C# Essentials - Methodes

Inhoudsopgave

1. Introductie

1.1. Wat is een method?

Een **method** is een blokje herbruikbare code dat een bepaalde taak uitvoert. Je kan een method oproepen telkens je die taak wil uitvoeren. Je kan zelf parameters meegeven en de method kan eventueel ook een resultaat teruggeven.

Voorbeelden van method-namen: `PrintInvoice()`, `CalculateTotal()`, `ResetScore()`.

1.2. Waarom gebruik je methods?

Methods maken je code:

- overzichtelijker en beter gestructureerd
- herbruikbaar
- makkelijker aanpasbaar
- eenvoudiger om in groep aan te werken

In plaats van dezelfde code telkens opnieuw te schrijven, plaats je die in een method die je kan hergebruiken.

1.3. Syntax

Een method heeft de volgende algemene vorm:

```
<access modifier> <returntype> <Name>(<parameter(s)>)
{
    // instructies
}
```

Voorbeeld:

```
public void Greet()
{
    Console.WriteLine("Hello!");
}
```

Of met returnwaarde en parameters:

```
public int Add(int a, int b)
{
    return a + b;
}
```

1.3.1. Toegankelijkheid

Met een access modifier bepaal je **wie de method mag oproepen**. Voorlopig gebruiken we hier `public`, in de module "Klassen" gaan we hier dieper op in.

1.3.2. Returntype

Een method kan een resultaat teruggeven. In dat geval gebruik je `return`, en geef je het returntype aan. Als je niets wil teruggeven, gebruik je `void`.

1.3.3. Parameters

Parameters zijn gegevens die je meegeeft aan de method. Je vermeldt telkens het type én de naam.

```
public void SayHello(string name)
{
    Console.WriteLine($"Hello, {name}!");
}
```

1.4. Methods ??? Functions ??? Event Handlers

In C# worden er verschillende termen gebruikt voor stukken code die iets **uitvoeren**, maar er is een klein verschil in gebruik:

- **Method**: Een algemene term voor een blokje code dat een taak uitvoert.
- **Function**: Wordt meestal gebruikt als een method **een waarde teruggeeft** via `return`. Alle functions zijn methods, maar niet elke method is een function.
- **Event handler**: Een speciale method die **automatisch wordt uitgevoerd wanneer een event plaatsvindt**, zoals een knop waarop geklikt wordt.

Omdat er geen events kunnen optreden in een console-applicatie wordt dit type method pas later behandeld

Dus: het zijn allemaal "methods", maar afhankelijk van het doel of de context worden ze soms anders genoemd.

1.5. Samenvatting

- Een method is een herbruikbaar blokje code dat iets uitvoert.
- Je kan parameters meegeven en eventueel een resultaat teruggeven.
- Gebruik `void` als de method niets teruggeeft.
- De toegangsmodifier bepaalt wie de method mag gebruiken (bv. `public`, `private`).

2. Static

2.1. Wat is een static method?

Een **static method** behoort niet tot een object, maar tot de **klasse zelf**. In een consoleapplicatie zonder klassen betekent dit dat je de method gewoon rechtstreeks kan oproepen — zonder eerst iets aan te maken met `new`.

Je herkent een static method aan het sleutelwoord `static`.

```
public static void ShowWelcome()
{
    Console.WriteLine("Welcome!");
}
```

Je kan deze methode direct oproepen in `Main()`:

Program.cs

```
ShowWelcome();
```

2.2. Waarom static methods gebruiken?

Je gebruikt static methods wanneer:

- de methode **geen gegevens van een object** nodig heeft
- je een algemene actie wil uitvoeren
- je werkt in een consoleapplicatie zonder objecten

Bijvoorbeeld: berekeningen, tekst afdrukken of het verwerken van input.

2.3. Voorbeelden

2.3.1. Methode zonder parameters

```
public static void ShowInstructions()
{
    Console.WriteLine("Press any key to continue...");
}
```

2.3.2. Methode met parameters

```
private static int Multiply(int a, int b)
{
    return a * b;
}
```

Gebruik:

```
int result = Multiply(3, 5);
Console.WriteLine(result); // Outputs: 15
```

2.4. Vergelijking met niet-static methods

Niet-static methods horen bij een object of instantie. Dat doen we later bij klassen. In een consoleapplicatie gebruik je voorlopig **alle methoden als static**, omdat er geen objecten zijn.

