

1.0.0

## H7: Methoden

# Methoden intro



## Methoden

Veel code die we hebben geschreven wordt meerdere keren, al dan niet op verschillende plaatsen, gebruikt. Dit verhoogt natuurlijk de foutgevoeligheid. Door het gebruik van methodes kunnen we de foutgevoeligheid van de code verlagen omdat de code maar op 1 plek staat én maar 1 keer dient geschreven te worden. Ook de leesbaarheid en dus onderhoudbaarheid van de code wordt verhoogd wanneer we methoden gebruiken om verschillende deeltaken van elkaar te scheiden.

### Wat is een methode

Een methode (bijna hetzelfde als een "functie" in andere programmeertalen) genoemd, is in C# een stuk code ('block') bestaande uit een 0, 1 of meerdere statements. Het is eigenlijk een klein stappenplan voor een onderdeel van je totale programma.

Zoals bij elk stappenplan is er een verschil tussen het vastleggen van de stappen en het uitvoeren. Het vastleggen van de stappen noemen we **definiëren** van de methode. Vergelijk met het tekenen van een flowchart voor een bepaalde taak, maar dan in code: de **definitie** van de methode stemt overeen met de flowchart die de deeltaak beschrijft.

Het uitvoeren van de stappen, dus het volgen van de flowchart, noemen we het **oproepen** of **uitvoeren** van de methode. De code die dit proces in gang zet noemen we een **oproep** (in het Engels: **call**) van de methode.

Je hebt intussen al vaak (ingebouwde) methodes opgeroepen, waaronder:

- de `Main` methode van de klasse `Program` (voor deze hoef je de call zelf niet te schrijven, ze start automatisch op)
- de `WriteLine` methode van de klasse `Console`
- de `Substring` methode van een stuk tekst

### Basissyntax

#### Definitie

In het begin schrijven we onze methodes zo (binnenkort zien we uitbreidingen):

```
public static void MethodeNaam()
{
    // code die een bepaalde taak uitvoert
}
```

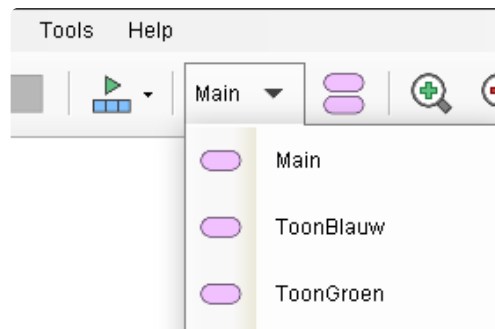
Dit stukje code definieert dus hoe het stappenplan er uit ziet. Het zorgt er niet voor dat het ook ooit wordt uitgevoerd. Daar is een call voor nodig. `public` zorgt dat deze methode opgeroepen kan worden van uit andere klassen, `static` is nodig omdat we klassen alleen als organisatieprincipe voor onze code gebruiken, `void` behandelen we iets verder op. Deze zaken, samen met de naam en de haakjes, noemen we ook de **signatuur** van de methode.

In Flowgorithm stemt de definitie van een methode overeen met een flowchart die in haar geheel getoond kan worden. Onderstaand voorbeeld bevat drie definities (flowcharts): `Main`, `ToonGroen` en `ToonBlauw`.



controlflowmethoden.fprg 1KB  
Binary

Je kan elke definitie (flowchart) terugvinden via dit menu:



## Oproep

Om de methode te gebruiken, moeten we een statement uitvoeren die er als volgt uit ziet:

```
MethodeNaam();
```

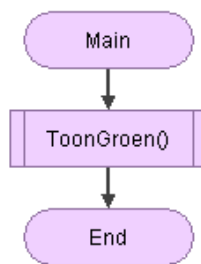
Aangezien elk programma begint met een oproep van de `Main` methode, zal dit programma dus wel de tekst `Groen` laten zien en niet de tekst `Blauw` :

```

class Program {
    public static void ToonGroen() {
        Console.WriteLine("Groen");
    }
    public static void ToonBlauw() {
        Console.WriteLine("Blauw");
    }
    public static void Main() {
        ToonGroen();
    }
}

```

In Flowgorithm herken je de oproep als volgt:



Een oproep van een methode betekent dus dat je een andere flowchart uitvoert alsof het één stap is van de flowchart waarin je bezig bent.

