Methoden

Zie Scherp Scherper - Hoofdstuk 7

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- uitleggen waarom **methoden** code herbruikbaar en leesbaar maken
- zelf een methode schrijven met een **returntype** en `return`
- **parameters** doorgeven en het verschil met *by value* begrijpen
- methoden **nesten** en oneindige methode-lussen herkennen
- **named** en **optionele** parameters en **overloading** gebruiken
- met `///`-commentaar en **IntelliSense** efficiënter werken

Luiheid als deugd

> *“I will always choose a lazy person to do a difficult job, because he will find an easy way to do it.”* - Bill Gates

We jagen op **zo weinig mogelijk code met zoveel mogelijk rendement**.
Loops waren een eerste stap, methoden de volgende.

> **Tip**
>
> Bestond `Console.WriteLine` niet, dan moest je telkens tientallen lijnen interne code overtypen. Wees blij dat methoden bestaan.

Wat is een methode?

- Een **blok code** dat een specifieke taak uitvoert
- Herkenbaar aan de **ronde haakjes** achteraan, al dan niet met parameters
- Je gebruikt ze al sinds les 1: elke `Console.WriteLine()` is een methode-aanroep

> Nieuw: we gaan nu **zelf** methoden definiëren, niet enkel bestaande gebruiken.

Methode-syntax

```
static returntype MethodeNaam(optioneel_parameters)
{
    //code van de methode
}
```

De eerste lijn is de **methode-signatuur**: returntype, naam en parameters.

> **Tip**
>
> Vergeet `static` voorlopig: beschouw het als een verplicht **toverwoord**. Waarom het er staat, zie je in hoofdstuk 11.

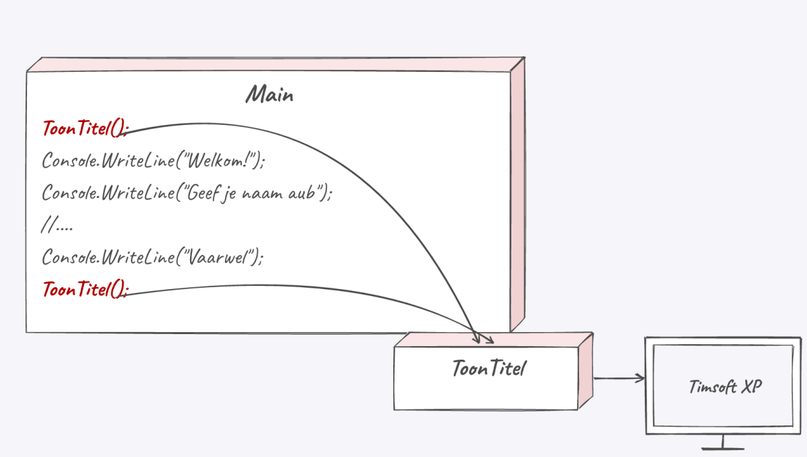
Een eenvoudige methode

```
1  static void ToonTitel()
2  {
3      Console.WriteLine("Timsoft XP");
4  }
```

Aanroepen, eender waar:

```
1  ToonTitel();
```

Verandert de titel ooit? Eén plek aanpassen volstaat.



De flow van de aanroep.

> **Opmerking**
>
> Ook **`Main`** is gewoon een methode: het startpunt van je programma.

Returntype en return

`void` betekent dat een methode **niets** teruggeeft. Vaak wil je wel iets terug:

```
1  static string VerkrijgAuteurNaam()
2  {
3      return "Tim Dams";
4  }
```

- **`return`** geeft het resultaat terug en verlaat de methode meteen
- Vang het resultaat op in een variabele van **hetzelfde type**:

```
1  string myName = VerkrijgAuteurNaam();
2  Console.WriteLine($"Auteur: {VerkrijgAuteurNaam()}");
```

