

C# Les 2

.TryParse(...)

In de vorige les hebben we gezien dat ons programma vastloopt wanneer het programma een getal probeert in te lezen en als de gebruiker per ongeluk (of moedwillig) iets anders dan een getal ingeeft.



Om dit op te lossen gaan we eerst controlleren of de ingegeven tekst wel kan worden omgezet naar een getal. We gebruiken hiervoor *float.TryParse()*.

Je geeft *TryParse(..., ...)* twee dingen mee, gescheiden door een komma:

1. De **string** die je wil proberen om te zetten naar een getal
2. Een **variabele van het type float** waarin je het **resultaat** van de omzetting wil plaatsen

TryParse geeft, net zoals *ReadLine*, een waarde terug die aangeeft of de omzetting gelukt is of niet. Die waarde is van het type **boolean**. Een boolean is een variabele waarin je enkel *false* of *true* in kan bewaren i.p.v. een getal of tekst.

Een aanroep van *TryParse* ziet er zo uit:

```
string strInput = Console.ReadLine();  
bool success = float.TryParse(strInput, out float flInput);
```

Wat is er nu gebeurd?

Met de eerste regel is er een tekst ingelezen in de variabele *strInput*. Dat kennen we al.

Met de tweede regel proberen we de variabele *strInput* om te zetten naar een float. Als dit lukt krijgt de variabele *flInput* de juiste waarde, anders krijgt deze de waarde 0.

Er is ook een variabele van het type **bool** (= boolean) gemaakt, met als naam *success* (deze naam mag je zelf kiezen). Als de omzetting gelukt is krijgt deze de waarde *true*, anders de waarde *false*.

We hebben nu een variabele die aangeeft of de omzetting gelukt is of niet. Je kan deze gebruiken in een if-statement:

```
string strInput = Console.ReadLine();  
bool success = float.TryParse(strInput, out float flInput);
```

```

if(success)
{
    Console.WriteLine("Het ingegeven getal was " + flInput);
}
else
{
    Console.WriteLine("Geef een correct getal in.");
}

```

Je hebt ook `int.TryParse` om tekst in gehele getallen om te zetten.

Meer informatie over `TryParse` vind je hier: <https://docs.microsoft.com/en-us/dotnet/api/system.single.parse?view=netcore-3.1>

Methods

We hebben in deze les en de vorige les de computer allerlei acties laten uitvoeren. Vaak gaat dit via dezelfde schrijfwijze:

```

Console.WriteLine("test");
Console.ReadLine();
Console.Clear();
float.Parse(str);
float.TryParse(str, out int intGetal);

```

Zoals je ziet hebben we steeds een woord gevolgd door een punt en dan terug een woord. Zo kan je acties **groeperen**. De acties die met de command line of console te maken hebben zijn gegroepeerd met het woord *Console*.

`Console` en `float` zijn **classes**. Een class groepeerd bijeenhorende acties: de methoden of **methods** van die class. `WriteLine`, `ReadLine` en `Clear` zijn methodes van de class `Console`. `Parse` en `TryParse` zijn methodes van de class `float` (of om helemaal correct te zijn van de class `Single` (`float` is een alias voor `Single`)).

Soms kan je informatie meegeven met een methode waarmee de methode iets zal doen. `WriteLine(...)` zal bijvoorbeeld de tekst in de command line tonen, `TryParse(...)` zal dan weer een tekst proberen om te zetten in een getal. De informatie die je meegeeft komt tussen de ronde haakjes te staan.

Als er lege ronde haakjes staan wil dit zeggen dat de methode geen informatie nodig heeft. De lege ronde haakjes wil zeggen: *doe iets*. De ronde haakjes met iets tussen wil zeggen: *doe iets met wat er tussen de ronde haakjes staat*.

```

Console.Clear(); // Lege ronde haakjes = doe iets
Console.WriteLine("Hallo"); // ronde haakjes = doe iets met de tekst "Hallo"

```

In sommige andere programmeertalen worden methodes **functies** genoemd.

Er zijn een heleboel reeds gemaakte methodes in C# waarvan je gebruik kan maken. Je kan ook zelf methodes gaan maken.

Een eigen methode maken

Een stukje code dat je meerdere keren wil uitvoeren gaat een goede programmeur niet copy-paste, maar wordt in een eigen methode geplaatst. Dit heeft verschillende voordelen: als de code moet worden veranderd hoeft dat maar op een plaats (= verbeterde onderhoudbaarheid). Ook wordt je code vaak korter en leesbaarder.