- i
 Net zoals eerder kan je dit ook in C# stap voor stap uitvoeren door de debugger te gebruiken. Om de werking van een methode in detail te zien, gebruik je "step into".

# Parameters



## Wat zijn parameters?

Een methode is een stappenplan voor een bepaalde taak. Met de syntax die we tot hiertoe gezien hebben, wordt deze taak altijd op exact dezelfde manier uitgevoerd. Dat beperkt de mogelijkheden enorm.

Een niet-digitaal voorbeeld. Ik plan een feest. Een van mijn deeltaken is om een simpele appeltaart te maken. De tweede is om een simpele perentaart te maken.

Het stappenplan voor de appeltaart is als volgt:

1. leg taartdeeg in een ronde bakvorm
2. doe er wat pudding op
3. snijd een appel in stukken
4. beleg met de stukjes fruit
5. zet in de oven

Het stappenplan voor de perentaart is als volgt:

1. leg taartdeeg in een ronde bakvorm
2. doe er wat pudding op
3. snijd een peer in stukken
4. beleg met de stukjes fruit
5. zet in de oven

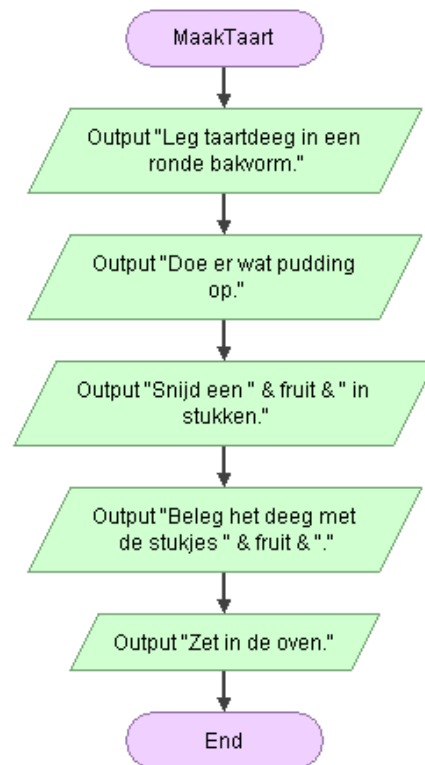
Dit is hetzelfde stappenplan, op het gebruikte fruit na. Eigenlijk kan je in stap 3 allerlei soorten fruit gebruiken, afhankelijk van het resultaat dat je wenst. We kunnen het stappenplan dus algemener formuleren:

1. leg taartdeeg in een ronde bakvorm
2. doe er wat pudding op
3. snij het stuk fruit dat je wil gebruiken in stukken
4. beleg met de stukjes fruit
5. zet in de oven

Op het moment dat we een taart willen bakken, moeten we pas voor een bepaald stuk fruit kiezen. In de tekst hierboven is "het stuk fruit" dus een soort variabele: we schrijven een naam maar we bedoelen daarmee een waarde die in de uitvoering wordt vastgelegd.

In code zijn dit soort variabelen **parameters** van een methode. In de definitie van de methode werken we met hun naam (zoals "het stuk fruit"). Wanneer we de methode oproepen, voorzien we hun waarde (zoals een appel of een peer of een ananas). In de call spreken we ook over **argumenten** in plaats van parameters. De termen worden soms door elkaar gehaald, maar ze betekenen dus niet exact hetzelfde.

In Flowgorithm kunnen kort noteren dat we een appeltaart **en** een perentaart willen maken:



Function Properties

Function

A function allows programs to both reuse code and simplify logic. Data is passed into functions using parameters.

