

howest
hogeschool

Methoden

Programming Basics

INHOUD

1	VOORBEREIDING: MEERDERE PROJECTEN IN ÉÉN SOLUTION	3
2	METHODEN DECLAREREN	4
2.1	Voorbeeld: methode ShowCountry	5
3	EEN METHODE AANROEPEN	6
4	PARAMETERS	9
5	RETURN VALUE	11
6	SCOPE	15
7	OVERLOADING	17
8	OPTIONELE PARAMETERS	20
9	EVENT HANDLER METHODEN ⇔ EIGEN METHODEN	24
9.1	Event handler methoden	24
9.2	Check je eigen methode	26
10	METHODEN DOCUMENTEREN	29

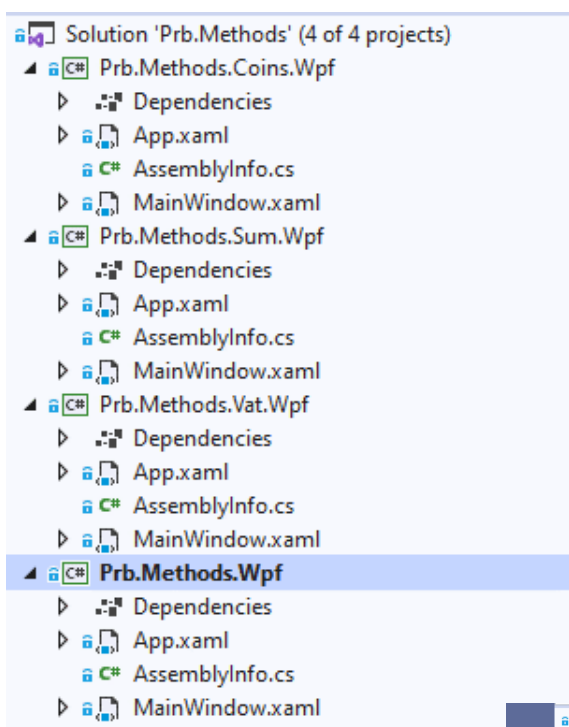
1 VOORBEREIDING: MEERDERE PROJECTEN IN ÉÉN SOLUTION

Vooraleer we met de leerstof van dit hoofdstuk starten, herhalen we kort iets uit het inleidende hoofdstuk.

In dit hoofdstuk werken we met één solution, maar binnen de solution maken we een aantal WPF projecten aan. Hoe dit gebeurt, zie je straks stap voor stap.

Het is echter zo dat wanneer je het programma laat uitvoeren, standaard altijd het eerste project wordt uitgevoerd: je tweede, derde, ... project krijg je schijnbaar niet opgestart.

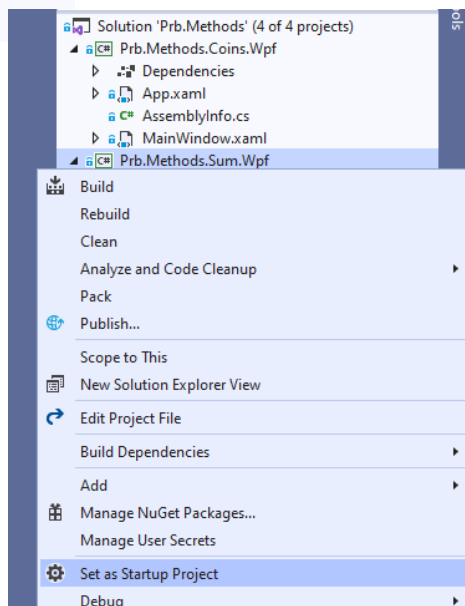
Dit komt omdat elke solution slechts één zogenaamd “Startup” project kan hebben. Om straks te kunnen schakelen tussen de verschillende projecten zal je dus telkens het project dat je wil bekijken als Startup project moeten instellen.



In dit voorbeeld zie je dat er vier projecten (Prb.Methods.Coins.Wpf, Prb.Methods.Sum.Wpf, Prb.Methods.Vat.Wpf en Prb.Methods.Wpf) in één solution (Prb.Methods) zitten.

Je ziet in de figuur ook dat de naam van het laatste project (Prb.Methods.Wpf) vetter afgebeeld wordt dan de drie andere projecten: dit geeft aan dat dit project momenteel het Startup project is. Als je m.a.w. je programma start, zal het dit project zijn dat gestart wordt.

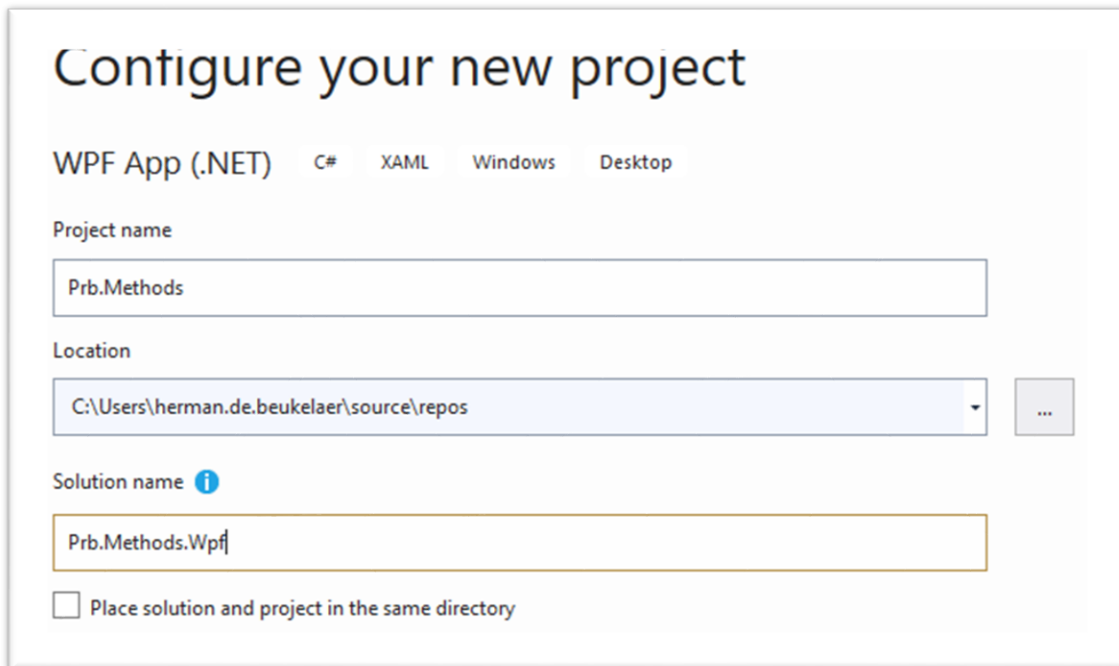
Wil je dat er een ander project start (bv het project Prb.Methods.Sum.Wpf) dan klik je daar met de rechtermuisknop op, en kies je in het snelmenu voor “Set as Startup Project”. Wanneer je nu je programma start, zal er met dit project gewerkt worden.