Later leer je:

- static → aanroepen via klassenaam
- niet-static → aanroepen via object

2.5. Samenvatting

- Een static method gebruik je zonder een object aan te maken.
- Je schrijft `static` vóór de returntype van de method.
- Alle methoden in een consoleapplicatie zonder klassen zijn static.
- Gebruik static methods voor algemene acties zoals berekeningen of weergave.

3. Returntype

3.1. Wat is een returntype?

Een **returntype** bepaalt welk soort waarde een method teruggeeft aan de plek waar je ze oproept. Je vermeldt dit altijd vóór de naam van de methode.

```
static int GetNumber()
{
    return 42;
}
```

Hier is `int` het returntype. Dat betekent dat de methode een geheel getal teruggeeft.

3.2. Waarom een returntype gebruiken?

Soms wil je een methode maken die iets berekent of bepaalt en het resultaat moet kunnen doorgeven aan de rest van het programma.

Bijvoorbeeld:

- Een getal verdubbelen
- Een tekst samenstellen
- Controleren of een getal even is

3.3. Een waarde teruggeven met `return`

Je gebruikt het sleutelwoord `return` om een waarde terug te geven.

```
static int DoubleNumber(int number)
{
    return number * 2;
}
```

3.4. Voorbeelden van returntypes

Type	Omschrijving
int	Geheel getal (bv. 5, 200, -3)
double	Kommagetal (bv. 3.14, -7.5)
string	Tekst (bv. "Hallo")
bool	`true` of `false`
void	Geeft niets terug

3.5. Gebruik van de returnwaarde

Je kan de waarde die een methode teruggeeft op verschillende manieren gebruiken:

- Opslaan in een variabele:

```
int resultaat = DoubleNumber(10);
```

- Onmiddellijk tonen:

```
Console.WriteLine(DoubleNumber(5));
```

- Gebruiken in een voorwaarde:

```
if (IsEven(6))
{
    Console.WriteLine("Het getal is even.");
}
```

3.6. Wat met `void`?

Soms heb je geen resultaat nodig. Dan gebruik je `void` als returntype.

```
static void PrintWelcome()
{
    Console.WriteLine("Welkom!");
}
```

Je gebruikt dan geen `return`-waarde.

3.7. Methode vroegtijdig verlaten

Ook bij een `void`-methode kan je het sleutelwoord `return` gebruiken zonder een waarde. Hiermee stop je de uitvoering van de methode op dat moment.

Voorbeeld:

```
static void ToonBericht(int leeftijd)
{
    if (leeftijd < 0)
    {
        Console.WriteLine("Ongeldige leeftijd.");
        return;
    }

    Console.WriteLine($"Je bent {leeftijd} jaar oud.");
}
```

Zodra `return;` wordt uitgevoerd, stopt de methode onmiddellijk en de rest van de code **in de methode** wordt niet meer uitgevoerd.

3.8. Samenvatting

- Het returntype bepaalt welk soort waarde een methode teruggeeft.
- Je gebruikt `return` om die waarde terug te geven.
- Veelgebruikte returntypes zijn `int`, `double`, `string` en `bool`.
- Gebruik `void` als je geen waarde wil teruggeven.
- De returnwaarde kan je opslaan, tonen of gebruiken in een test.

4. Parameters

4.1. Wat zijn parameters?

Parameters zijn **gegevens die je aan een methode doorgeeft** wanneer je ze oproept. Ze staan tussen de haakjes in de methodedefinitie.

```
static void SayHello(string name)
{
    Console.WriteLine($"Hallo, {name}!");
}
```

In dit voorbeeld is `name` een parameter van het type `string`.

4.2. Waarom parameters gebruiken?

Je gebruikt parameters om **dezelfde methode te kunnen hergebruiken met verschillende gegevens**. Zo moet je niet telkens een nieuwe methode schrijven.

Voorbeeld:

```
SayHello("Luna");
SayHello("Jonas");
```

De methode `SayHello()` wordt twee keer opgeroepen, telkens met een andere waarde.