Function Name:

MaakTaart

Parameters:

String fruit

↑

↓

Add

Edit

Remove

Return Type:

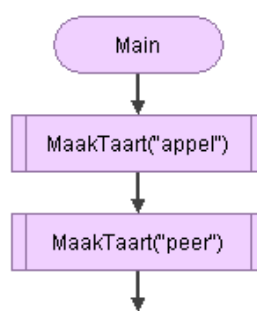
None

Return Variable:

OK

Cancel

Het stappenplan om om het even welke fruittaart te maken.





De code om snel twee fruittaarten te maken op basis van hetzelfde stappenplan.



taart.fprg 1KB

Binary

In de praktijk maken we natuurlijk geen taarten, maar doen we bewerkingen met gegevens. Volgende methode berekent en toont een bepaalde macht van een bepaald getal. De werkwijze is altijd dezelfde, maar de gebruiker beslist zelf welke macht hij wil berekenen en van welk getal:

```
public static void Macht(int grondtal, int macht)
{
    int resultaat = grondtal;
    for (int i = 1; i < macht; i++)
    {
        resultaat *= grondtal;
    }
    Console.WriteLine($"De {macht}e macht van {grondtal} is {resultaat}");
}
```

Dit is een definitie van een methode, dus vergelijkbaar met een stappenplan of een flowchart. We geven aan dat er twee stukjes informatie zijn die per uitvoering kunnen verschillen: het grondtal en de macht. Omdat dit eigenlijk variabelen zijn (met een scope beperkt tot deze methode) geven we ook hun type, net zoals bij variabelen die we op de reeds gekende wijze declareren.

Als we deze methode willen gebruiken, kunnen we dit doen:

```
public static void Main() {
    Console.WriteLine("Welk grondtal wil je gebruiken?");
    int grond = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Tot welke macht wil je verheffen?");
    int gewensteMacht = Convert.ToInt32(Console.ReadLine());
    Macht(grond, gewensteMacht);
}
```

De laatste regel bevat de call van de methode. De definitie bevat een paar "gaten" en de call vult deze in met de gewenste getallen. Het is belangrijk dat de gebruikte types en de verwachte types compatibel zijn. Het is bijvoorbeeld niet mogelijk `Macht("Hello World", "blabla");` te schrijven als call want strings zijn geen ints.

## De verbinding van definitie en oproep

Als je het stappenplan bovenaan deze pagina gebruikt om een appeltaart te maken, heb je achteraf geen appel meer. Hier kan het mechanisme van methodes je wat verrassen. Je moet in het achterhoofd houden dat de parameters van een methode hetzelfde werken als gewone variabelen.

Een voorbeeld:

```
int a = 3;
int b = a;
b = b+1;
Console.WriteLine(a);
Console.WriteLine(b);
```

Dit programma zal je eerst 3 tonen en dan, op de volgende regel, 4. Dit komt omdat de = op regel 2 niet betekent "a is hetzelfde als b", maar wel "kopieer het ding met naam a en geef de kopie de naam b". Dit is hoe toekenning werkt met alle types die we tot hertoe gezien hebben.

⚠ Dit is niet hoe toekenning altijd werkt. Met name voor arrays is het wat anders, maar dat geval zullen we later in meer detail behandelen.