Een eigen methode maken doe je zo:

```
static void Main(string[] args)
{
    PrintDatum();
}

static void PrintDatum()
{
    Console.Write("Het is vandaag ");
    Console.Write(DateTime.Now.ToString("m"));
    Console.WriteLine(".");
}
```

In bovenstaand stukje code hebben we een eigen methode aangemaakt met als naam PrintDatum (het onderste accolade-blok). Je roept de methode op door de naam te vermelden en er ronde haakjes achter te plaatsen (dat zie je hierboven in de Main gebeuren). We gebruiken DateTime.Now.ToString om de huidige datum op te vragen.

Voor meer informatie zie hier: <https://docs.microsoft.com/en-us/dotnet/api/system.datetime.now?view=netcore-3.1>

Een method is een blokje code dat je kan **hergebruiken**. Een eenvoudige methode start met de woorden **static void** en heeft een **eigen naam**. Om de code van de methode uit te voeren vermeld je de naam met ronde haakjes eraan vast.

Method definition en signature

We hebben de methode zelf nog eens uit bovenstaand voorbeeld gelicht en hieronder geplaatst. Het geheel wat hier beneden staat is de **method definition** of **methode-definitie**.

PrintDatum() noemt men de **method signature** of **methode-signatuur**. Deze gaan we zo meteen nog verder uitbreiden door iets in de ronde haakjes te zetten.

PrintDatum is de **naam** van de methode.

```
static void PrintDatum()
{
    Console.Write("Het is vandaag ");
    Console.Write(DateTime.Now.ToString("m"));
    Console.WriteLine(".");
}
```

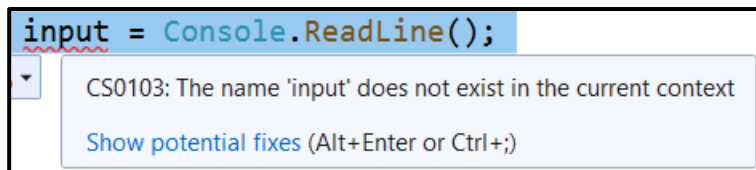
Een methode een resultaat laten terugsturen met return

Vaak is het handig om data te delen tussen je methode en de plaats waar deze wordt aangeroepen (in ons voorbeeld is dat de Main). Dit gaat misschien niet zoals je zou verwachten. Onderstaand voorbeeld werkt bijvoorbeeld niet 😞.

```
static void Main(string[] args)
{
    string input;
}

static void VraagInput()
{
    input = Console.ReadLine();
}
```

Visual Studio zegt "The name 'input' does not exist in the current context".



Dat heeft te maken met het feit dat **elke variabele die je maakt enkel zichtbaar is binnen de accolades waarin hij staat**. De variabele `input` bestaat enkel binnen de accolades van `Main`. Dit is een heel belangrijk principe in het programmeren: elke variabele heeft een eigen beperkt bereik of **scope**. Overal waar je accolades plaatst (ook bij de accolades van `if` of `while`) is de scope van de variabele beperkt. Dat is vaak heel handig omdat je zo niet altijd nieuwe namen moet gaan verzinnen over heel je project en variabelen elkaar ook niet ongewild kunnen overschrijven.

Elke variabele heeft een eigen bereik of **scope** waarbinnen hij zichtbaar is. Buiten zijn scope is de variabele onzichtbaar en bestaat de variabele eigenlijk niet. De scope wordt bepaald door de dichtstbijzijnde accolades waarin de variabele zit.

Maar hoe krijgen we het resultaat van `ReadLine` binnen onze methode dan in onze `Main`? We gaan een tunnel maken van de methode naar de `Main` met een **return**.



Onderstaande code werkt wel:

```
static void Main(string[] args)
{
    string input = VraagInput();
}

static string VraagInput()
{
    string strInput = Console.ReadLine();
    return strInput; // dit is de magische regel die iets terugstuurt!
}
```

Er zijn een paar zaken gewijzigd:

- De methode-aanroep VraagInput() **in de Main** zit nu in een toekenning
- Er staat een return statement **in de VraagInput** methode, deze stuurt het resultaat terug
- Voor de naam van onze methode staat niet meer het woordje *void* maar het woordje **string**. Dit woordje duidt aan welk data-type de methode terugstuurt. (Void wil zeggen dat een methode niets teruggeeft.)

