutes samen bieden je momenteel " +
                  _aantalKmWandelroutes.ToString() + " km wandelplezier aan!";
    }
}
```

We voorzien een veld `int _aantalKmWandelroutes` om straks het totaal aantal kilometer bij te houden.

We declareren en initialiseren het data-accessobject `_wandelrouteDA`.

Via de methode `ReadAantalKm()` kan `_wandelrouteDA` het totaal aantal kilometer van alle wandelroutes in `tblwandelroutes` aan de databank bevragen.

```
_aantalKmsWandelroutes = _wandelrouteDA.ReadAantalKm();
```

Dit gevonden totaal verwerken we in het label onderaan het formulier.

```
infoLabel.Text = "Alle gekende wandelroutes samen bieden je momenteel " +
                 _aantalKmWandelroutes.ToString() + " km wandelplezier aan!";
```

Omdat we dit alles in de constructor van het formulier `WandelroutesWelkomForm` regelden, krijgen we bij het openen van het formulier onmiddellijk onderaan de juiste tekst te zien!

Opmerking:

- Telkens we in dit hoofdstuk naar een nieuw formulier overschakelden om een ander databankgerelateerde voorbeeldje uit te werken, lieten we jullie eerst in **program.cs** dit formulier instellen als startformulier in het project.

  Aan het project `Presentation` hebben wij echter ook nog het formulier `StartOp-StapForm` toegevoegd. Kan je die nu even (via **program.cs**) als startformulier nemen:

  ```
  Application.Run(new StartOpStapForm());
  ```

Je krijgt nu bij het starten van de toepassing volgend venster te zien:

Met de drie knopjes kan je respectievelijk `WandelroutesForm` (om wandelroutes te bekijken, te wijzigen, toe te voegen en te verwijderen), `WandelroutesFilterForm` (om wandelroutes te filteren per provincie) en `WandelroutesWelkomForm` (met het totaal aantal kilometer) laten openen.

Of had je liever dat we dit al vroeger getoond hadden?

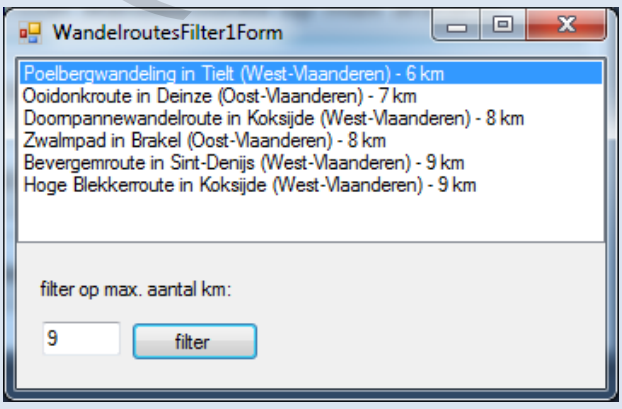
8.5 Even oefenen

Oefening 8.2 OpStapOefening - Bij de brongegevens van de oefeningen in hoofdstuk 8 hebben we een solution `OpStapOefening` geplaatst.

Naast de gekende formulieren met het overzicht van alle wandelroutes (`WandelroutesForm`) en het info-formulier waarmee we een afzonderlijke wandelroute kunnen opvragen en editen (`WandelrouteInfoForm`), hebben we extra formulieren toegevoegd die als basis zullen dienen voor enkele nieuwe programmeeropdrachten.

Het formulier `WandelroutesFilter1Form`

In het formulier `WandelroutesFilter1Form` (je opent dit formulier met de knop 'Oef 1: Filter wandelroutes op max afstand' op het startformulier), laten we de gebruiker in het tekstvak onderaan een maximum afstand intikken. Het knopje 'filter' zou alle routes die precies diezelfde afstand hebben of die korter zijn, moeten weergeven in de `ListBox`. Graag laten we de wandelroutes hierbij op hun afstand sorteren.



Bij het knopje 'filter' hebben wij alle nodige code reeds geschreven:

```
private void filterButton_Click(object sender, EventArgs e)
{
    // waarde uit tekstvakken ophalen
    int max = Convert.ToInt32(maxKmTextBox.Text);

    // de lijst op basis van filter opvullen via ons data-accessobject
    _wandelroutesLijst = _wandelrouteDA.ReadTableFilterMaxAfstand(max);

    // de lijst met wandelroutes vernieuwen in de listbox
    wandelroutesListBox.DataSource = null;
    wandelroutesListBox.DataSource = _wandelroutesLijst;
}
```

Je leest in deze code dat we de (gefilterde) wandelroutes ophalen via de methode `ReadTableFilterMaxAfstand()` van ons **data-accessobject** `_wandelrouteDA`. Bij deze methode geven we als parameter de waarde mee die de gebruiker intikte in het tekstvak `maxKmTextBox`. Dit is dus de maximum afstand waarvoor we wandelroutes willen selecteren. De verkregen lijst met wandelroutes schrijven we tenslotte weg naar de `ListBox` op het formulier.

Ook de methode `ReadTableFilterMaxAfstand()` (zie de klasse `WandelrouteDA`) hebben wij al voor 95% geprogrammeerd. Aan jullie om de laatste, belangrijke, stukjes code te vervolledigen.

Merk (nog eens) op dat deze methode inderdaad een parameter neemt:

```
public List<Wandelroute> ReadTableFilterMaxAfstand(int afstand) {}
```

De parameter `int afstand` stelt hier dus ons maximaal aantal kilometer voor, waarvoor we de wandelroutes willen selecteren!

To do, telkens onder een "`// aanvullen`"-commentaarregel:

1. Vul de instructie aan waar je het **SQL-statement** opbouwt in de variabele `sql`. Dit SQL-statement selecteert (gesorteerd op de afstand) alle wandelroutes waarvoor de afstand (kolom `kilometers`) kleiner of gelijk is aan een bepaalde waarde (*maar welke waarde dat is, dat weten we nu nog niet ...*).

   Vul hieronder de bewuste instructie in:

   ```
   String sql =
   ```

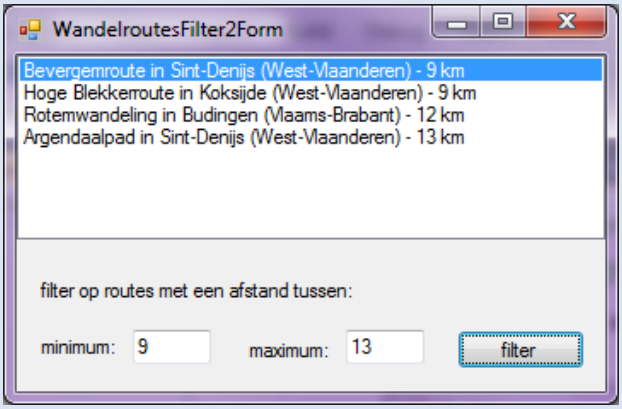
2. Waarschijnlijk heb je (een) **@parameter**(s) toegevoegd aan je SQL-statement.

   Het tweede stukje code dat je in de methode `ReadTableFilterMaxAfstand()` moet aanvullen, is het onderdeel waar je de @parameter(s) laat vervangen door hun eigenlijke waarde.

Kon je de twee bovenstaande opdrachten foutloos uitvoeren, dan zou dit filterformulier moeten werken. Test maar even uit ...

Het formulier `WandelroutesFilter2Form`

In een tweede filterformulier `WandelroutesFilter2Form` (je opent dit formulier via de knop 'Oef 2: Filter wandelroutes op range afstand' op het startformulier), moet de gebruiker zowel een minimum- als een maximumafstand opgeven. Bedoeling is om in de `ListBox` enkel de wandelroutes te tonen die binnen die grenzen vallen (inclusief min en max):



Ook bij deze opdracht hebben wij al voor de presentatiecode achter de filter-button gezorgd. Om de gewenste wandelroutes op te halen doen we er als volgt een beroep op het **data-accessobject** `_wandelrouteDA`:

```
// de lijst op basis van filter opvullen via ons data-accessobject
_wandelroutesLijst = _wandelrouteDA.ReadTableFilterRangeAfstand(min,max);.
```

Deze keer is het dus jullie taak om in de klasse `WandelrouteDA` één en ander aan te vullen in deze methode `ReadTableFilterRangeAfstand()`.

Houd er zeker rekening mee dat deze methode twee parameters neemt:

```
public List<Wandelroute> ReadTableFilterAfstandrange(int minAfstand, int
maxAfstand) {}
```

To do, telkens onder een "`// aanvullen`"-commentaarregel:

1. Vul de instructie aan waar je het betreffende **SQL-statement** opbouwt in de variabele `sql`. Het verschil met de 1e oefening is dat je nu zowel moet filteren op een minimum- als op een maximumafstand.

   Noteer ook hier het SQL-statement dat je hiervoor liet opstellen:

   ```
   String sql =
   ```

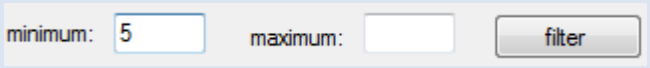
2. De **@parameters** uit je SQL-statement, zal je hun concrete waarde moeten geven.

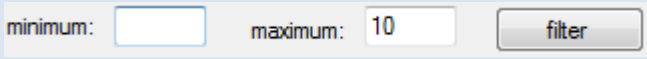
Test opnieuw of `WandelroutesFilter2Form` de juiste wandelroutes laat filteren.

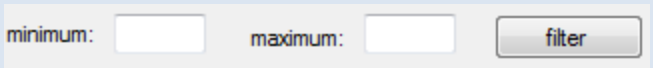
(⌛) Uitbreiding

Als je de 'filter'-knop aanklikt, terwijl je het tekstvak met het minimum en/of het maximum leeg liet, dan crasht het formulier. Zo'n fout zouden we moeten kunnen voorkomen!

We proberen dit hier op een *slimme*, intuïtieve manier op te lossen. Drie voorbeeldjes om duidelijk te maken waar we naartoe willen:

- 

Als je enkel als minimum '5 km' opgeeft (en het maximum leeg laat), zou het niet vreemd zijn om met de 'filter'-knop alle wandelroutes te tonen van 5 of meer kilometer.
- 

Als je enkel als maximum '10 km' opgeeft (en het minimum leeg laat), mag je verwachten dat de 'filter'-knop alle wandelroutes oplevert van 10 of minder kilometer.
- 

Als je geen minimum en geen maximum ingeeft, zou het meest logische zijn om met de 'filter'-knop alle wandelroutes op te halen (er wordt dan in principe dus geen filter toegepast).

Lukt het je om bovenstaande bijkomende functionaliteit achter het 'filter'-knopje te steken?

Als je het slim aanpakt, hoef je enkel in de **Presentation Layer** (of dus bij het formulier) bijkomende code te schrijven, waar je o.a. controleert welke tekstvakken leeg bleven. Wij slaagden erin om de Data Access Layer (of de klasse `WandelrouteDA`) in onze modeloplossing ongewijzigd te laten en hebben voldoende aan de methode `ReadTableFilterAfstandrange()`.

Het formulier `VerwijderWandelroutesGemeenteForm`

We willen toch een beetje geld verdienen aan onze wandelroutes-toepassing. Daar wij met ons programma reclame maken voor de verschillende wandelroutes, mogen we als tegenprestatie wel een kleine donatie vragen aan de betreffende gemeenten.

Als een gemeente weigert bij te dragen, even goede vrienden, maar hun wandelroutes verdwijnen sito presto uit onze lijsten.

Met het formulier `VerwijderWandelroutesGemeenteForm` willen we het mogelijk maken om in één beweging (alle) wandelroutes van een gegeven gemeente te wissen. De gemeente tik je hiervoor in het tekstvak in:

Het knopje verwijderen kreeg van ons al volgende code:

```
private void verwijderButton_Click(object sender, EventArgs e)
{
    // opvragen van welke gemeente je de wandelroutes wil wissen
    String gemeente = gemeenteTextBox.Text;

    // wandelroutes laten wissen via het data-accessobject
    _wandelrouteDA.DeleteRecordsGemeente(gemeente);
}
```

Je begrijpt dat de methode `DeleteRecordsGemeente()` van het **data-accessobject** `_wandelrouteDA` hier de verantwoordelijkheid krijgt om de databankactie uit te voeren waarmee we de wandelroutes verwijderen.

Ook de methode `DeleteRecordsGemeente()` is reeds voor een groot gedeelte voorgekauwd (zie de klasse `WandelrouteDA`). Jullie zorgen voor de finishing touch door de laatste stukjes code te vervolledigen.

Nog even meegeven dat deze methode de gemeente waarvoor we de wandelroutes willen droppen, als `String`-parameter meekrijgt:

```
public void DeleteRecordsGemeente(String gemeente){}
```

To do, telkens onder een "`// aanvullen`"-commentaarregel:

1. Vul de instructie aan waar je het **SQL-statement** opbouwt in de variabele `sql`. Dit SQL-statement zou alle wandelroutes die in een gegeven gemeente gelegen zijn (*maar welke gemeente dat precies is, is nu nog onbekend*), moeten wissen.

   Vul hieronder de bewuste instructie in:

   ```
   String sql =
   ```

2. De **@parameter**(s) die je ongetwijfeld in het SQL-statement verwerkt hebt, zal je moeten laten substitueren door een 'echte' waarde.

3. Je moet deze keer ook zelf nog de instructie schrijven om het `MySqlCommand` (we noemden dit object `mySqlCommand`) op te starten. Pas dan wordt ons SQL-statement effectief uitgevoerd (of worden de wandelroutes *echt* gewist uit de bewuste tabel in de databank).

Vooraleer je test, zal je eerst eens goed moeten kijken welke gemeenten er wandelroutes hebben. Eventueel voeg je voor een bepaalde gemeente nog enkele (test)wandelroutes toe. Dit alles kan geregeld worden via de formulieren die je opent via het eerste knopje op het startformulier.

Nadat je in `VerwijderWandelroutesGemeenteForm` de wandelroutes van een bewuste gemeente liet wissen, mag je deze wandelroutes niet meer te zien krijgen in de `ListBox` als je het formulier met alle wandelroutes (eerste knop op startformulier) opnieuw opent.

Opmerking:

- Als je ons SQL-statement om de wandelroutes te wissen (per gemeente) *rechtstreeks in MySQL Workbench* test, is het mogelijk dat dit als een onveilige instructie gezien wordt (en daardoor geblokkeerd wordt). Een instructie waarmee je meerdere rijen ineens wist kan potentieel immers grote schade aanbrengen aan je databank.

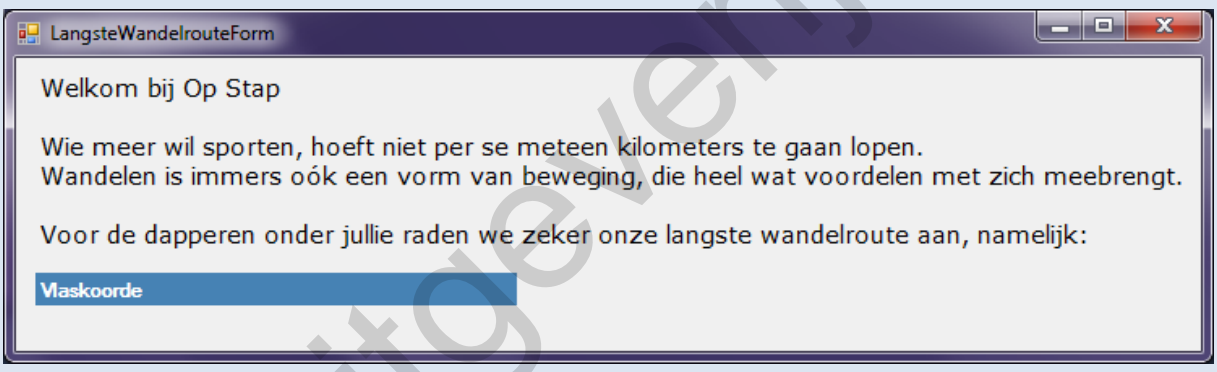
  Met volgende instelling in MySQL Workbench kan je aangeven dat we vertrouwd mogen worden als we een dergelijk SQL-statement uitvoeren:

  Edit | Preferences | SQL Editor | "Safe updates (rejects UPDATES …)" uitvinken.

  Afhankelijk van de versie van MySQL Workbench wordt een iets andere formulering voor deze instelling gebruikt.

Het formulier `LangsteWandelrouteForm`

De laatste oefening in deze reeks werken we uit in het formulier `LangsteWandelroute-Form`. In het blauwe labeltje in `LangsteWandelrouteForm`, willen we **de naam** van **de langste wandelroute** laten zetten:



Opmerking:

- Het zou nuttiger zijn om in het blauwe kadertje ook de afstand en de gemeente van die langste wandelroute te printen, maar - in het kader van deze oefening - wilden we ons beperken tot *één enkel gegeven* (namelijk de naam van de wandelroute).

De presentatielaagcode (de code in het formulier) is opnieuw al volledig af:

```
public LangsteWandelrouteForm()
{
    InitializeComponent();

    // het veld _wandelrouteDA initialiseren
    _wandelrouteDA = new WandelrouteDA();

    // het veld _naamLangsteWandelroute instellen via ons data-accessobject
    _naamLangsteWandelroute = _wandelrouteDA.ReadLangsteWandelroute();