2 METHODEN DECLAREREN

Maak een nieuwe Visual Studio solution **Prb.Methods**.

Maak in de solution Methoden een nieuwe WPF Application **Prb.Methods.Wpf**



Configure your new project

WPF App (.NET) C# XAML Windows Desktop

Project name

Prb.Methods

Location

C:\Users\herman.de.beukelaer\source\repos

Solution name ⓘ

Prb.Methods.Wpf

☐ Place solution and project in the same directory

Location: de locatie kies je uiteraard zelf.

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

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

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

```
returnType MethodeNaam (parameterlijst)
{
    //statements binnen de methode
}
```

2.1 VOORBEELD: METHODE SHOWCOUNTRY

Maak in je code-behind venster een methode die het land “België” in een berichtvenster toont.

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

    void ShowCountry()
    {
        string country = "België";
        MessageBox.Show("Het land is " + country);
    }
}
```

- De nieuwe methode heeft de naam ShowCountry.
- ShowCountry geeft niks terug: sleutelwoord void. Bij sommige andere methoden wordt als resultaat van de statements een string, getal of welk object ook als ‘uitkomst’ teruggestuurd. Meer hierover later.
- ShowCountry ontvangt geen parameters: lege ronde haken na de naam van de methode.

Als je nu je programma zou runnen, dan zal er niets gebeuren.

Je hebt wel een methode geschreven maar er is nog “niets” dat deze methode gebruikt (= aanroept).

Hoe je dit doet, zie je in volgende paragraaf.



Wat moet je onthouden?

- Een methode maak je aan binnen een klasse.
- Een methode die enkel “iets doet” en niets teruggeeft, begint met het sleutelwoord **void**
- De naam van een methode is vrij te kiezen maar
 - omschrijft dus best wat de methode zal doen : ShowCountry, CalculateSum, CalculateSalesPrices ...
 - gebruik de meervoudsvorm indien van toepassing: dient een methode om meerdere landen te tonen, dan gebruik je dus liever ShowCountries, dient de methode maar 1 land te tonen, dan gebruik je bijvoorbeeld ShowCountry, ...
 - je gebruikt de PascalCase notatie (alle woorden aan elkaar en elk woord, ook het eerste, begint met een hoofdletter)
- De naam van de methode wordt steeds gevolgd door 2 ronde haakjes. Verwacht de methode geen argumenten (parameters), dan plaats je ook niets tussen deze ronde haakjes.
- De instructies (statements) van de methoden bevinden zich steeds tussen accolades.

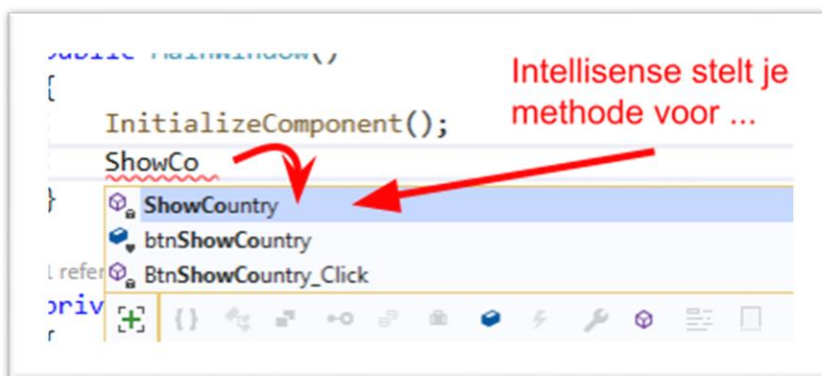
3 EEN METHODE AANROEPEN

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

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

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

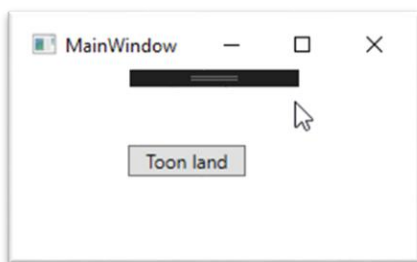
Merk op dat je eigen methode onmiddellijk ter beschikking is met Visual Studio Intellisense.



Start je programma even op en bekijk het resultaat.

Verwijder de zopas gemaakte instructie (de aanroep van de methode dus) terug uit de MainWindow constructor.

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



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

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



Je merkt dat Visual Studio standaard de event handler een naam geeft die start met de naam van de control (hier de knop “btnShowCountry”) en eindigt met de naam van het event (hier “Click”), gescheiden door een underscore.

We hanteren echter de conventie dat methodenamen met een hoofdletter moeten starten. Event handlers zijn ook (speciale) methoden, dus ook hier geldt onze naamgevingsconventie. Visual Studio geeft zelf ook aan dat de huidige naam eigenlijk in strijd is met de conventies.

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

IDE1006: Naming rule violation: These words must begin with upper case characters: btnShowCountry_Click
Show potential fixes (Alt+Enter or Ctrl+.)

Bovendien kan Visual Studio het probleem ook snel voor je oplossen. Klik even op het gloeilamp-icoontje:

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

Fix Name Violation: BtnShowCountry_Click
Suppress or Configure issues

IDE1006 Naming rule violation: These words must begin with upper case characters: btnShowCountry_Click
Lines 14 to 16

```
private void btnShowCountry_Click(object sender, RoutedEventArgs e)
private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    ...
}
```

Change Style Options
Preview changes

Klik nu op “Fix Name Violation: ...”. Visual studio past de naam van de methode automatisch aan, zowel in de code behind als in de XAML.

We gaan nu verder met de aangepaste event handler, met naam volgens de conventies:

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

- Vanuit deze event handler methode kan je je eigen methode ShowCountry aanroepen:

```
private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    ShowCountry();
}
```

- Onze volledige code ziet er nu zo uit:

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

```
private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    ShowCountry();
}

void ShowCountry()
{
    string country = "België";
    MessageBox.Show("Het land is " + country);
}
}
```


4 PARAMETERS

In de methode ShowCountry kunnen we maar één land laten zien via de MessageBox. Om ook andere landen te kunnen tonen, declareren we de variabele niet in de methode, maar maken we er een parameter (= argument) van.

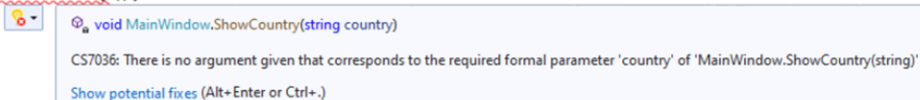
```
void ShowCountry(string country)
{
    MessageBox.Show("Het land is " + country);
}
```

Je zal merken dat zich nu in de oproep van de methode in de event-handler van btnShowCountry een fout voordoet:

```
private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    ShowCountry();
}
```

De reden is dat wij onze methode ShowCountry hebben aangepast, waarbij er nu een parameter verwacht wordt: void ShowCountry() werd void ShowCountry(string country). Als je aangeeft dat je methode een parameter verwacht, dan moet je die ook verplicht meegeven wanneer je de methode gebruikt.