Een uitgewerkte methode

```
static int FaculteitVan5()
{
    int resultaat = 1;
    for (int i = 1; i <= 5; i++)
    {
        resultaat *= i;
    }
    return resultaat;
}
```

```
Console.WriteLine($"Faculteit van 5 is {FaculteitVan5()}");  // 120
```

void: niets opvangen

```
static void ToonVersie()
{
    Console.WriteLine("Dit is versie 8.31");
}
```

Een `void`-methode geeft niets terug. Deze twee **mogen dus niet**:

```
string result = ToonVersie();      // FOUT
Console.WriteLine(ToonVersie());   // FOUT
```

Not all code paths return a value

`return` mag overal, en sluit de methode meteen af. Maar elke weg door je methode moet wél iets teruggeven:

```
1 switch (r.Next(0, 4))
2 {
3     case 0: return "noord";
4     // ...
5 }
6 return "onbekend"; // nooit bereikt, tóch vereist
```

> ⚠ **Waarschuwing**
>
> *"Not all code paths return a value"*: ergens kan je het einde bereiken zonder `return`.

Parameters doorgeven

Veralgemeen `FaculteitVan5` met een **parameter**:

```
static int BerekenFaculteit(int grens)
{
    int resultaat = 1;
    for (int i = 1; i <= grens; i++)
        resultaat *= i;
    return resultaat;
}
```

```
int resultaat = BerekenFaculteit(5);  // 120
```

> **Tip**
>
> De parameternaam (`grens`) hoeft niet gelijk te zijn aan de meegegeven variabele. Enkel de **waarde** telt.

By value

Parameters gaan standaard **by value**: de methode krijgt een **kopie**.

```
static void JaartjeOuder(int leeftijd)
{
    leeftijd++;
    Console.WriteLine($"Je bent {leeftijd} jaar geworden.");
}
```

```
Je bent 40 jaar.
Je bent 41 jaar geworden.
Je bent 40 jaar.   <- origineel ongewijzigd
```

> **Opmerking**
>
> *By reference* (de echte variabele aanpassen) komt aan bod vanaf het hoofdstuk over arrays.

Volgorde van parameters

De eerste meegegeven waarde gaat naar de eerste parameter, enz.

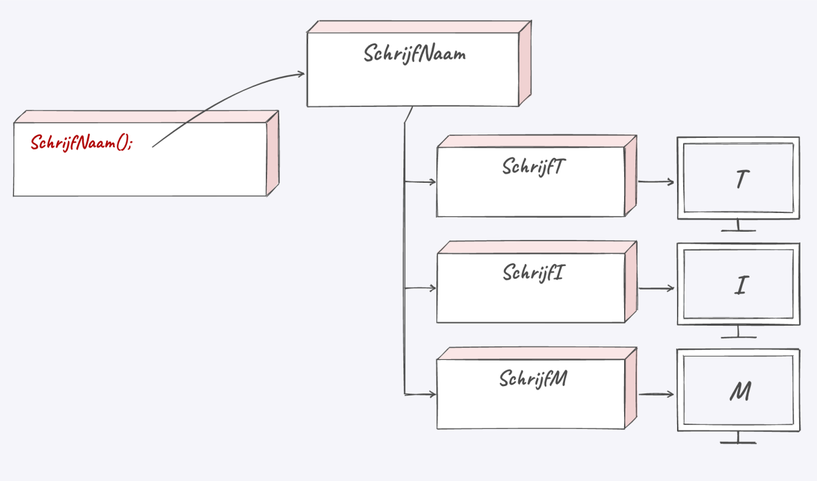
```
static void ToonInfo(string name, int age)
{
    Console.WriteLine($"{name} is {age} jaar");
}
```

```
ToonInfo("Tim", 37);   // ok
ToonInfo(37, "Tim");   // FOUT: compileert niet
```

Methoden nesten

Een methode mag andere methoden aanroepen:

```
1  static void SchrijfNaam()
2  {
3      SchrijfT();
4      SchrijfI();
5      SchrijfM();
6  }
```



Aanroepen van methoden uit methoden.