    // de naam van de langste wandelroutes weergeven in het label
    langsteRouteLabel.Text = _naamLangsteWandelroute ;
}
```

Jullie begrijpen dat het **data-accessobject** `_wandelrouteDA` hier via de methode `ReadLangsteWandelroute()` de langste wandelroute zal opvragen. Bemerk dat deze leesactie ons enkel **één `String`** zal opleveren (dus geen lijst van `Wandelroute`-objecten)!

Je ziet ook dat we de gevonden wandelroute laten wegschrijven naar het (blauwe) labeltje `langsteRouteLabel` op het formulier.

In de Data Access Layer (zie klasse `WandelrouteDA`) ontbreken bij de methode `ReadLangsteWandelroute()` echter nog enkele noodzakelijke instructies.

To do, telkens onder een "`// aanvullen`"-commentaarregel:

1. Jullie moeten het **SQL-statement** toevoegen om **de naam** van de langste wandelroute te selecteren! Chapeau als je - zonder onze tip(🔔) - het SELECT statement vindt dat als resultaat **één celletje** (één waarde) oplevert met deze wandelroutenaam.

   Om dit SQL-statement niet meer te vergeten, maken we hieronder een beetje plaats om je code neer te pennen:

   ```
   String sql =
   ```

2. De leesactie wordt pas uitgevoerd als we ons commando `mySqlCommand` laten opstarten. Wij weten ondertussen dat dit `MySqlCommand` (door ons specifieke SQL-statement) maar **één waarde** zal opleveren.

   Met een instructie laat je `mySqlCommand` opstarten en zorg je dat de opgehaalde uitkomst weggeschreven wordt in de `String`-variabele `langste`. Er zal hierbij een cast tussen gegevenstypen nodig zijn!

   ```
   langste =
   ```

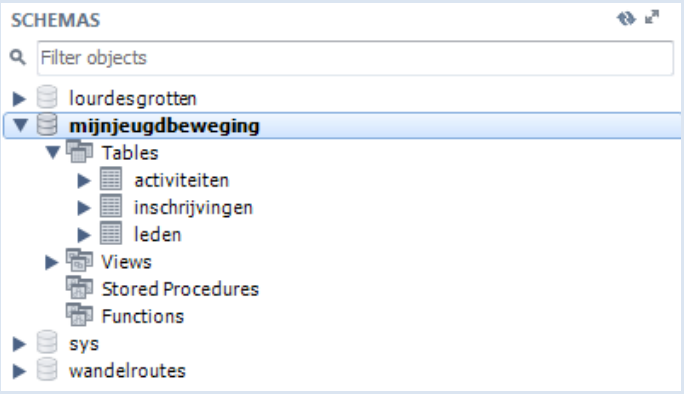
Deze keer hebben we niet gesproken over de stap waar we de **@parameters** hun eigenlijke waarde geven, simpelweg omdat ons SQL-statement geen @parameters behoeft.

Tip:

🔔 Om jullie te helpen om het juiste SQL-statement op te stellen, laten we nonchalant even het sleutelwoord **LIMIT** vallen.

Oefening 8.3 Jeugdbeweging - Bij de vorige oefeningenreeks stonden de bronprojecten al boordevol met programmacode en moesten jullie *slechts* enkele afzonderlijke instructies in het hart van de Data Access Layer aanvullen. Bij deze oefening willen we eens *all the way* gaan en bouwen jullie in een nieuwe applicatie alvast de Data Access Layer helemaal, *from scratch*, op.

We gebruiken voor de verandering eens een nieuwe databank. In MySQL Workbench zou je het bestand **Dump_mijnjeugdbeweging.sql** (zie **_hoofdstuk_8/_oefeningen/_databanken**) moeten importeren. Hierna heb je volgende databank `mijnJeugdbeweging` beschikbaar:



De databank bevat drie tabellen:

- tabel `activiteiten`: activiteiten die de jeugdbeweging organiseert;
- tabel `inschrijvingen`: welke leden schrijven in voor welke activiteiten;
- tabel `leden`: de leden van de jeugdbeweging.

Wij zullen ons enkel focussen op de tabel met de activiteiten:

activiteitID	beschrijving	datumtijd
1	Kerstfeestje	2017-12-25 00:00:00
2	Halloweentocht	2017-10-31 00:00:00
3	Dropping	2017-04-10 00:00:00
4	Nachtspel	2017-07-11 00:00:00
5	Vakantiekamp	2017-08-01 00:00:00
6	Ruiltocht	2017-01-12 00:00:00

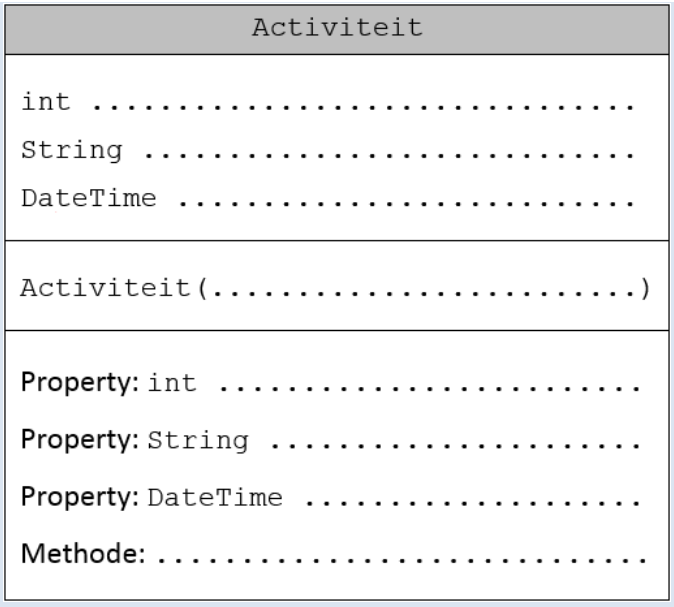
We gaan twee formulieren aanpakken:

- formulier `ActiviteitenForm`: waar we alle activiteiten (gesorteerd) kunnen bekijken.
- formulier `ActiviteitToevoegenForm`: waarmee we nieuwe activiteiten kunnen toevoegen.

Deze formulieren (of de **Presentation Layer**) vind je in de solution terug in het project `Presentation`.

De **Business Layer** hebben we op ons genomen. We hebben daarvoor in het project `Business` de klasse `Activiteit` uitgewerkt! Deze klasse hebben we helemaal gemodelleerd naar de tabel `activiteiten` in de MySQL-databank `mijnjeugdbeweging`. Zo hebben we voor elke kolom in de tabel `activiteiten` een veld en een Property voorzien in de klasse `Activiteit`.

Omdat we jullie aandacht toch graag even naar deze klasse `Activiteit` willen richten, de opdracht om in onderstaand klassendiagram alle velden, de constructor, alle properties en methoden aan te vullen.



Opmerking

- Met het gegevenstype `DateTime` kunnen we in C# een datum (met eventueel een tijdstip) voorstellen. In Java (of de cursus BlueJ) bestaat er geen enkelvoudig gegevenstype om data (en tijdstippen) op te slaan.

Voorbereiding

Een project voor de Data Access Layer

Omdat we jullie bij deze oefening de **Data Access Layer** helemaal zelf willen laten uitwerken, is er in de Solution `Jeugdbeweging` nog geen halve regel data-accesscode terug te vinden.

Alle code in verband met de databank zullen we onderbrengen in een apart project.

We geven het startschot door jullie aan de solution een nieuw project `DataAccess` te laten toevoegen. Dit is een project van het type **Class Library (.NET Framework)**.

References leggen

Het is geen een slecht idee om zo snel mogelijk de nodige referenties (**references**) in de toepassing te leggen. Dat is dan al een zorg minder.

1. Voeg in het nieuwe project `DataAccess` een reference toe naar de `MySql.Data` Assembly, om zo de nodige klassen beschikbaar te maken om een MySQL-databank te kunnen benaderen. De referentie die wij hiervoor nodig hebben, vind je in de lijst bij **Assemblies** | **Extensions**, waar je het item `MySql.Data` aanvinkt. Bevestig met de knop 'OK'.

2. Het project `DataAccess` zal de klasse `Activiteit` (uit het project `Business`) moeten kennen. Leg daarom in het project `DataAccess` een **reference** naar het project `Business`.

3. In onze formulieren (Presentation Layer) zullen we via data-accessobjecten de tabel `activiteiten` in de databank aanspreken. Daarom moeten we in het project `Presentation` nog een **reference** toevoegen naar het project `DataAccess`.

De Server Explorer

Om rechtstreeks vanuit Visual Studio onze MySQL-databank `mijnjeugdbeweging` te kunnen inspecteren (bijvoorbeeld nagaan welke kolommen in de tabel `activiteiten` staan), gaan we in de **Server Explorer** via de knop 'Connect to Database'  een verbinding leggen met deze MySQL-databank `mijnjeugdbeweging`.

De klasse `ActiviteitDA`

Alle code waarmee we vanuit onze toepassing de tabel `activiteiten` benaderen, verzamelen we in één klasse. De logische naam voor deze klasse is `ActiviteitDA`.

1. Voeg de klasse `ActiviteitDA` toe aan het project `DataAccess`. Zorg dat deze klasse publiek is (waarschijnlijk moet je in de header zelf nog het sleutelwoord **public** typen).

2. Neem bovenaan in deze nieuwe klasse **using** statements op naar de namespaces `MySql.Data.MySqlClient` en `Business`. Zo kunnen we er gemakkelijker gebruik maken van de klassen uit deze namespaces.