```
private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    ShowCountry();
}
```



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

```
string country = "Frankrijk";
ShowCountry(country);
```

Onze volledige code ziet er nu zo uit:

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

```

private void BtnShowCountry_Click(object sender, RoutedEventArgs e)
{
    string country = "Frankrijk";
    ShowCountry(country);
}

void ShowCountry(string country)
{
    MessageBox.Show("Het land is " + country);
}
}

```



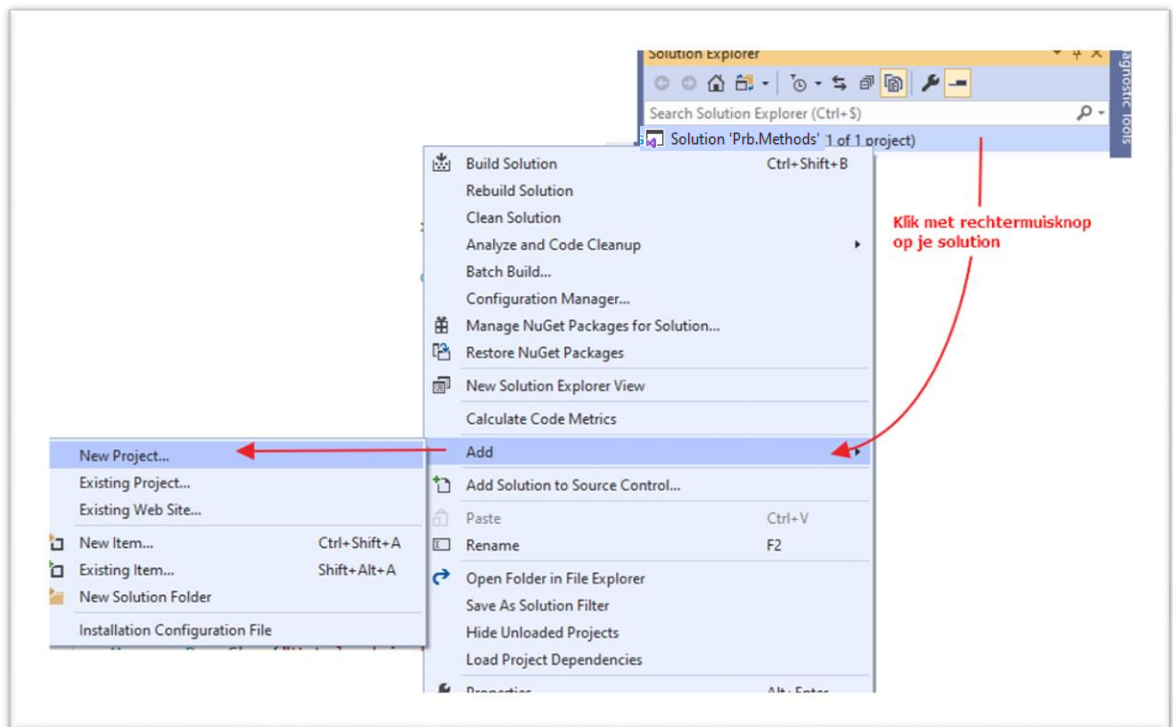
Wat moet je onthouden?

- Een methode kan 0, 1 of meer parameters verwachten.
- Verwacht een methode parameters, dan dien je die verplicht mee te geven wanneer je de methode gebruikt (tenzij het optionele parameters zijn, zie verder).
- Parameters definieer je tussen de ronde haakjes na de naam van de methode.
Een parameter definieer je op dezelfde manier als je een variabele declareert (met andere woorden: de naam van de parameter dient voorafgegaan te worden door zijn type).
Bijvoorbeeld: void CalculateSum(int number1, int number2)
- Wanneer je een methode oproept (= gebruikt) die argumenten verwacht, dan geef je uiteraard enkel de waarden door.
Bijvoorbeeld: CalculateSum(5,7)
Bijvoorbeeld: CalculateSum(numberOne, numberTwo)
- De volgorde waarin methoden zich binnen je klasse bevinden, speelt geen rol.
Het maakt geen verschil of je zelfgeschreven methode ShowCountry voor of achter de event handler BtnShowCountry_Click staat. We spreken wel af dat we onze eigen methoden groeperen en dus niet door elkaar zetten tussen de event handlers in.

5 RETURN VALUE

Hierboven zagen we methodes die niets retourneerden. De code in de methode werd uitgevoerd, maar er werd geen resultaat van de bewerkingen teruggegeven aan het statement dat de methode aanriep. We gaan nu aan de slag met methoden die wel een resultaat teruggeven, bv. de som van twee getallen.

- Maak een nieuw project aan in de bestaande solution Prb.Methods en noem dit nieuw project **Prb.Methods.Sum.Wpf**.



Configure your new project

WPF App (.NET) C# XAML Windows Desktop

Project name

Location

 ...

- Voeg in de **code behind** een nieuwe methode CalculateSum toe:

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

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }
}
```

Deze methode zullen we nu even ontleden:

- De methode heeft als naam CalculateSum:

```
int CalculcateSum(int numberOne, int numberTwo)
```

- CalculateSum retourneert een geheel getal (int) naar de plaats van aanroep:

```
int CalculateSum(int numberOne, int numberTwo)
```

- CalculateSum ontvangt twee gehele getallen als parameters:

```
int CalculateSum(int numberOne, int numberTwo)
```

- Binnen de methode worden de twee gehele getallen opgeteld

```
return nubmerOne + numberTwo;
```

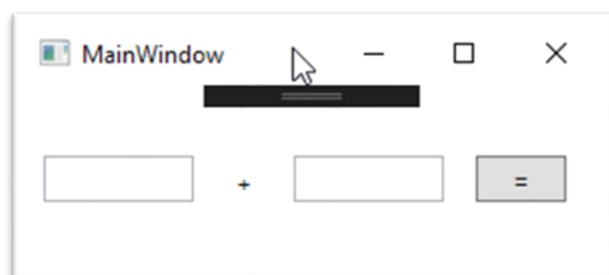
- CalculateSum retourneert het resultaat van de optelling naar de plaats van aanroep

```
return numberOne + numberTwo;
```

- We zullen deze methode nu gebruiken in onze applicatie:

- Voeg twee textboxes toe

- De linker textbox geef je de naam "txtLeftNumber" en de content "0".
- De rechter textbox geef je de naam "txtRightNumber" en ook de content "0".



- Voeg een label toe met content “+” en plaats het tussen de twee textboxes (geen naam nodig).
- Voeg een button toe, geef deze de naam “btnCalculateSum” en plaats deze na de tweede textbox.
- Dubbelklik nu op de button zodat er automatisch een event-handler-methode aangemaakt wordt. Pas de standaard naam aan zodat de methode start met een hoofdletter, volgens de conventies.