4.3. Syntax van parameters

Je moet altijd het **type** en de **naam** van elke parameter vermelden:

```
static void ShowAge(int age)
{
    Console.WriteLine($"Je bent {age} jaar oud.");
}
```

De waarde die je doorgeeft bij het oproepen, wordt opgeslagen in de parameter.

4.4. Meerdere parameters

Je kan meerdere parameters combineren door ze met komma's te scheiden:

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

Gebruik:

```
ShowPersonData("Emma", 22);
```

De volgorde van de argumenten moet overeenkomen met de volgorde van de parameters.

4.5. Standaardwaarden voor parameters

Je kan een parameter een **standaardwaarde** geven. Als je bij het oproepen geen waarde meegeeft, wordt de standaard gebruikt.

```
static void Greet(string name = "gast")
{
    Console.WriteLine($"Welkom, {name}!");
}
```

Gebruik:

```
Greet();            // Toont: Welkom, gast!
Greet("Timo");      // Toont: Welkom, Timo!
```

- parameters met standaardwaarden moeten **achteraan** staan
- je mag ze combineren met gewone parameters

4.6. Samenvatting

- Parameters zijn invoergegevens die je meegeeft aan een methode.
- Je vermeldt altijd het type en de naam van de parameter.
- Meerdere parameters scheid je met een komma.
- Met standaardwaarden kan je optionele parameters maken.
- De volgorde van de argumenten bij het oproepen moet overeenkomen met de definitie.

5. Overloading

5.1. Wat is method overloading?

Method overloading betekent dat je **meerdere methoden met dezelfde naam** mag definiëren, zolang ze **verschillende parameters** hebben.

De compiler kiest automatisch welke versie van de methode moet uitgevoerd worden, op basis van het aantal en type van de argumenten die je meegeeft.

5.2. Waarom overloading gebruiken?

Overloading is handig omdat:

- je dezelfde actie met verschillende gegevens wil uitvoeren
- je je methode een logische en herkenbare naam wil geven
- je geen aparte methoden met namen als `AddInt`, `AddDouble` hoeft te maken

5.3. Voorbeeld

```
static void ShowMessage()
{
    Console.WriteLine("Hello!");
}

static void ShowMessage(string name)
{
    Console.WriteLine($"Hello, {name}!");
}
```

Gebruik:

```
ShowMessage();           // Output: Hello!
ShowMessage("Luna");     // Output: Hello, Luna!
```

De compiler weet op basis van de argumenten welke versie hij moet gebruiken.

5.4. Verschil in parameters vereist

Je mag enkel overloaden als de parameterlijst verschilt:

```
static void Multiply(int number)
{
    Console.WriteLine(number * 2);
}

static void Multiply(int number, int factor)
{
```

```
    Console.WriteLine(number * factor);
}
```

Dit is geldig omdat het aantal parameters verschilt.

 Dit is **niet geldig**:

```
// FOUT: dezelfde parameterlijst
static void PrintText(string text) { }
static void PrintText(string message) { }
```

5.5. Combinatie met standaardwaarden

Let op: standaardwaarden kunnen conflicteren met overloading:

```
static void Log(string message) { }

static void Log(string message, string level = "INFO") { }
// FOUT: bij oproep Log("Test") is het onduidelijk welke methode moet gebruikt
worden
```

Probeer overloading en standaardwaarden niet te mengen, tenzij het verschil duidelijk blijft.

5.6. Samenvatting

- Bij method overloading definieer je meerdere methoden met dezelfde naam.
- De parameterlijst moet verschillen in aantal of type.
- De compiler kiest de juiste methode op basis van de argumenten.
- Overloading maakt je code leesbaarder en consistenter.

6. By Value vs By Reference

Dit hoofdstuk behoort niet tot de verplichte leerstof van deze cursus.

Het verschil tussen `by value` en `by reference` wordt in C# gebruikt om te bepalen hoe gegevens worden doorgegeven aan methodes. Hoewel dit een belangrijk concept is in de programmeertaal zelf, komt het in de praktijk zelden voor dat je expliciet `ref` of `out` gebruikt in moderne toepassingen.

Daarom behandelen we dit niet in de verplichte leerstof. Toch bieden we dit hoofdstuk aan voor wie nieuwsgierig is naar hoe dit werkt onder de motorkap van C# of wie graag wat extra inzicht wil in hoe parameters precies worden doorgegeven aan methodes.