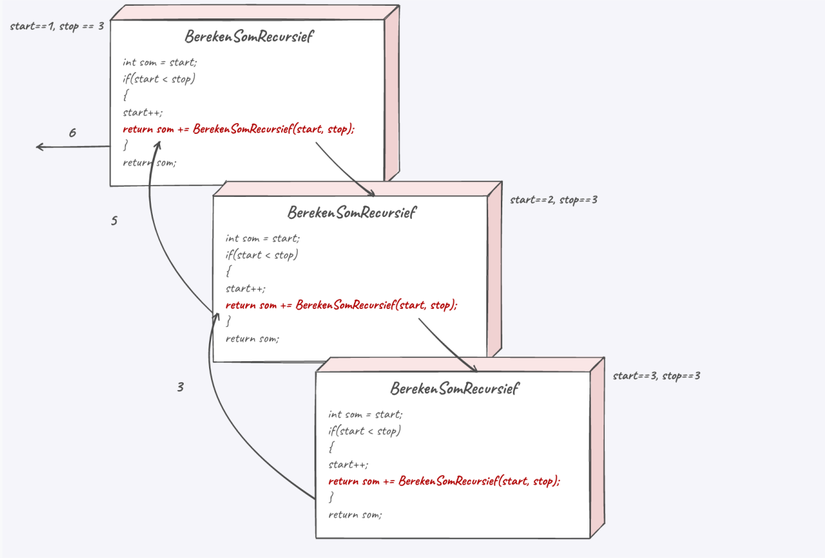
Resultaat: “Tim”.

> ⚠ **Waarschuwing**
>
> Een methode die **zichzelf** ongecontroleerd aanroept, loopt oneindig en crasht als het geheugen op is.

Recursie (kort)

Recursie = een methode die zichzelf aanroept, maar **wel stopt**:

```
1  static int BerekenSom(int start, int stop)
2  {
3      int som = start;
4      if (start < stop)
5          return som += BerekenSom(start + 1, stop)
6      return som;
7  }
```



Flow van de recursie.

Een geavanceerd concept; we komen het pas in hoofdstuk 18 kort terug tegen.

Lokale functies: niet doen

Sinds C# 7 mag een methode in een andere methode staan, maar als beginner doe je dat **niet**:

```
1  static void Main(string[] args)
2  {
3      TimVindtDitNietLeuk();
4
5      static void TimVindtDitNietLeuk() // NIET DOEN
6      {
7          Console.WriteLine("Doe dit niet!");
8      }
9  }
```

Je kan zo'n lokale functie nergens anders aanroepen. Zet je methoden gewoon **naast** `Main`.

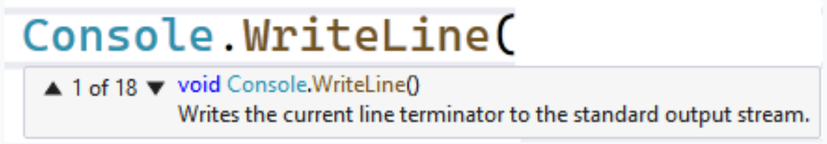
Documentatie met ///

```
/// <summary>
/// Berekent de macht van een getal.
/// </summary>
/// <param name="grondtal">Het getal dat je verheft</param>
/// <param name="exponent">De exponent</param>
static int Macht(int grondtal, int exponent) { /* ... */ }
```

Typ `///` boven een methode en VS maakt het skelet. Bij elke aanroep krijg je dan **tooltips** met uitleg.

Bestaande methoden en IntelliSense

- Typ een methode en stop na het **eerste haakje**: IntelliSense toont alle manieren (de *overloads*)
- Cursor op een methode + `F1` opent de online documentatie
- Typ `Console.` en zie alles wat die bibliotheek kan



IntelliSense bevat een schat aan info.

Herbruikbare input

Komt deze constructie vaak voor?

```
1 Console.WriteLine("Geef leeftijd");
2 int leeftijd = int.Parse(Console.ReadLine());
```

Verhuis ze naar een methode:

```
1 static int VraagInt(string zin)
2 {
3     Console.WriteLine(zin);
4     return int.Parse(Console.ReadLine());
5 }
6
7 int leeftijd = VraagInt("Geef leeftijd");
```

Je `Main` blijft net en leesbaar.