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

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }

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

- Voorzie de methode-statements:

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

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }

    private void BtnCalculateSum_Click(object sender, RoutedEventArgs e)
    {
        int leftNumber = int.Parse(txtLeftNumber.Text);
        int rightNumber = int.Parse(txtRightNumber.Text);
        int sum = CalculateSum(leftNumber, rightNumber);
        MessageBox.Show("De som van " + leftNumber + " en " + rightNumber +
            " is " + sum, "Som berekenen");
    }
}
```

- Deze statements doen het volgende:

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

Parse de inhoud van de “txtLeftNumber” naar het data type int en plaats deze in een nieuwe variabele “int leftNumber”:

```
int leftNumber = int.Parse(txtLeftNumber.Text);
```

- Doe hetzelfde met “txtRightNumber”: parse de inhoud van “txtRightNumber” naar het data type int en plaats deze in een nieuwe variabele “int rightNumber”:

```
int rightNumber = int.Parse(txtRightNumber.Text);
```

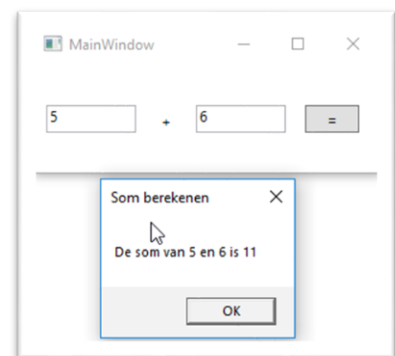
- We declareren een nieuwe variabele van data type int met de naam “sum”. Deze variabele ontvangt de geretourneerde waarde van onze methode “CalculateSum” die variabelen “leftNumber” en “rightNumber” meekrijgt als parameters. Met andere woorden: we geven het linker- en het rechtergetal mee aan onze methode CalculateSum. In deze methode worden de getallen opgeteld en het resultaat wordt geretourneerd naar deze plaats van aanroep. Daar wordt het resultaat in de nieuwe variabele “int sum” gestopt:

```
int sum = CalculateSum(leftNumber, rightNumber);
```

- Via een MessageBox tonen we deze optelling met onze drie variabelen “leftNumber”, “rightNumber” en “sum”:

```
MessageBox.Show("De som van " + leftNumber + " en " + rightNumber + " is " + sum, "Som berekenen");
```

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



Wat moet je onthouden?

- Wens je een methode te maken die een waarde retourneert, vervang dan het woord “void” voor de methodenaam door het type van het gegeven dat de methode zal retourneren
Bijvoorbeeld : string MakeFullName(string firstName, string lastName)
Bijvoorbeeld : double MultiplyNumbers(double number1, double number2)
- Je ontvangt de retourwaarde van een methode door deze toe te kennen aan een variabele, de eigenschap van een control ...
Bijvoorbeeld : double result = MultiplyNumbers(value1, value2);
Bijvoorbeeld : string fullName = MakeFullName(“Jan”, “Janssens”);
- Hoewel je dit uiteraard doorgaans wel gaat doen, ben je in principe niet verplicht om de retourwaarde van een methode op te vangen. Je kan dus bijvoorbeeld schrijven :
MultiplyNumbers(5,8);
Dit zal GEEN fout opleveren, alleen kun je de vraag stellen waarom je deze methode dan gaat oproepen?

6 SCOPE

Binnen de methode BtnCalculateSum_Click werden drie variabelen gedeclareerd: leftNumber, rightNumber en sum.

Variabelen die gedeclareerd werden binnen een methode hebben deze methode als bereik of scope. Dit wil zeggen dat deze variabelen buiten deze methode totaal ongekend zijn en dus onbruikbaar.

- Voorzie een nieuwe methode ScopeTest:

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

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }

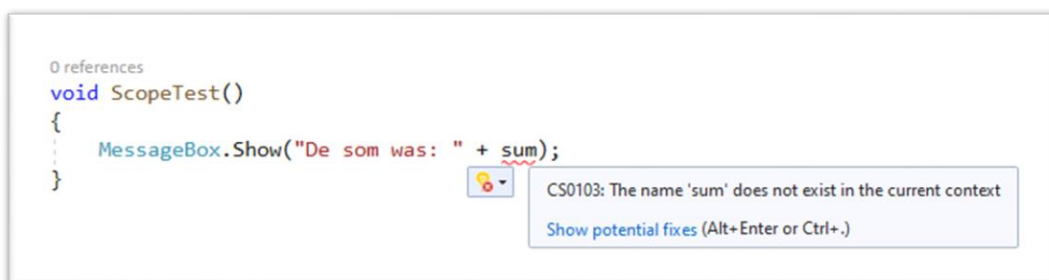
    private void BtnCalculateSum_Click(object sender, RoutedEventArgs e)
    {
        int leftNumber = int.Parse(txtLeftNumber.Text);
        int rightNumber = int.Parse(txtRightNumber.Text);
        int sum = BerekenSommen(leftNumber, rightNumber);
        MessageBox.Show("De som van " + leftNumber + " en " + rightNumber + " is "
+ sum, "Som berekenen");
    }

    void ScopeTest()
    {
        MessageBox.Show("De som was: " + sum);
    }
}
```

Wanneer je de toepassing wil uitvoeren krijg je een compileerfout:

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

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



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

Wanneer je de variabele “sum” wenst te gebruiken buiten de methode “BtnCalculateSum_Click” kan je deze declareren op klassenniveau (= globale variabele): buiten elke methode, maar binnen de accolades van de klasse.

- Pas de toepassing als volgt aan:

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

    public MainWindow()
    {
        InitializeComponent();
    }

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }

    private void BtnCalculateSum_Click(object sender, RoutedEventArgs e)
    {
        int leftNumber = int.Parse(txtLeftNumber.Text);
        int rightNumber = int.Parse(txtRightNumber.Text);
        sum = BerekenSommen(leftNumber, rightNumber);
        MessageBox.Show("De som van " + leftNumber + " en " + rightNumber + " is "
+ sum, "Som berekenen");
    }

    void ScopeTest()
    {
        MessageBox.Show("De som was: " + sum);
    }
}
```

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



Wat moet je onthouden?

- Het bereik van een variabele is beperkt tot de methode of de klasse waarbinnen deze gedeclareerd wordt (tenzij je het sleutelwoord public gebruikt, zie later).
- Declareer je een variabele in een methode, dan is deze variabele enkel gekend in de methode. Dit noemen we een **lokale variabele**.
- Declareer je een variabele buiten alle methodes, maar binnen de klasse, dan is deze variabele in alle methodes (en dus ook event-handlers) van de klasse gekend. Dit noemen we een **globale variabele**.
- Werk zoveel mogelijk met lokale variabelen en denk er goed over na welke variabelen echt globaal moeten zijn. Hoe meer informatie je deelt tussen methoden, hoe minder onafhankelijk ze worden. Zo worden ze minder herbruikbaar, en ga je makkelijker fouten maken. Een wildgroei aan globale variabelen kan vaak vermeden worden door de methoden anders te organiseren, bv. door argumenten door te geven van de ene naar de andere methoden.

7 OVERLOADING

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

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

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

    public MainWindow()
    {
        InitializeComponent();
    }

    int CalculateSum(int numberOne, int numberTwo)
    {
        return numberOne + numberTwo;
    }

    int CalculateSum(string numberOneText, string numberTwoText)
    {
        int numberOne = int.Parse(numberOneText);
        int numberTwo = int.Parse(numberTwoText);
        return CalculateSum(numberOne, numberTwo);
    }

    private void BtnCalculateSum_Click(object sender, RoutedEventArgs e)
    {
        string leftNumber = txtLeftNumber.Text;
        string rightNumber = txtRightNumber.Text;
        sum = CalculateSum(leftNumber, rightNumber);
        MessageBox.Show("De som van " + leftNumber + " en " + rightNumber + " is "
+ sum, "Som berekenen");
    }

    void ScopeTest()
    {
        MessageBox.Show("De som was: " + sum);
    }
}
```