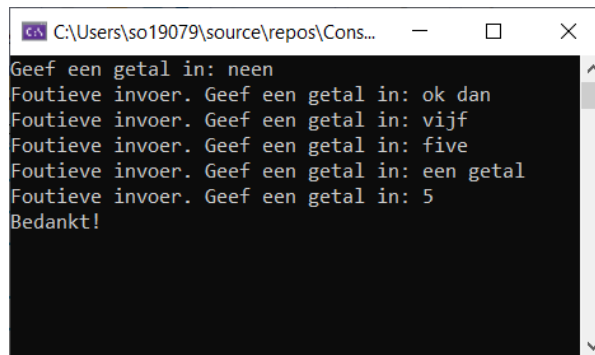
Merk trouwens op dat we de twee variabelen hernoemd heb naar *strInput* en *flInput* voor de duidelijkheid. We hadden de 2 variabelen ook alletwee *input* kunnen noemen, het gaat immers over 2 verschillende plaatsen met elk hun eigen omgeving of scope.

Met het **return**-statement sturen we iets terug vanuit een methode naar de plaats waar deze werd aangeroepen. Net als elke variabele krijgt ook het teruggestuurde resultaat een eigen data-type. Dit moet je vermelden tussen het woordje static en de naam van de methode.

Oefening: Enkel getallen

Kan je een methode maken die de gebruiker input blijft vragen tot er een getal wordt ingegeven?

```
static void Main(string[] args)
{
    float input = VraagInput();
    Console.Write("Bedankt!");
}
```



The screenshot shows a console window titled "C:\Users\so19079\source\repos\Cons...". The output is as follows:

```
Geef een getal in: neen
Foutieve invoer. Geef een getal in: ok dan
Foutieve invoer. Geef een getal in: vijf
Foutieve invoer. Geef een getal in: five
Foutieve invoer. Geef een getal in: een getal
Foutieve invoer. Geef een getal in: 5
Bedankt!
```

Hint:

- Maak gebruik van een TryParse, maar **niet** in combinatie met een if zoals het voorbeeld hierboven.
- Maak i.p.v. de if gebruik van een **while-statement** dat controleert of de success boolean **false** is. Dat doe je met het uitroepteken. Zo:

```
while(!success)
{
    // nog geen juiste invoer
}
```

Als je te lang vastzit vraag dan hulp aan een medecursist of aan de leraar.

Informatie doorgeven aan methods

We hebben net gezien hoe we iets kunnen **terugsturen naar de aanroeper** van een methode. Het kan ook andersom: iets **doorsturen naar een methode** bij het aanroepen. We hebben dit eigenlijk al eens gedaan bij de bestaande methodes in C# zoals WriteLine en TryParse.

```
Console.WriteLine("test"); // het woordje "test" wordt meegegeven

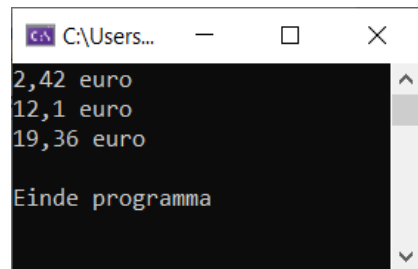
float.TryParse(str, out int intGetal); // str en intGetal worden meegegeven
```

We kunnen onze eigen methodes ook iets laten ontvangen.

```
static void Main(string[] args)
{
    float prijs = 2;
    PrintPrijsMetBtw(prijs);
    PrintPrijsMetBtw(10);
    PrintPrijsMetBtw(prijs * 8);
}

static void PrintPrijsMetBtw(float p)
{
    Console.Write(p * 1.21);
    Console.WriteLine(" euro");
}
```

Zoals je ziet krijgt de methode *PrintPrijsMetBtw* een waarde mee bij de aanroep door iets tussen de ronde haakjes te plaatsen. Binnen de methode wordt deze waarde gekoppeld aan de naam die wordt vermeld in de methode-signatuur. In de 2^{de} aanroep van *PrintPrijsMetBtw* hierboven wordt het getal 10 meegegeven. Binnen de methode is de waarde 10 gekoppeld aan de variabele *p*.