3. In de klasse `ActiviteitDA` declareer je 2 velden:

   Een veld `_connString` om de informatie over de correcte naam, plaats en authentificatiegegevens van de MySQL-databank te kunnen opslaan. Dit veld is van het type `String` omdat dit nog informatie is in de vorm van tekst.

   Een veld `_mySqlConnection` waarin we de fysieke connectie naar deze databank zullen bijhouden. Dit veld is een object van de klasse `MySqlConnection`.

4. In de constructor van de klasse zullen we beide velden initialiseren.

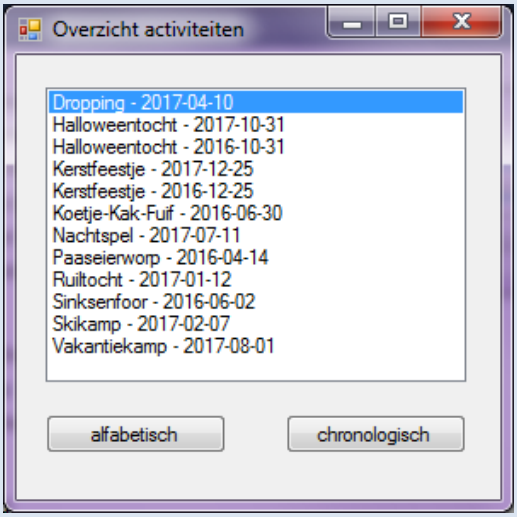
   Het veld `_connString` stel je in op de connectiestring van onze databank `mijnjeugdbeweging`. De credentials hierbij zijn:

   -> server: localhost;
   -> user id: root;
   -> password: Leerling123;
   -> database: mijnjeugdbeweging;

   Voor het initialiseren van het veld `_mySqlConnection` gebruik je de constructor van die klasse. Deze constructor vraagt een connectiestring als parameter.

Activiteiten weergeven in `ActiviteitenForm`

In het formulier `ActiviteitenForm` willen we in een `ListBox` alle activiteiten van de jeugdbeweging alfabetisch weergeven:



Databankgegevens inlezen met de methode `List<Activiteit> ReadTable()`

Aan de data-accessklasse `ActiviteitDA` voeg je een methode `ReadTable()` toe, waarmee we de activiteiten uit de tabel `activiteiten` zullen inlezen. Deze methode moet daarom een **lijst** (`List`) retourneren van `Activiteit`-objecten! Vandaar dus het retour-type `List<Activiteit>`.

In deze methode voer je volgende acties uit:

1. Aangezien we een lijst van activiteiten moeten retourneren, zullen we in de methode zo'n lijst moeten declareren en initialiseren.
2. We declareren een hulpvariabele `sql` van het type `String` waarin we het passende **SQL-statement** bewaren om alle activiteiten te lezen. Zorg ervoor dat jullie SQL-statement de activiteiten alfabetisch laat ordenen volgens de beschrijving.
3. Om het SQL-statement om te toveren van een tekst (type `String`) naar een echt commando, heb je een object nodig van de klasse `MySqlCommand`. We declareren dus zo'n object (en laten het onmiddellijk initialiseren).

   Bij het initialiseren van een `MySqlCommand`-object moet je trouwens twee parameters aanleveren: De eerste is het SQL-statement in tekstformaat (zie hulpvariabele) en de tweede is de connectie met de databank waarop je dit SQL-statement wilt toepassen.
4. We kunnen een `MySqlCommand` enkel uitvoeren als we de connectie naar de databank geopend hebben.

5. Ons `MySqlCommand` bevat een lees-operatie (SELECT) die meerdere waarden teruggeeft als resultaat. Dit betekent dat je dit commando moet uitvoeren met de methode `ExecuteReader()`. De velden/records die deze methode retourneert zullen we tijdelijk opslaan in het object `mySqlDataReader` van de klasse `MySqlDataReader`.

6. De volgende stap is om de ingelezen records uit `mySqlDataReader` in onze lijst met activiteiten te krijgen. Hiervoor zullen we met een lusje alle records uit `mySqlDataReader` overlopen, om deze zo één voor één te kunnen behandelen. Je gebruikt hiervoor een while-lus, die met een voorwaarde controleert of er nog een record van `mySqlDataReader` te lezen valt (`mySqlDataReader.Read()==true`).

   In de lus zal je voor elke record in `mySqlDataReader` een nieuw `Activiteit`-object aanmaken. Kijk na welke parameters (in welke volgorde) de constructor van de klasse `Activiteit` neemt .

   Je gebruikt de expressie `object["veld"]` om de waarde van een bepaald veld op te vragen bij de actieve record in `mySqlDataReader` (bijvoorbeeld: `mySqlDataReader["beschrijving"]`). Denk hierbij ook aan de nodige conversies. Casten naar het type `DateTime` lukt hier trouwens ook met de korte ronde haken-notatie!

   Niet vergeten om het nieuw aangemaakte `Activiteit`-object toe te voegen aan de lijst (`List`).

7. Connecties moeten weer gesloten worden.

8. Laat de opgebouwde lijst met activiteiten retourneren.

De methode `ReadTable()` toepassen

In het formulier `ActiviteitenForm` willen we nu nuttig gebruik maken van de methode `ReadTable()`. We moeten immers eerst alle activiteiten ophalen uit de databank om deze in de `ListBox` te kunnen droppen.

In het formulier `ActiviteitenForm` voer je volgende acties uit:

1. De businessgegevens die we in het formulier tonen, worden in de vorm van een lijst met `Activiteit`-objecten bijgehouden. Daarom declareer je in het formulier (als veld) zo'n lijst met `Activiteit`-objecten. Je laat in de constructor deze lijst initialiseren.

2. Om in ons formulier de methoden te kunnen aanspreken die werden gedefinieerd in de data-accessklasse `ActiviteitDA`, zullen we in het formulier zo'n **data-accessobject** nodig hebben. Declareer daarom (als veld) in de formulierklasse een object `_activiteitDA` van deze klasse `ActiviteitDA` en laat dit object in de constructor initialiseren.

   Wordt de klasse `ActiviteitDA` nog in het rood onderlijnd, dan heb je bovenaan in je formuliermodule nog geen using naar de namespace `DataAccess` gezet.

3. Ook nog bij de constructor van ons formulier laten we onze lijst (zie puntje 1) nu opvullen met de activiteiten uit de MySQL-databank. Via het data-accessobject `_activiteitDA` (zie puntje 2) ga je deze activiteiten ophalen met de methode `ReadTable()`.

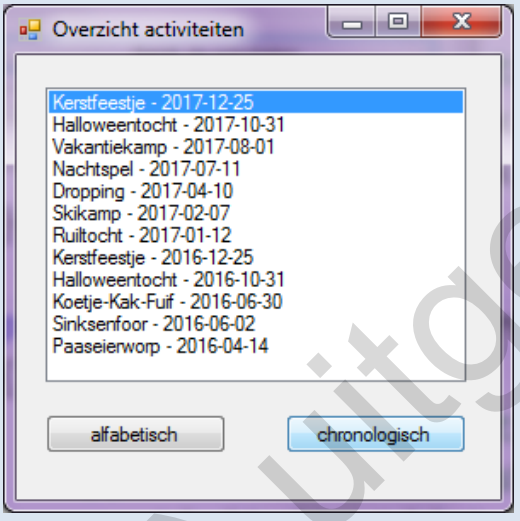
4. Tenslotte stel je de lijst met de activiteiten in als bron voor de `ListBox`. Een lijst toekennen aan een `ListBox` gebeurt via de Property `DataSource` van deze `ListBox`. Ook dit valt te programmeren in de constructor van de formulierklasse.

Sortering in `ActiviteitenForm`

In het formulier `ActiviteitenForm` worden de activiteiten van de jeugdbeweging nu alfabetisch gesorteerd op de beschrijving.

Misschien is het ook wel handig om de activiteiten in *omgekeerde* chronologische volgorde beschikbaar te hebben. Zo zie je veel makkelijker wat de recentste activiteiten in de jeugdbeweging zijn, want deze zullen bovenaan verschijnen.

We gaan de gebruiker zelf de mogelijkheid geven om te kiezen hoe hij/zij de activiteiten wil sorteren. Concreet gaan jullie hiervoor twee knopjes aan het formulier toevoegen:



Databankgegevens chronologisch inlezen met de methode `List <Activiteit> ReadTableSortDatumTijd()`

De volgende stap is om de data-accessklasse `ActiviteitDA` uit te breiden met een nieuwe methode.

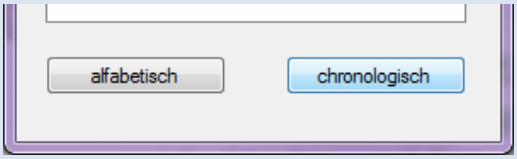
Concreet is jullie opdracht om er een methode `ReadTableSortDatumTijd()` te voorzien, die net als de methode `ReadTable()` een lijst (`List`) met alle `Activiteit`-objecten oplevert. Het enige verschil is dat de objecten in deze lijst anders gesorteerd worden. We willen hier een aflopende sortering op het veld `datumtijd` uit onze tabel `activiteiten`.