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

```
sum = CalculateSum(;  
Message ▲ 1 of 2 ▼ int MainWindow.CalculateSum(int numberOne, int numberTwo)  
  
sum = CalculateSum(;  
Message ▲ 2 of 2 ▼ int MainWindow.CalculateSum(string numberOneText, string numberTwoText)
```

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

```
Console.WriteLine() _  
▲ 1 of 19 ▼ void Console.WriteLine()  
Writes the current line terminator to the standard output stream.
```



Wat moet je onthouden?

Een methode overladen kan je op 2 manieren:

- Het aantal argumenten van de methoden (die dus dezelfde naam krijgen) zijn gelijk maar minstens 1 argument heeft een ander datatype.

Bijvoorbeeld :

- double DivideNumbers(int number1, int number2)
- double DivideNumbers (double number1, double number2)
- double DivideNumbers(double number1, int number2)

Bovenstaande methoden kunnen alle drie perfect binnen dezelfde klasse voorkomen: ze hebben alle drie dezelfde naam en ze verwachten steeds twee argumenten, maar in alle drie de gevallen zijn het type van de argumenten verschillend (int en int, double en double, double en int).

- Het aantal argumenten van de methoden (die dus dezelfde naam krijgen) zijn verschillend.

Bijvoorbeeld :

- int AddNumbers(int number1, int number2)
- int AddNumbers(int number1, int number2, int number3)
- int AddNumbers(int number1, int number2, int number3, int number4)

Bovenstaande methoden kunnen alle drie perfect binnen dezelfde klasse voorkomen: het aantal argumenten dat ze verwachten zijn immers verschillend.

OPGEPAST 1: overladen kan NIET door enkel het retourtype te wijzigen.

Bijvoorbeeld onderstaande methoden kunnen NIET binnen dezelfde klasse voorkomen:

- `int AddNumbers(int number1, int number2)`
- `double AddNumbers(int number1, int number2)`

De reden hiervoor is dat de compiler in dit geval niet meer kan afleiden welk van beide methoden je wil gebruiken wanneer je bv. `AddNumbers(2,5)` oproept.

OPGEPAST 2: onderstaande kan dan wel weer:

- `int AddNumbers(int number1, int number2)`
- `double AddNumbers(double number1, int number2)`

Dit kun dus wel omdat de argumenten verschillende types hebben. Wordt de eerste versie gebruikt, dan wordt een `int` geretourneerd, wordt de tweede versie gebruikt, dan wordt een `double` geretourneerd.

8 OPTIONELE PARAMETERS

Soms maak je methoden waarbij het niet altijd noodzakelijk is dat de gebruiker van de methode alle argumenten meegeeft.

In de voorgaande voorbeelden was je steeds verplicht, wanneer je één van de bestaande methoden met parameters gebruikte, om telkens netjes alle argumenten mee te geven.

Je kan die verplichting omzeilen door argumenten optioneel te maken.

Er zijn hiervoor 2 regels :

- Om een argument optioneel te maken, dien je er een standaardwaarde aan toe te kennen.
- Optionele argumenten komen steeds na vereiste argumenten in de signatuur van de methode.

Maak een nieuw project aan binnen je solution en geef het deze keer de naam Prb.Methods.Currency.Wpf.

In plaats van jullie zelf controls op je venster te laten tekenen en de eigenschappen bij te werken, maken we je het eens wat makkelijker en plakken hieronder de XAML code van ons venster.

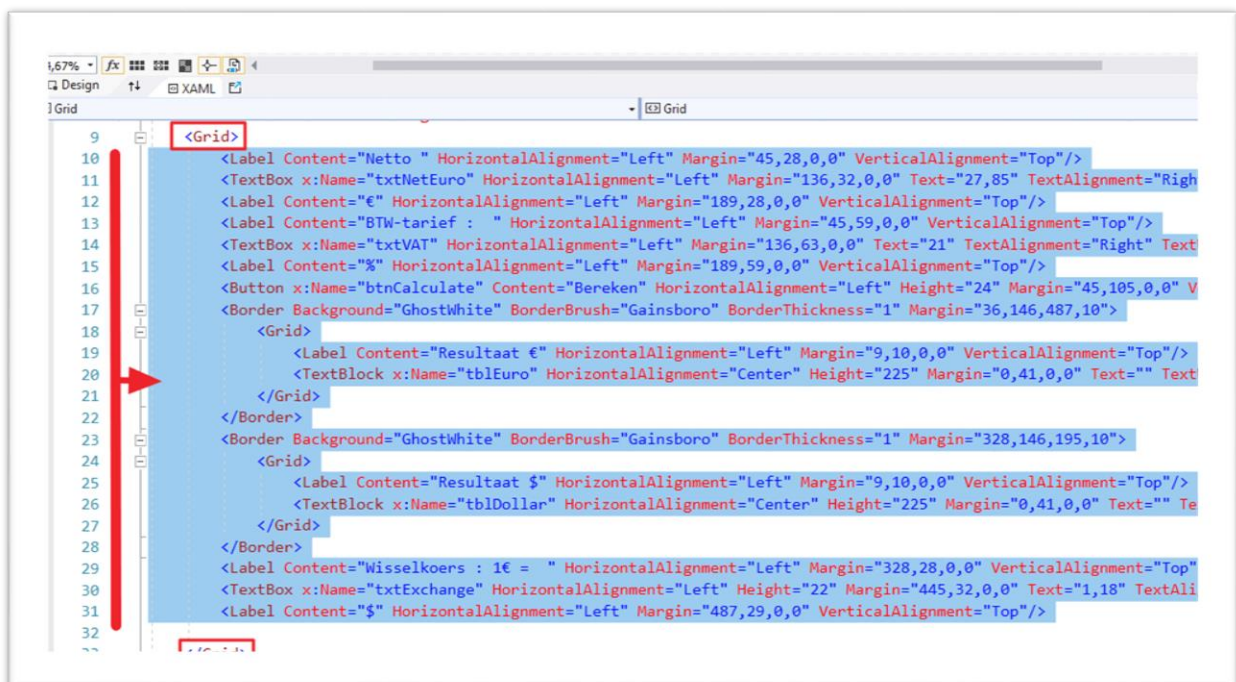
Kopieer en plak dus onderstaande XAML code in je XAML venster tussen de tags <GRID> en </GRID>:

```
<Label Content="Netto " HorizontalAlignment="Left" Margin="45,28,0,0"
VerticalAlignment="Top"/>
<TextBox x:Name="txtNetEuro" HorizontalAlignment="Left" Margin="136,32,0,0"
Text="27,85" TextAlignment="Right" TextWrapping="Wrap" VerticalAlignment="Top"
Width="48"/>
<Label Content="€" HorizontalAlignment="Left" Margin="189,28,0,0"
VerticalAlignment="Top"/>
<Label Content="BTW-tarief : " HorizontalAlignment="Left" Margin="45,59,0,0"
VerticalAlignment="Top"/>
<TextBox x:Name="txtVAT" HorizontalAlignment="Left" Margin="136,63,0,0"
Text="21" TextAlignment="Right" TextWrapping="Wrap" VerticalAlignment="Top"
Width="48"/>
<Label Content="%" HorizontalAlignment="Left" Margin="189,59,0,0"
VerticalAlignment="Top"/>
<Button x:Name="btnCalculate" Content="Bereken" HorizontalAlignment="Left"
Height="24" Margin="45,105,0,0" VerticalAlignment="Top" Width="172"
Click="btnCalculate_Click"/>
<Border Background="GhostWhite" BorderBrush="Gainsboro" BorderThickness="1"
Margin="36,146,487,10">
    <Grid>
        <Label Content="Resultaat €" HorizontalAlignment="Left"
Margin="9,10,0,0" VerticalAlignment="Top"/>
        <TextBlock x:Name="tblEuro" HorizontalAlignment="Center" Height="225"
Margin="0,41,0,0" Text="" TextWrapping="Wrap" VerticalAlignment="Top" Width="255"/>
    </Grid>
</Border>
<Border Background="GhostWhite" BorderBrush="Gainsboro" BorderThickness="1"
Margin="328,146,195,10">
    <Grid>
        <Label Content="Resultaat $" HorizontalAlignment="Left"
Margin="9,10,0,0" VerticalAlignment="Top"/>
```