Je moet dit dus niet kennen voor oefeningen, labo's of examens. **Wel moet je weten dat, wanneer je een waarde als parameter doorgeeft aan een method, je eigenlijk een kopie van die waarde doorgeeft.**

6.1. By value (standaard)

In C# worden parameters standaard doorgegeven **by value**. Dat betekent dat je **een kopie** van de waarde doorgeeft aan de methode.

Veranderingen die je binnen de methode maakt, hebben **geen invloed op de originele variabele**.

Voorbeeld:

```
static void Increase(int number)
{
    number = number + 1;
    Console.WriteLine($"Inside method: {number}");
}

int myNumber = 5;
Increase(myNumber);
Console.WriteLine($"Outside method: {myNumber}"); // still 5
```

De waarde van `myNumber` buiten de methode verandert niet.

6.2. By reference met `ref`

Met `ref` geef je de variabele **by reference** door. Dat betekent dat de methode **werkt met de originele variabele**.

Voorbeeld:

```
static void Increase(ref int number)
{
    number = number + 1;
    Console.WriteLine($"Inside method: {number}");
}

int myNumber = 5;
Increase(ref myNumber);
Console.WriteLine($"Outside method: {myNumber}"); // now 6
```

Let op:

- de variabele moet al een waarde hebben voordat je ze doorgeeft met `ref`
- je moet `ref` schrijven op **twee plaatsen**: bij de methodedefinitie én bij het oproepen

6.3. By reference met `out` ⁉️⁉️

`out` is vergelijkbaar met `ref`, maar je gebruikt het als je **een waarde wil teruggeven via een parameter**. De variabele **moet niet vooraf geïnitialiseerd zijn**.

Voorbeeld:

```
static void CalculateSquare(int number, out int result)
{
    result = number * number;
}

int square;
CalculateSquare(7, out square);
Console.WriteLine($"The square is: {square}");
```

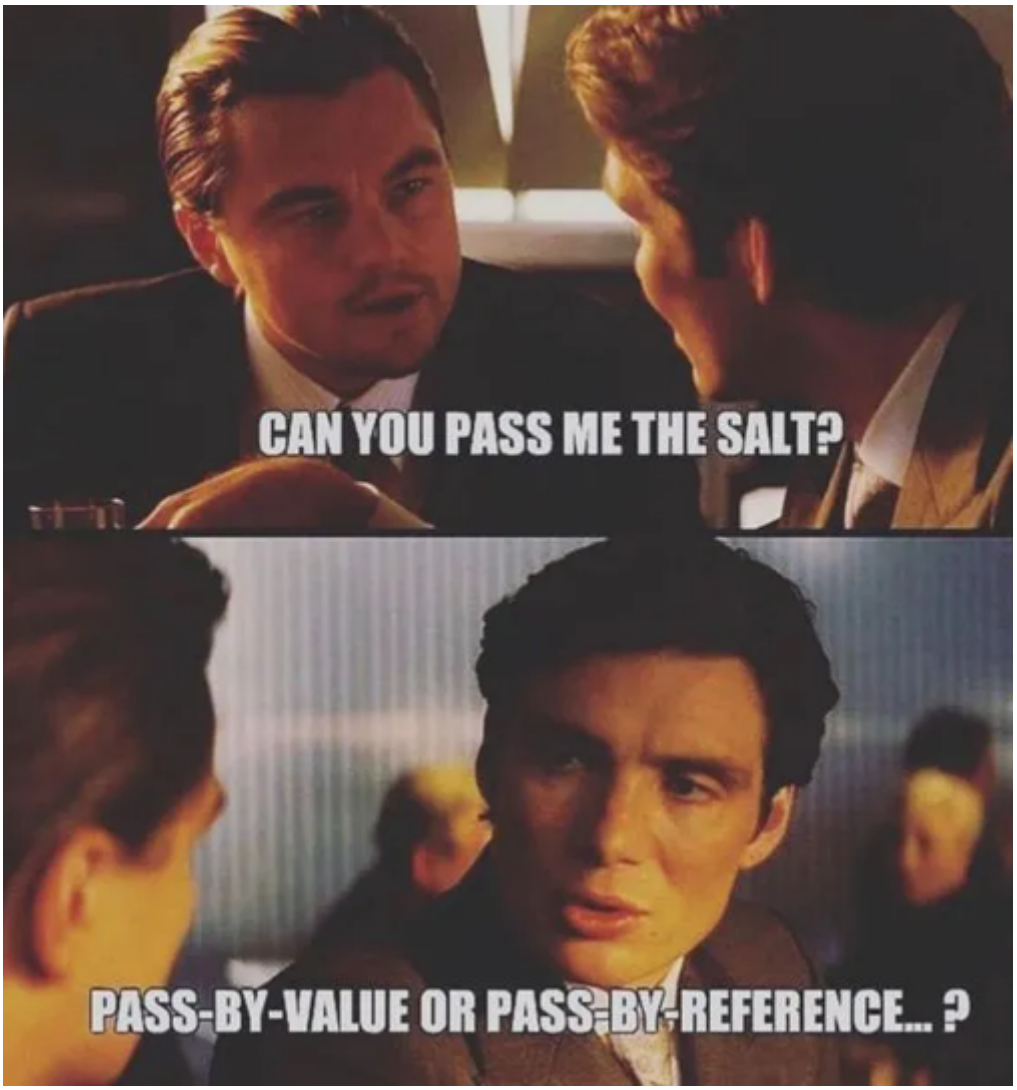
- je **moet** de out-parameter **een waarde geven binnen de methode**
- je gebruikt `out` zowel in de definitie als bij het oproepen