Omdat deze methode `ReadTableSortDatumTijd()` bijna een identiek tweelingbroertje is van de methode `ReadTable()`, zouden jullie kopieer/plak-gewijs deze methode snel

geprogrammeerd moeten hebben. Enkel het **SQL-statement** moet een klein beetje aangepast worden.

De methode `ReadTableSortDatumTijd()` toepassen

Plaats op het formulier `ActiviteitenForm` onder de `ListBox` de volgende twee knoppen:



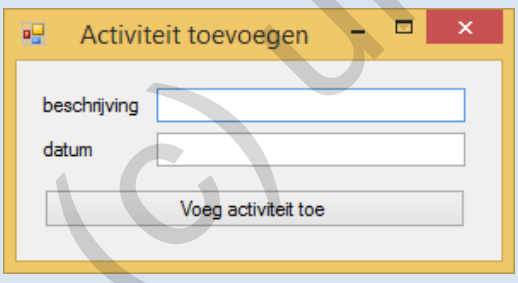
Met een klik op de knop 'chronologisch', worden volgende acties uitgevoerd:

1. Laat de lijst met activiteiten (veld dat je hiervoor in het formulier declareerde) opnieuw opvullen met de activiteiten uit de databank. Omdat we de activiteiten nu *omgekeerd* chronologisch willen sorteren, roep je hiervoor de methode `ReadTableSortDatum-Tijd()` op bij ons data-accessobject `_activiteitDA`.
2. Laat de activiteiten in de `ListBox` vernieuwen door de lijst opnieuw in te stellen als `DataSource` voor de `ListBox`. Niet vergeten dat om een `ListBox` te vernieuwen, je eerst de `DataSource` eens leeg moet maken (instellen op **null**)!

Schrijf ook de code achter de knop 'alfabetisch', waarmee je de activiteiten weer alfabetisch in de `ListBox` ordent.

Een activiteit toevoegen in `ActiviteitToevoegenForm`

In de jeugdbeweging wil men uiteraard ook *nieuwe* activiteiten op de kalender kunnen zetten. Vandaar het formulier `ActiviteitToevoegenForm`:



Een activiteit toevoegen met de methode `void CreateRecord()`

Een record toevoegen aan de tabel `activiteit` is duidelijk een databankactie. De data-accessklasse `ActiviteitDA` zullen we daarom uitbreiden met een nieuwe methode `CreateRecord()` die deze taak zal regelen.

Omdat we aan deze methode zullen moeten vertellen *welke* activiteit we willen toevoegen, definiëren we een parameter van het type `Activiteit`. De methode `CreateRecord()`

leest geen gegevens in uit de tabel `activiteten` en hoeft dus niets te retourneren. Vandaar dat we de methode een `void` als retourtype geven.

Wat extra informatie om deze methode correct te programmeren:

1. We declareren een hulpvariabele `sql` van het type `String` waarin we het passende **SQL-statement** opbouwen om één activiteit toe te voegen aan de tabel `activiteiten`.

   Het veld `activiteitID` is een autonummeringveld. Dit veld mag je dus negeren bij het opstellen van het SQL-statement.

   Uiteraard zullen we de kolommen `beschrijving` en `datumtijd` van de nieuwe record wel een waarde moeten geven. De nieuwe waarde voor deze kolommen stel je voorlopig nog voor via **@parameters**.

2. Om het SQL-statement om te toveren van tekst naar een echt commando, heb je een object nodig van de klasse `MySqlCommand`. We declareren dus zo'n object en laten het onmiddellijk initialiseren.

3. Omdat het SQL-statement (en dus ook het `MySqlCommand`) nog @parameters bevat (er zijn nog *onbekenden*), kan het `MySqlCommand` nog niet uitgevoerd worden!

   Jullie moeten elke @parameter laten vervangen door een actuele waarde.

   Als tip vertellen we dat de methode `CreateRecord()` een parameter van het type `Activiteit` neemt. De methode moet immers weten *welke* activiteit toegevoegd moet worden aan de tabel `activiteiten`. Aan het `Activiteit`-object dat je als parameter binnenkrijgt, kan je dus de eigenlijke beschrijving en datumtijd opvragen.

4. We kunnen een `MySqlCommand` enkele uitvoeren als we de connectie naar de databank openen.

5. Ons `MySqlCommand` bevat geen lees-operatie (SELECT), maar voert wel een actie uit in de databank (INSERT, UPDATE of DELETE). Dit betekent dat je dit commando moet opstarten via de methode `ExecuteNonQuery()`.

6. Niet vergeten om tenslotte de connectie weer te sluiten.

<u>De methode `CreateRecord()` toepassen</u>

Als je in het formulier `ActiviteitToevoegenForm` in de tekstvakken een beschrijving en een datum invoert en daarna op de knop klikt, zou je de betreffende activiteit moeten laten toevoegen aan de tabel `activiteiten`.

Om in het formulier `ActiviteitToevoegenForm` databankacties uit te voeren, hebben we een **data-accessobject** nodig.

Laat in het formulier bijgevolg een veld `_activiteitDA` van de klasse `ActiviteitDA` declareren. In de constructor van het formulier laat je `_activiteitDA` initialiseren als een nieuw `ActiviteitDA`-object.

Wordt de klasse `ActiviteitDA` bij jullie nog in het rood onderlijnd, dan heb je bovenaan in je formuliermodule nog geen using naar de namespace `DataAccess` gezet.

En nu zorgen dat de nieuwe activiteit in de databank terechtkomt als we klikken op het knopje 'Voeg activiteit toe'. Schrijf dus code bij de gebeurtenis `Click` van deze knop.

We onderscheiden volgende drie stappen:

1. De activiteit die we willen toevoegen aan de tabel `activiteiten` moeten we verpakken als een `Activiteit`-object. We maken dus een nieuw `Activiteit`-object aan. In de klasse `Activiteit` kan je opzoeken welke parameters (in welke volgorde) de constructor hiervoor nodig heeft.

   Gebruik de beschrijving en de datum die de gebruiker in de tekstvakken ingetikt heeft, om dit nieuwe `Activiteit`-object te initialiseren.

   Bij de parameters van de constructor van de klasse `Activiteit` moet je ook een activiteitID opgeven. Omdat het definitieve id pas toegekend wordt eens de activiteit in de databank terecht komt (`activiteitID` is autonummering), maakt het niet uit welk activiteitID je hier gebruikt. Ons voorstel is om als activiteitID voor de nieuwe activiteit een neutrale 0 te gebruiken.

   Waarschijnlijk bots je nog op een conversieprobleem. De constructor verwacht een (derde) parameter van het type `DateTime` (dit is de datum van de activiteit). Als we die datum uit het tekstvak lezen, zitten we echter met een waarde van het type `String` opgescheept. Een `String` omzetten naar het type `DateTime` kan met de expressie: "`Convert.ToDateTime(String-waarde)`". De kortere ronde haken-notatie om te converteren, werkt in deze context niet.

2. Het nieuwe `Activiteit`-object dat je in stap 1 aangemaakt hebt, laat je nu effectief toevoegen aan de databank! Gebruik hiervoor de methode `CreateRecord()` van het data-accessobject `_activiteitDA`.

3. Laat de tekstvakken met de beschrijving en met de datum terug leegmaken, zodat het formulier klaarstaat om een volgende nieuwe activiteit te kunnen ingeven.

Test zeker uit of de nieuwe activiteiten die je toevoegt via `ActiviteitToevoegenForm` ook te zien zijn in het formulier `ActiviteitenForm`.

9 Spiekbriefjes

9.1 Switch

Voorbeeld 1:

```
public String BepaalGraad(int punten)
{
    String graad = "";

    switch (punten)
    {
        case 12:
        case 13:
            graad = "voldoening";
            break;
        case 14:
        case 15:
            graad = "onderscheiding";
            break;
        case 16:
        case 17:
            graad = "grote onderscheiding";
            break;
        case 18:
        case 19:
        case 20:
            graad = "grootste onderscheiding";
            break;
        default:
            graad = "onvoldoende";
            break;
    }

    return graad;
}
```

Opmerking:

- De switch in **C#** is iets strikter dan in **Java**.

  Een extra regel is er dat je elke case waarbij je code opneemt, moet afsluiten met een `break;`. In bovenstaand voorbeeldje is dit zo voor `case 13:`, `case 15:`, `case 17:`, en `case 20:`.

Bij Java was je niet verplicht om bij elke case een `break;` te plaatsen, al kon er ééntje vergeten je programma wel danig in de war sturen.

Als de switch een `default:` bevat, moet deze in C# ook afgesloten worden met een `break;`.

9.2 Wiskundige operatoren

Als je twee gehele getallen (gegevenstype `int`) door elkaar deelt met de **/-operator**, dan wordt de **gehele deling** toegepast. Een gehele deling levert altijd een `int`-waarde op als resultaat.

Enkele voorbeeldjes:

```
Immediate Window
? 6 / 3
2
? 7 / 3
2
? 8 / 3
2
? 9 / 3
3
```

Bij de gehele deling levert 8 / 3 dus geen 2,6666… op, maar worden alle decimalen (cijfers na de komma) gewoon 'weggesmeten' en is de uitkomst bijgevolg een ronde 2.

In deze context kunnen we het ook over de **modulo**-bewerking hebben. Deze bewerking levert **de rest** na een gehele deling op en wordt in C# net als in Java uitgevoerd met de **%-operator**.

Bijvoorbeeld:

```
Immediate Window
? 6 % 3
0
? 7 % 3
1
? 8 % 3
2
? 9 % 3
0
```

7 % 3 levert als uitkomst 1 op, want als je een gehele deling 7 / 3 uitvoert blijf je met een rest van 1 over.

9.3 Conversies met numerieke gegevenstypen en `String`

Een waarde van een bepaald type opslaan in een variabele van een ander type vraagt bijna altijd een expliciete **conversie** (=cast) in C#. Met onderstaande voorbeeldjes zou je alle mogelijke conversies tussen numerieke typen en het type `String` moeten kunnen uitvoeren.

Conversie van een numeriek type naar `String`

Een voorbeeldje waar een `int`-waarde omgezet wordt naar het type `String`.

```
int getal = 5;
String getalInTekst = getal.ToString();
```

De variabele `getalInTekst` zal na bovenstaande code de tekstwaarde **"5"** bevatten.

Bij de methode `ToString()` kan je via een parameter een notatie laten forceren. Bijvoorbeeld:

```
double getal = 2.99;
String getalInTekst = getal.ToString("C");
```

De variabele `getalInTekst` zal na bovenstaande code de tekstwaarde **"€ 2,99"** bevatten.

```
double getal = 0.05;
String getalInTekst = getal.ToString("P0");
```

De variabele `getalInTekst` zal na bovenstaande code de tekstwaarde **"5%"** bevatten.

Conversie van `String` naar een numeriek type

Een voorbeeldje:

```
String getalInTekst = "100";
int getal = Convert.ToInt32(getalInTekst);
```

De variabele `getal` zal na bovenstaande code de waarde **100** bevatten.

- Gebruik `Convert.ToDecimal()` om een `String`-waarde om te zetten naar het type `decimal`.
- Gebruik `Convert.ToDouble()` om een `String`-waarde om te zetten naar het type `double`.

Conversie tussen numeriek typen

Een conversie van het type `double` naar `int`:

```
double getal1 = 2.99;
int getal2 = (int) getal1
```

De variabele `getal2` zal na bovenstaande code de waarde **3** bevatten.

Een conversie van het type `double` naar het type `decimal`:

```
double getal1 = 2.99;
decimal getal2 = (decimal) getal1
```

De variabele `getal2` zal na bovenstaande code de waarde **2.99** bevatten.

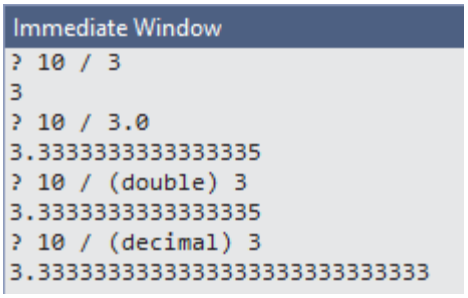
Opgelet:

```
decimal getal1 = (decimal) 2.99
```

Het letterlijke getal **2.99** wordt aanzien van het type `double`. Wil je deze waarde opslaan in een `decimal`-variabele is er dus een conversie naar het type `decimal` nodig.

Numerieke gegevenstypen en delingen

Twee getallen delen door elkaar kan - afhankelijk van het type van deze getallen - verschillende resultaten opleveren. Bijvoorbeeld in het venster Immediate:



Bovenstaande vier delingen leverden respectievelijk een resultaat op van het type: `int`, `double`, `double` en `decimal`.

9.4 Kleuren (Color)

In het venster Properties kan je via de eigenschap `BackColor` de besturingselementen op een formulier een kleur geven. Deze eigenschap `BackColor` kan je bijgevolg ook in je programmacode gebruiken.

Om een tekstvak `uitvoerTextBox` in het geel te plaatsen:

```
uitvoerTextBox.BackColor = Color.Yellow;
```

9.5 De collectie `List<>`

In onderstaande voorbeeldjes gebruiken we:

- `Leerling`: **de klasse** waarmee we een leerling beschrijven.
- `leerling`: **een object** van de klasse `Leerling`.
- `_leerlingen`: **een lijst** (`List`) waarin we alle leerlingen uit een bepaalde klas willen verzamelen.
- `nr`: een getal dat **het indexnummer** van een bepaald `Leerling`-object in de lijst `_leerlingen` voorstelt.

Een lijst declareren

```
private List<Leerling> _leerlingen;
```

Een lijst initialiseren

```
_leerlingen = new List<Leerling>();
```

Een element toevoegen aan een lijst

```
_leerlingen.Add(leerling);
```

Hoeveel elementen bevat een lijst

```
_leerlingen.Count
```

Een element opvragen uit een lijst (op basis van het indexnummer)

```
_leerlingen[nr]
```

Een element wissen uit een lijst (op basis van het indexnummer)

```
_leerlingen.RemoveAt(nr);
```

Met een foreach-lus alle elementen uit een lijst doorlopen

```
foreach (Leerling l in _leerlingen)
{}
```

Met een for-lus alle elementen uit een lijst (van voor naar achter) doorlopen

```
for (int i = 0; i < _leerlingen.Count; i++)
{}
```

Met een for-lus alle elementen uit een lijst (van achter naar voor) doorlopen

```
for (int i = _leerlingen.Count-1; i >= 0; i--)
{}
```

9.6 Code in de Data Access Layer

Connectie met databank

De noodzakelijke code om in een Data Access klasse (hier noemden we de klasse `KlasseDA`) de connectie met de MySQL-databank te kunnen leggen.

```
class KlasseDA
{
    private String _connString;
    private MySqlConnection _mySqlConnection;

    public KlasseDA()
    {
        // connectiestring voor de MySQL-databank wandelroutes
        _connString = "server=localhost;user id=root;Password=Leerling123;
                       database=naamdatabase";
        // initialiseer de connectie op basis van de connectiestring
        _mySqlConnection = new MySqlConnection(_connString);
    }
}
```

Data Access methode - 1e vorm

Als je een SQL-statement wil uitvoeren waarmee je gegevens **leest** uit de databank en die gelezen gegevens bestaan uit **meerdere rijen en/of kolommen**!

We laten deze methode een lijst terug geven van objecten. Hier van de fictieve klasse `Klasse`.

```
public List<Klasse> leesMethode(eventueel met parameters)
{
    List<Klasse> lijst = new List<Klasse>();

    // SQL-statement (eventueel met @parameter)
    String sql = "stel hier je SQL-statement op";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // eventuele @parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@Parameter1", waarde voor parameter 1);
    mySqlCommand.Parameters.AddWithValue("@Parameter2", waarde voor parameter 2);
    ....

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // met _____________ laat je het leescommando opstarten
    // ingelezen informatie komt in mySqlDataReader terecht
    MySqlDataReader mySqlDataReader = mySqlCommand._____________;

    // lusje om alle records in mySqlDataReader te overlopen
    while (mySqlDataReader.Read()==true)
    {
        // nieuwe object maken met de actieve record in mySqlDataReader
        Klasse object =
             new Klasse ((int) mySqlDataReader["veld1"],
                         mySqlDataReader["veld2"].ToString(),
                         (decimal) mySqlDataReader["veld3"], ...)

        // voeg object toe aan de lijst
        lijst.Add(object);
    }

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();

    // lijst met alle objecten teruggeven
    return lijst;
}
```

Data Access methode - 2e vorm

Als je een SQL-statement wil uitvoeren waarmee je gegevens **wijzigt** in de databank! Met "gegevens wijzigen" bedoelen we o.a. updaten, toevoegen of verwijderen:

```
public void actieMethode(eventueel met parameters)
{
    // SQL-statement (eventueel met @parameters)
    String sql = "stel hier je SQL-statement op";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);
```

```
    // eventuele @parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@Parameter1", waarde voor parameter 1);
    mySqlCommand.Parameters.AddWithValue("@Parameter2", waarde voor parameter 2);
    ....

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // ____________________ om een MySqlCommand te starten dat GEEN gegevens inleest
    mySqlCommand.____________________;

    // de connectie met de databank terug sluiten
    _mySqlConnection.Close();
}
```

Data Access methode - 3e vorm

Als je een SQL-statement wil uitvoeren waarmee je uit de databank **één waarde leest**!

In onderstaande code hebben we als voorbeeld een methode uitgewerkt die een `int`-waarde uit de databank leest (en laat retourneren). Uiteraard kon dit evengoed een `String`, `decimal`, … zijn.

```
public int leesMethode(eventueel met parameters)
{
    // SQL-statement (eventueel met @parameter)
    String sql = "stel hier je SQL-statement op";

    // SQL-commando aanmaken op basis van ons SQL-statement
    MySqlCommand mySqlCommand = new MySqlCommand(sql, _mySqlConnection);

    // eventuele @parameters in het SQL-commando hun waarde geven
    mySqlCommand.Parameters.AddWithValue("@Parameter1", waarde voor parameter 1);
    mySqlCommand.Parameters.AddWithValue("@Parameter2", waarde voor parameter 2);
    ....

    // de connectie met de databank openen
    _mySqlConnection.Open();

    // met _____________ laat je het leescommando opstarten
    // ingelezen informatie komt in hulpvariabele terecht (hier een int)
    int resultaat = Convert.ToInt32(mySqlCommand._________________);

    // de connectie met de databank sluiten
    _mySqlConnection.Close();

    return resultaat;
}
```