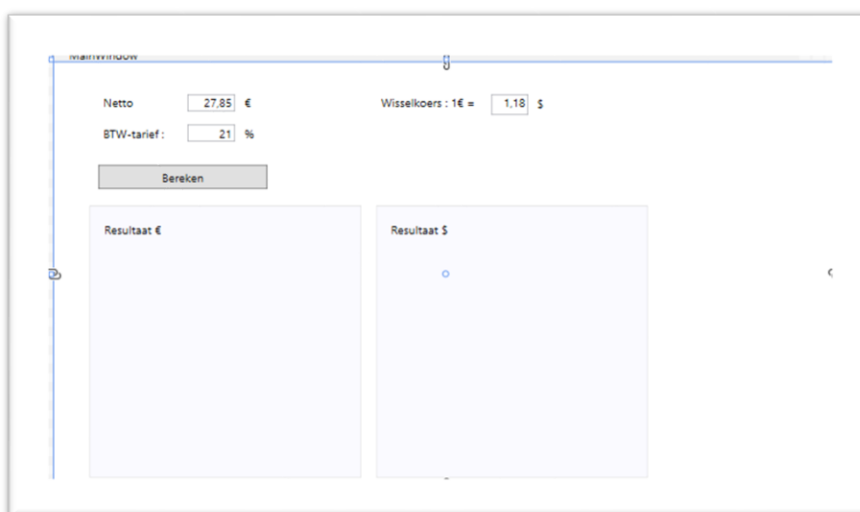
```

        <TextBlock x:Name="tblDollar" HorizontalAlignment="Center"
Height="225" Margin="0,41,0,0" Text="" TextWrapping="Wrap" VerticalAlignment="Top"
Width="255"/>
    </Grid>
</Border>
<Label Content="Wisselkoers : 1€ = " HorizontalAlignment="Left"
Margin="328,28,0,0" VerticalAlignment="Top"/>
<TextBox x:Name="txtExchange" HorizontalAlignment="Left" Height="22"
Margin="445,32,0,0" Text="1,18" TextAlignment="Right" TextWrapping="Wrap"
VerticalAlignment="Top" Width="38"/>
<Label Content="$" HorizontalAlignment="Left" Margin="487,29,0,0"
VerticalAlignment="Top"/>

```



Je design zou er als volgt moeten uitzien:



Als we dit programma uitvoeren is het de bedoeling om de totaalprijs te berekenen zowel in Euro als in Dollar, rekening houdend met het ingevulde BTW-tarief:

MainWindow

Netto €

Wisselkoers : 1€ = \$

BTW-tarief: %

Resultaat €

Netto bedrag = 27,85 EURO

BTW 21,00% = 5,85 EURO

Totaal bedrag = 33,70 EURO

Resultaat \$

Netto bedrag = 32,86 Dollar

BTW 21,00% = 6,90 Dollar

Totaal bedrag = 39,76 Dollar

Dubbeltklik op de knop btnCalculate zodat je in de btnCalculate_Click event handler terecht komt. Laat Visual Studio de naming violation oplossen zodat de event handler methode hernoemd wordt naar BtnCalculate_Click.

Ga net boven deze event handler gaan staan en maak onderstaande methode aan:

```
string GenerateReport(string netInText, string vatInText, string exchangeRateInText = "1",
string currency = "Euro")
{
    double net = double.Parse(netInText);
    double vat = double.Parse(vatInText);
    double exchangeRate = double.Parse(exchangeRateInText);
    net = net * exchangeRate;

    double vatTotal = net * (vat / 100);
    double totalAmount = net + vatTotal;
    string report = $"Netto bedrag = {net.ToString("#,##0.00")} {currency}\n" +
        $"BTW {vat.ToString("#,##0.00")} = {vatTotal.ToString("#,##0.00")} {currency}\n" +
        $"Totaal bedrag = {totalAmount.ToString("#,##0.00")} {currency}";

    return report;
}
```

Verklaring :

- We maken een methode aan met de naam GenerateReport die een string-waarde zal retourneren.
- De methode verwacht 4 argumenten, waarvan er 2 vereist zijn en 2 optioneel.
- De eerste 2 (netInText, vatInText) wijken niet af van wat we eerder zagen en zijn dus vereist.
- Bij de volgende 2 wordt aan de parameters onmiddellijk een (default) waarde toegekend :
 - string exchangeRateInText = "1"
 - string currency = "Euro"

Dit zijn 2 optionele parameters.

Wanneer je deze methode oproept en je geeft GEEN 3^e en 4^e parameterwaarde mee, dan zullen de opgegeven waarden ("1" en "Euro") in de plaats gebruikt worden.