Het is ook mogelijk om meer dan 1 waarden mee te geven. Dit gaat zo:

```
static void Main(string[] args)
{
    float getal1 = 2;
    float som = MaakSom(getal1, 55 * 2);
    Console.WriteLine(som);
}

static float MaakSom(float x1, float x2)
{
    return x1 + x2;
}

// uitvoer: 112
```

In het voorbeeld hierboven worden *x1* en *x2* *parameters* genoemd. De methode *MaakSom* heeft dus 2 parameters (en geeft een waarde terug met *return*). De methode *PrintPrijsMetBtw* (hierboven) heeft maar 1 parameter: *p*.

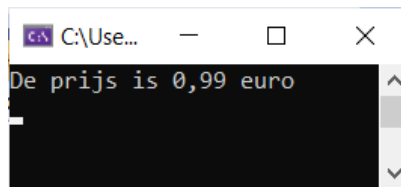
Als de methode wordt aangeroepen worden de parameters vervangen door concrete waarden: de *argumenten*. In bovenstaand voorbeeld wordt *MaakSom* opgeroepen met de argumenten *getal1* (= 2) en *55 x 2* (= 110).

Je kan informatie meesturen met een methode door 1 of meerdere **parameters** te voorzien. Parameters hebben ook weer een eigen data-type en worden in de methode-signatuur gezet.

Concateneren en string interpolation

Het samenplakken van stukken tekst wordt *concatenation* of *concateneren* genoemd. Je kan dat op verschillende manieren doen. Tot nu toe hebben we het in deze cursus met verschillende `Console.Write` statements opgelost (wat niet zo handig is):

```
float prijs = 0.99F; // de F suffix is nodig om een float te maken
Console.Write("De prijs is ");
Console.Write(prijs);
Console.WriteLine(" euro");
```



Iets handiger is het om het plus-teken te gebruiken, dat is de concatenatie-operator. Dan kan het in 1 regel:

```
float prijs = 0.99F; // de F suffix is nodig om een float te maken
Console.WriteLine("De prijs is " + prijs + " euro");
```

Merk op dat het plusteken 2 verschillende dingen doet al naargelang welke data-types je er rond zet. Zet je er stukjes tekst rond (strings) dan gaat het plusteken concateneren. Maar als je er getallen rond zet dan gaat het plusteken de getallen optellen.



Sinds C# 6 is het mogelijk om *string-interpolation* te gebruiken. Dat is de kortste, handigste en meest leesbare manier. We gaan vanaf nu deze manier gebruiken. Om string-interpolation in te schakelen zet je een dollar-teken voor aanhalingstekens van de string. Nu kan je met accolades variabele-namen of kleine berekeningen (of zelfs methode-aanroepen) in je tekst gaan plaatsen.

```
float prijs = 0.99F; // de F suffix is nodig om een float te maken
Console.WriteLine($"De prijs is {prijs} euro");
```


Misschien ga je online nog een andere manier tegenkomen om variabelen in een string te plakken (composite formatting), dat is een oudere en minder leesbare methode. Meer informatie over string interpolation en composite formatting vind je hier: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated>

Oefening: "Refactoring" oppervlakte berekenen

- A. Herschrijf de oefening van vorige keer, maar deze keer met methods. Gebruik de onderstaande gegeven Main methode en verplaats de verschillende berekeningen naar de juiste methodes:

```
static void Main(string[] args)
{
    string input = "";

    while (input != "q")
    {
        input = ToonMenuEnVraagInput();

        if(input == "1")
        {
            OppervlakteRechthoek();
        }
        else if(input == "2")
        {
            OppervlakteDriehoek();
        }
        else if(input == "3")
        {
            OppervlakteCirkel();
        }
        else if(input == "q")
        {
            Console.WriteLine("Tot de volgende keer!");
        }
        else
        {
            Console.WriteLine("Ongeldige keuze :(");
            Console.ReadLine();
        }
    }
}
```

- B. Zorg er voor dat de gebruiker enkel getallen kan ingeven bij de berekening van de oppervlaktes zodat het programma niet kan crashen. Kan je dit op te lossen zonder code te copy-pasten? (Hint: Je kan ook methodes vanuit andere methodes aanroepen.)

Refactoring is het herschrijven van code om deze duidelijker of efficiënter te maken zonder dat de werking verandert.

Existing Code



Your Plan To Refactor



Your Actual Refactor