Met argumenten van een methode-oproep is het hetzelfde. Nog een voorbeeld:

```
public static void VeranderGetal(int getal) {
    getal = getal - 1;
    Console.WriteLine($"Het getal is {getal}");
}

public static void Main() {
    int mijnGetal = 4;
    Console.WriteLine($"Het getal is {mijnGetal}");
    VeranderGetal(mijnGetal);
    Console.WriteLine($"Het getal is {mijnGetal}");
}
```

Je ziet dat het getal enkel gewijzigd is binnen de methode VeranderGetal en daarna weer teruggezet lijkt op de oude waarde. Dit is niet helemaal correct. Wat eigenlijk gebeurt, is het volgende:

1. een waarde 4 krijgt de naam mijnGetal
2. de waarde met naam mijnGetal wordt getoond
3. de waarde met naam mijnGetal wordt gekopieerd en krijgt de naam getal
4. de naam mijnGetal wordt toegekend aan het getal dat je verkrijgt door de kopie met één te verminderen en verwijst nu dus naar een 3
5. deze 3 wordt getoond
6. de oproep eindigt
7. heel de tijd is mijnGetal blijven verwijzen naar 4 en daarom wordt er terug een 4 getoond

mijnGetal en getal hebben dus nooit dezelfde data bevat. Bij het begin van een oproep van VeranderGetal zijn hun data wel kopieën van elkaar, maar na regel 2 is zelfs dat niet meer waar.

## Gekende parameters

Je hebt al vaak waarden als argument meegegeven:

- de tekst die je laat zien via `WriteLine`
- de grenzen waarbinnen een willekeurig getal bepaald moet worden met `Next`
- de tekst die je wil vervangen en de tekst die hem vervangt met `Replace`

Ook hier wordt altijd hetzelfde stappenplan gevolgd, maar de manier waarop dat plan doorlopen wordt, verschilt afhankelijk van de gebruikte argumenten.

## De rol van scope

Voor de eenvoud hebben we in bovenstaand voorbeeld twee namen gebruikt: `mijnGetal` en `getal`. Eigenlijk hoefde dat niet. We kunnen ook dit doen:

```
public static void VeranderGetal(int getal) {
    getal = getal - 1;
    Console.WriteLine($"Het getal is {getal}");
}

public static void Main() {
    int getal = 4;
    Console.WriteLine($"Het getal is {getal}");
    VeranderGetal(getal);
    Console.WriteLine($"Het getal is {getal}");
}
```

De parameter `getal` van `VeranderGetal` is niet dezelfde variabele als de variabele `getal` van `Main`. Elke functiedefinitie bakent een scope af. Vermits de definitie van `VeranderGetal` niet genest is in de definitie van `Main` (en het omgekeerde ook niet waar is) staan hun scopes los van elkaar. Binnenin `Main` bestaat er een variabele met naam `getal`, binnenin `VeranderGetal` bestaat er een andere variabele met naam `getal`.

Er staat wel een oproep van `VeranderGetal` in `Main`, maar dit heeft geen effect op de variabelen in scope. Enkel de definities zijn hier van belang.

# Return waarden

## ✓ Kennisclip return

## ✓ Kennisclip return en parameters samen

Een stappenplan, en dus een methode, voer je uit met het oog op een bepaald resultaat. Soms moet dat resultaat terugbezorgd worden aan een opdrachtgever. Soms niet. Methodes staan toe beide variaties op een stappenplan te schrijven.

We gebruiken opnieuw een proces uit het echte leven om de interactie te omschrijven voor we de sprong maken naar code.

## Voorbeeld uit het echte leven

Denk aan een ouderwetse bakker. Het stappenplan dat deze bakker volgt is er een om brood te maken. Om brood te maken heb je bloem nodig. Bloem maken is niet echt een onderdeel van het takenpakket van de bakker. Hij laat de molenaar dit voor hem doen. Hij vraagt niet gewoon aan de molenaar om bloem te **maken**, maar ook om deze aan hem te **bezorgen** zodat hij er zelf mee verder kan werken. De bakker is niet geïnteresseerd in hoe de molenaar bloem maakt, maar hij heeft het eindproduct nodig om zijn brood af te maken.