- De rest van de code zou voor zichzelf moeten spreken.
Wat voor jou waarschijnlijk wel nieuw is, is de manier waarop de variabele report gevuld wordt.

Deze manier van werken wordt “String interpolation” genoemd.

We komen hier verder in de cursus zeker nog op terug. We geven wel al kort mee:

- Om string interpolation te gebruiken, plaats je voor de eerste dubbele quote een dollar teken.
- Binnen de tekst (dus binnen de dubbele quotes) kan je de inhoud van variabelen invoegen, door de naam van de variabele wiens waarde je wil gebruiken tussen accolades te plaatsen.
- Om tekst op verschillende regels te krijgen kan je het \n karakter gebruiken (= new line).

In je event handler zelf neem je onderstaande code over:

```
private void BtnCalculate_Click(object sender, RoutedEventArgs e)
{
    tblEuro.Text = GenerateReport(txtNetEuro.Text, txtVAT.Text);
    tblDollar.Text = GenerateReport(txtNetEuro.Text, txtVAT.Text, txtExchange.Text,
    "Dollar");
}
```

We roepen dus onze methode GenerateReport 2 keer op:

- Een eerste keer geven we slechts 2 parameterwaarden mee. Dit betekent dat tijdens de uitvoering van deze methode de standaardwaarden “1” en “Euro” gebruikt zullen worden.
- Een tweede keer geven we de 4 parameterwaarden mee. Dit betekent in dit voorbeeld dat niet de standaardwaarden gebruikt zullen worden, maar de inhoud van txtExchange.Text en “Dollar”.



Wanneer je de versie gebruikt met de optionele argumenten (onze tweede instructie in onze event handler) dan MOET je het argument exchangeRateInText meegeven wanneer je aan het argument currency een waarde wenst toe te kennen.

M.a.w. deze oproep is NIET correct:

```
GenerateReport(txtNetEuro.Text, txtVAT.Text, "Dollar");
```

Waarom niet? Omdat het woord “Dollar” nu zal terechtkomen in het argument exchangeRateInText (die we verder in de code proberen om te zetten naar een double, wat dus niet kan).

Er is eigenlijk wel een mogelijkheid om het argument exchangeRateInText toch over te slaan, nl. door gebruik te maken van *named arguments*. We gaan hier echter in deze cursus niet op in. Wie er meer over wil te weten komen, kan de documentatie van Microsoft over *Named and Optional Arguments* erop naslaan:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments>.

9.1 EVENT HANDLER METHODEN

Event handlers zijn ook methoden, specifieke methoden die worden uitgevoerd op het moment dat er een gebeurtenis (een event) plaatsvindt op een control: bv. een klik op een button, een item dat aangeklikt wordt in een listbox ...

Een event handler methode beperk je in principe best tot enkel de volgende zaken:

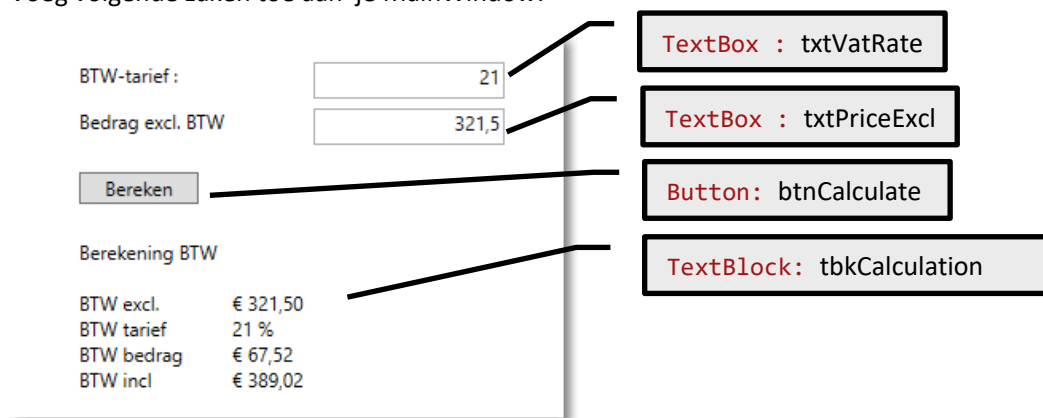
- Declaratie van lokale variabelen.
- Inlezen van de input van de gebruiker.
- Toewijzen van waarden aan de variabelen (indien eenvoudig statement).
- Call(s) naar andere, eventueel eigen methode(n).
- Feedback naar de gebruiker.
- Eventueel code om de applicatie gebruiksvriendelijker te maken (focus,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

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

Event handler methodes moeten altijd simpel blijven om te lezen. De echte intelligentie ligt bij de methoden en later bij de klassen (zie hoofdstuk over objecten en klassen). De event handlers zijn er om de koppeling te maken met de Wpf controls, maar zouden zelf geen algemeen (her)bruikbare logica mogen bevatten die niks met Wpf te maken heeft.

Maak binnen je solution een nieuw project aan met de naam **Prb.Methods.Vat.Wpf**.

Voeg volgende zaken toe aan je MainWindow:



```
private void BtnCalculate_Click(object sender, RoutedEventArgs e)
{
    decimal priceExcl;
    decimal priceIncl;
    float vatRate;
    decimal vat;
    string summary;

    vatRate = float.Parse(txtVatRate.Text);
    priceExcl = decimal.Parse(txtPriceExcl.Text);

    vat = priceExcl * (decimal)vatRate / 100;
    priceIncl = priceExcl + vat;
}
```



```

summary = "Berekening BTW\n\n" +
    $"BTW excl.\t€ {priceExcl.ToString("0.00")}\n" +
    $"BTW tarief\t{vatRate} %\n" +
    $"BTW bedrag\t€ {vat.ToString("0.00")}\n" +
    $"BTW incl\t\t€ {priceIncl.ToString("0.00")}\n";

tbkCalculation.Text = summary;
}

```

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

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

Op die manier houden we de code binnen de event handler methode simpeler. We kunnen de geschreven methodes ook opnieuw gebruiken vanuit andere event handlers en eventueel ook andere projecten. Het berekenen van BTW is immers totaal niet afhankelijk van de Wpf context waarin we nu “toevallig” aan het werken zijn. De code in de event handlers is dat wel: ze interageert met de controls van het Wpf venster.

```

private void BtnCalculate_Click(object sender, RoutedEventArgs e)
{
    decimal priceExcl;
    decimal priceIncl;
    float vatRate;
    decimal vat;
    string summary;

    vatRate = float.Parse(txtVatRate.Text);
    priceExcl = decimal.Parse(txtPriceExcl.Text);

    vat = CalculateVat(vatRate, priceExcl);
    priceIncl = CalculatePriceInclVat(vatRate, priceExcl);

    summary = "Berekening BTW\n\n" +
        $"BTW excl.\t€ {priceExcl.ToString("0.00")}\n" +
        $"BTW tarief\t{vatRate} %\n" +
        $"BTW bedrag\t€ {vat.ToString("0.00")}\n" +
        $"BTW incl\t\t€ {priceIncl.ToString("0.00")}\n";

    tbkCalculation.Text = summary;
}

decimal CalculateVat(float vatRate, decimal priceExcl)
{
    Decimal vat = priceExcl * (decimal)vatRate / 100;
    return vat;
}

decimal CalculatePriceInclVat(float vatRate, decimal priceExcl)
{
    decimal vat = CalculateVat(vatRate, priceExcl);
    return priceExcl + vat;
}