6.4. Vergelijking ⁉️

Type	In methode	Effect op originele variabele
by value	werkt met een kopie	geen effect
ref	werkt met origineel	verandert buiten de methode
out	verwacht output	vult een variabele in

6.5. Samenvatting

- Parameters worden standaard doorgegeven **by value** (kopie).
- Met `ref` geef je de variabele door **by reference** en kan je ze wijzigen.
- Met `out` geef je een lege variabele door en de methode vult ze in.
- Gebruik `ref` als je een bestaande waarde wil aanpassen.
- Gebruik `out` als je een extra resultaat wil teruggeven.



7. Scope van variabelen

7.1. Scope in een methode

Elke methode vormt een eigen **scope-blok**. Variabelen die je binnen een methode aanmaakt, bestaan enkel daar.

```
static void Main(string[] args)
{
    int number = 5;
    ShowNumber();
    Console.WriteLine(number); // OK
}

static void ShowNumber()
{
    int number = 10;
    Console.WriteLine(number); // OK
}
```

Dit zijn **twee aparte variabelen** `number`, elk met hun eigen scope. De `number` in `ShowNumber` bestaat **alleen daarbinnen**.

7.2. Variabele delen tussen methodes? Niet zomaar!

```
static void Main(string[] args)
{
    int score = 100;
    ShowScore();
}

static void ShowScore()
{
    Console.WriteLine(score); // FOUT! score is hier niet gekend
}
```

De variabele `score` werd aangemaakt in `Main`, maar is **niet beschikbaar** in `ShowScore`. Elke methode ziet enkel zijn eigen variabelen.

7.3. Oplossing: gebruik parameters

Als je een waarde wil gebruiken in een andere methode, moet je die **doorgeven via parameters**.

```
static void Main(string[] args)
{
    int score = 100;
    ShowScore(score); // geef score mee als argument
}
```

```
static void ShowScore(int value)
{
    Console.WriteLine($"Je score is: {value}");
}
```

In dit voorbeeld geef je `score` door als argument aan de methode `ShowScore`. De parameter `value` krijgt dan die waarde.

7.4. Samenvatting

Regel	Uitleg
Elke methode heeft zijn eigen scope	Variabelen bestaan enkel binnen hun eigen methode
Je kan een variabele niet gebruiken in een andere methode	Tenzij je ze meegeeft via parameters
Wil je info delen tussen methodes?	Gebruik parameters (of later: instance-variabelen in een klasse)

7.5. Oefening

Wat gebeurt er wanneer je deze code uitvoert?

```
static void Main(string[] args)
{
    string name = "Alice";
    Greet();
}

static void Greet()
{
    Console.WriteLine($"Hallo {name}!");
}
```

Oplossing

Je krijgt een foutmelding: `name` is gedeclareerd in `Main` en is niet gekend in `Greet()`. Je moet de naam als parameter doorgeven als je ze daar wil gebruiken.