Named parameters

Geef expliciet aan welke waarde bij welke parameter hoort:

```
1 static void PrintDetails(string seller, int orderNum, string product) { }
2
3 PrintDetails(orderNum: 31, product: "Red Mug", seller: "Gift Shop");
```

- Met **named parameters** maakt de volgorde niet uit
- Meng je named en gewone parameters, dan **moet** je wél de signatuur-volgorde respecteren

Optionele parameters

Geef een **default waarde** mee; die parameters staan **achteraan**:

```
1  static void BookFile(int required, string optName = "unknown", int age = 10)
```

```
1  BookFile(15, "tim", 25);
2  BookFile(20, "dams");  // age = 10
3  BookFile(35);          // optName = "unknown", age = 10
```

> **Tip**
>
> Wil je enkel `age` zetten en `optName` overslaan? Gebruik dan een named parameter: `BookFile(3, age: 4)`.

Method overloading

Dezelfde naam en returntype, maar **andere parameters**. De compiler kiest de juiste:

```
static int BerekenOpp(int lengte, int breedte)
{
    return lengte * breedte;
}
static int BerekenOpp(int straal)
{
    return (int)(Math.PI * straal * straal);
}
```

```
BerekenOpp(5, 6);  // rechthoek
BerekenOpp(7);     // cirkel
```

> **Opmerking**
>
> Twijfelt de compiler tussen types, dan kiest hij via de **betterness rule** de best passende. Dat is verdiepingsstof voor de gevorderden.

Methoden debuggen: step in

Herinner je de debugknoppen uit hoofdstuk 4? Eén lieten we toen liggen: **step in**.

- **Step over** springt over een methode-aanroep heen
- **Step in** gaat **in** je eigen methode en laat je daar lijn per lijn stappen

Simpel, maar oefen het toch een paar keer.

Conclusie

- Een **methode** is een herbruikbaar codeblok: minder fouten, betere leesbaarheid
- `void` geeft niets terug; anders gebruik je `return` met het juiste type
- Parameters gaan **by value** (een kopie); de **volgorde** telt
- **Named** en **optionele** parameters en **overloading** maken je methoden flexibel
- Met `///` en **IntelliSense** werk je sneller en met betere documentatie

> **Tip**
>
> Volgende halte: **arrays**, om vele waarden samen in één variabele bij te houden.

Meer weten

Kennisclips

- Methoden
- Goede methoden schrijven
- Bestaande bibliotheken
- Geavanceerde methodetechnieken
- Fun with methods

Oefeningen H7 · Quizlet flashcards

Zie verder: andere talen

Dezelfde concepten uit dit hoofdstuk, maar dan in andere talen. Handig om later code in Python of JavaScript te leren lezen.

Zie verder: methode-signatuur

Python doet dit zo:

```
1 def tel_op(a, b):
2     return a + b
```

Geen returntype, geen types voor parameters. Dat scheelt typewerk, maar Python merkt een verkeerd type pas tijdens het uitvoeren.

Zie verder: bibliotheken importeren

In C# haal je bibliotheken binnen met `using`. **Python** doet dit met `import`:

```
1 import math
2
3 print(math.sqrt(16))
```

Hetzelfde idee als bij C#: je geeft aan welke bibliotheek je wilt en spreekt de methoden aan via de naam (`math.sqrt`, zoals `Math.Sqrt`).

Zie verder: overloading

Python kent geen overloading: een tweede functie met dezelfde naam overschrijft de eerste. Pythonisten gebruiken default-parameters:

```
def bereken_opp(lengte, breedte=None):
    if breedte is None:
        return int(3.14159 * lengte * lengte)  # cirkel
    return lengte * breedte                     # rechthoek
```

C# laat de compiler op voorhand de juiste versie kiezen. In Python schrijf je die keuze zelf met een `if`.

Samenvattende poster

Een handige samenvattende poster (Opgelet: gemaakt m.b.v. ChatGPT Image Gen 2).