```

Ook het samenstellen van feedback aan de gebruiker op basis van verscheidene stukken informatie, kan beter in een methode ondergebracht worden, aangezien dit geen triviale code is. Zo wordt de event handler weer leesbaarder.

```
private void BtnCalculate_Click(object sender, RoutedEventArgs e)
{
    float vatRate = float.Parse(txtVatRate.Text);
    decimal priceExcl = decimal.Parse(txtPriceExcl.Text);
    tbkCalculation.Text = ShowVatCalculation(vatRate, priceExcl);
}

decimal CalculateVat(float vatRate, decimal priceExcl)
{
    decimal vat = priceExcl * (decimal)vatRate / 100;
    return vat;
}

string ShowVatCalculation(float vatRate, decimal priceExcl)
{
    decimal vat = CalculateVat(vatRate, priceExcl);
    decimal priceIncl = priceExcl + vat;
    string summary = "Berekening BTW\n\n" +
        $"BTW excl.\t€ {priceExcl.ToString("0.00")}\n" +
        $"BTW tarief\t{vatRate} %\n" +
        $"BTW bedrag\t€ {vat.ToString("0.00")}\n" +
        $"BTW incl\t\t€ {priceIncl.ToString("0.00")}\n";
    return samenvatting;
}
```

9.2 CHECK JE EIGEN METHODE

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

```
private void BtnCheckControlDigits_Click(object sender, RoutedEventArgs e)
{
    CheckControlDigits();
}

void CheckControlDigits()
{
    string registrationNumber;
    int registrationNumber9Digits;
    int controlDigits;
    int expectedControlDigits;
    bool correctControlDigits;

    registrationNumber = txtRegistrationNumber.Text;

    registrationNumber9Digits = int.Parse(registrationNumber.Substring(0, 9));
    controlDigits = int.Parse(registrationNumber.Substring(9, 2));
    expectedControlDigits = 97 - (registrationNumber9Digits % 97);

    correctControlDigits = (controlDigits == expectedControlDigits);

    chkCorrectRegistrationNumber.IsChecked = correctControlDigits;
}
```

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

Bekijken we de methode CheckControlDigits echter in functie van hergebruik in andere omstandigheden, dan is er wel een probleem.

Er wordt informatie opgehaald uit `txtRegistrationNumber`. In een andere omgeving is het mogelijk dat deze Wpf control niet bestaat. We geven daarom beter de info die we ophalen mee als `parameter`. We moeten de info wel ophalen in de event handler methode die de andere eigen methode oproept.

```
private void BtnCheckControlDigits_Click(object sender, RoutedEventArgs e)
{
    string registrationNumber = txtRegistrationNumber.Text;
    CheckControlDigits(registrationNumber);
}
```

```
void CheckControlDigits(string registrationNumber)
{
    ...

    correctControlDigits = (controlDigits == expectedControlDigits);
    chkCorrectRegistrationNumber.IsChecked = correctControlDigits;
}
```

Een tweede probleem is dat we een property van een Wpf control (hier checkbox) willen aanpassen om feedback te geven naar de gebruiker.. Ook hier bestaat deze Wpf control enkel in de huidige applicatie, terwijl we de methode die controleert of het een geldig rijksregisternummer is wel graag zouden willen kunnen hergebruiken op andere plaatsen.

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

```
private void BtnCheckControlDigits_Click(object sender, RoutedEventArgs e)
{
    string registrationNumber = txtRegistrationNumber.Text;
    chkCorrectRegistrationNumber.IsChecked = CheckControlDigits(registrationNumber);
}
```

```
bool CheckControlDigits(string rijksNummer)
{
    ...


    correctControlDigits = (controlDigits == expectedControlDigits);
    return correctControlDigits;
}
```

Een ander voordeel hiervan is dat de gebruiker van de methode de geretourneerde info kan gebruiken zoals hij wil. Een control kan aangepast worden, de waarde kan verder gebruikt worden in het programma, een messagebox kan getoond worden... Het is eigenlijk niet de verantwoordelijkheid van de methode CheckControlDigits om te beslissen wat er met het antwoord gebeurt!



Code repository

De volledige broncode van dit hoofdstuk is te vinden op

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

10 METHODEN DOCUMENTEREN

Via de signatuur (= de naam, argumenten en het returntype) van een methode kun je al voor een groot stuk afleiden wat de bedoeling van de methode is. Het kan echter noodzakelijk zijn om nog wat meer uitleg te verschaffen. Dit kan bv. handig zijn als een collega op je code moet verder werken of als je na enige tijd je code moet aanpassen.

We kunnen een methode documenteren in het codebestand zelf, door drie maal op / te drukken. Visual Studio genereert dan automatisch al een kader om de documentatie te schrijven.

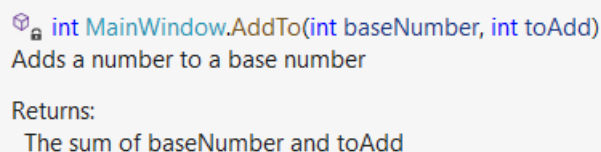
```
/// <summary>
///
/// </summary>
/// <param name="baseNumber"></param>
/// <param name="toAdd"></param>
/// <returns></returns>
int AddTo(int baseNumber, int toAdd)
{
    return baseNumber + toAdd;
}
```


Het is de bedoeling om op basis hiervan de nodige informatie te verschaffen.

```
/// <summary>
/// Adds a number to a base number
/// </summary>
/// <param name="baseNumber">Initial value</param>
/// <param name="toAdd">Number that needs to be added to the base number</param>
/// <returns>The sum of baseNumber and toAdd</returns>
int AddTo(int baseNumber, int toAdd)
{
    return baseNumber + toAdd;
}
```

Eens de documentatie toegevoegd is, wordt die ook zichtbaar via de Intellisense. Op het moment dat de muis op de gedocumenteerde methode staat, verschijnt de info over de summary en de return value.

```
int total = AddTo(7, 11);
```



 `int MainWindow.AddTo(int baseNumber, int toAdd)`
Adds a number to a base number

Returns:
The sum of baseNumber and toAdd

Anderszijds kunnen deze commentaren ook dienen om een xml-bestand aan te maken als basis voor een documentatie-site. Die kan dan verder ontwikkeld worden met tools als DocFX en Sandcastle.



Wil je meer te weten komen over documentatie aan de hand van XML-comments? Raadpleeg dan de documentatie van Microsoft: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/xml/doc/recommended-tags>.