We vertalen nu deze interactie naar code en verklaren daarna de vertaling:

**i** Dit is maar een voorbeeld om de flow te verduidelijken. Er is geen "juiste" manier om een methode te schrijven om brood te bakken (tenzij je misschien een broodmachine programmeert).

```
public static void Bakker() {
    // we stellen elk ingrediënt voor als string
    string ingredient1 = "water";
    string ingredient2 = "gist";
    string ingredient3 = Molenaar();
    Console.WriteLine($"Ik maak brood met {ingredient1}, {ingredient2} en {ingredient3}");
}

public static string Molenaar() {
    return "bloem";
}
```

Hier vallen twee zaken op: in de signatuur (d.w.z. alles van het begin van de regel tot en met het gesloten ronde haakje) van de methode `Molenaar` staat niet `void`, maar `string`. Dit komt omdat deze methode een resultaat **teruggeeft** aan de opdrachtgever en dat resultaat van het type `string` is. Daarnaast heb je het sleutelwoord `return`, gevolgd door de tekst `"bloem"`. Het woordje `return` betekent: "bezorg dit aan de code die deze methode heeft opgeroepen". Anders gezegd: dit is hoe de molenaar de geproduceerde bloem overhandigt aan zijn opdrachtgever, de bakker. Waar je de call `Molenaar()` schrijft komt tijdens de uitvoering het geproduceerde resultaat dankzij die `return`.

## Het `return` type

Je moet dus noteren wat voor resultaat er achter `return` staat: een `string` of eventueel iets anders. Als er een `string` volgt na `return`, zeggen we dat `string` het **return type** is van die methode. Als een methode een geheel getal produceert, heeft ze het return type `int` of een verwant getaltype.

Maar niet alle methodes bezorgen iets terug aan hun opdrachtgever. Sommige hebben gewoon een effect. Voor deze methodes schrijven we `void` als return type. Als een methode dit return type heeft, kunnen we het resultaat van een call dus niet toekennen aan een variabele, want er is geen resultaat.

❗ "Er is geen resultaat" wil niet zeggen dat een `void` methode niets doet. `Console.WriteLine` is bijvoorbeeld een methode met return type `void`. Het punt is dat de call je niets terugbezorgt waarmee je verder kan werken. Je kan bijvoorbeeld niet schrijven: `string tekst = Console.WriteLine("Dit is tekst");`

Er is dus een groot verschil tussen `return "tekst";` en `Console.WriteLine("tekst");`. Bij de eerste code is er geen garantie dat de geproduceerde tekst ooit op het scherm verschijnt, maar je kan er wel mee verder werken. Bij de tweede verschijnt hij per definitie wel op het scherm, maar kan je hem niet toekennen aan een variabele om later mee verder te werken.

We kunnen dus wel doen: `string ingredient3 = Molenaar();` (want het return type van `Molenaar()` is `string`) maar we kunnen niet schrijven: `string product = Bakker();` (want `Bakker()` heeft return type `void` en produceert dus geen resultaat). Deze code compileert dan ook niet.

## Gekend voorbeeld van een vaak gemaakte fout

Je kent de methode `Replace` van strings. Je gebruikt deze als volgt:

```
string ingevoerdeTekst = Console.ReadLine();
string tekstZonderSpaties = ingevoerdeTekst.Replace(" ", "");
Console.WriteLine($"Jouw tekst zonder spaties is: {tekstZonderSpaties}");
```

Veel beginners doen het volgende:

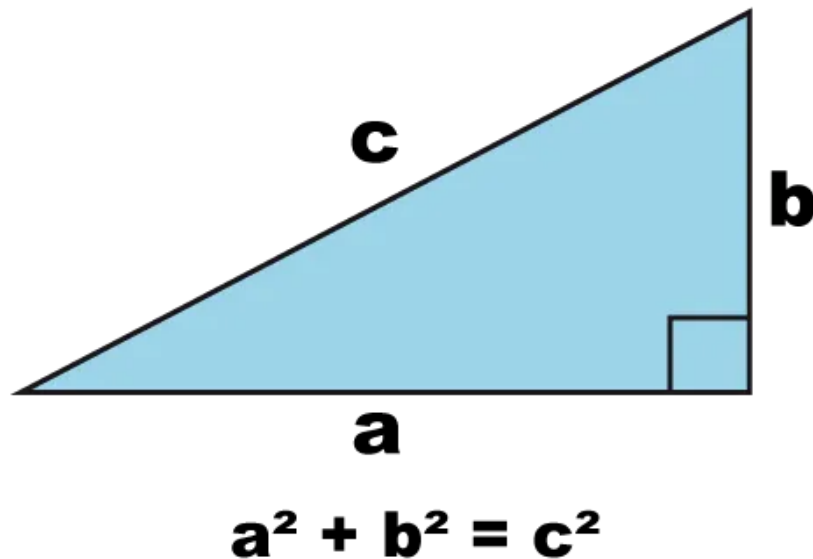
```
string ingevoerdeTekst = Console.ReadLine();
ingevoerdeTekst.Replace(" ", "");
Console.WriteLine($"Jouw tekst zonder spaties is: {ingevoerdeTekst}");
```

Dit is fout. `Replace` berekent het resultaat van de aanpassing en geeft dat terug aan de opdrachtgever. Het past de ingevoerde tekst niet aan. In de code van `Replace` staat dus ook een `return` !

## Parameters en returnwaarden samen

Als je parameters en returnwaarden combineert, worden methoden erg flexibel. Dan kan je al echte bouwsteentjes van een programma gaan schrijven.

Een voorbeeld: de stelling van Pythagoras vertelt ons hoe lang de schuine zijde van een driehoek met twee rechte hoeken is:



Als we deze berekening regelmatig uitvoeren voor verschillende waarden van `a` en `b`, kunnen we ze als volgt in een methode gieten:

```
public static double Pythagoras(double a, double b) {
    double som = a * a + b * b;
    return Math.Sqrt(som);
}
```

Nu kunnen we heel snel allerlei schuine zijdes uitrekenen, bv.:

```
public static void Main() {
    Console.WriteLine($"De schuine zijde van een driehoek met benen van 4cm en 3cm is {Pythago
    Console.WriteLine($"De schuine zijde van een driehoek met benen van 12cm en 2cm is {Pythag
}
```

Bovendien maken we gebruik van het feit dat de methode `Sqrt` (*square root* is vierkantswortel) ook een parameter heeft en een resultaat teruggeeft. Eigenlijk verloopt een uitvoering dus zo:

1. de methode `Main` start op
2. we willen bepaalde tekst uitprinten, maar we moeten die eerst nog bepalen met de methode `Pythagoras`
3. de methode `Pythagoras` start op met de argumenten 4 en 3
4. de methode berekent de som van de kwadraten en noemt deze `som`
5. de methode maakt zelf gebruik van nog een methode, `Sqrt`, en geeft daarbij `som` als argument
6. `Sqrt` rekt de vierkantswortel uit en geeft deze terug aan `Pythagoras`
7. `Pythagoras` geeft deze verder door aan `Main`
8. de zin wordt correct afgeprint

Meestal is het dus een goed idee zo weinig mogelijk informatie te communiceren met `WriteLine` en zo veel mogelijk met `return` te werken. Meerbepaald: gebruik `WriteLine` wanneer je de uiteindelijke informatie in haar uiteindelijke vorm hebt en deze echt aan de gebruiker moet tonen. Gebruik `return` wanneer je het resultaat van een deeltaak aan een opdrachtgever wil bezorgen.

Volgend Flowgorithm toont aan hoe je dankzij methoden een probleem kan opbreken in meerdere kleine stappen, waarbij je informatie doorgeeft via parameters en returnwaarden:



pythagoras-uitgebreid.fprg 2KB

Binary

# Geavanceerde methoden

Volgende sectie is grotendeels gebaseerd op het volgende [artikel](#).



## Named parameters

Wanneer je een methode aanroept is de volgorde van je argumenten belangrijk: deze moeten meegeven worden in de volgorde zoals de methode parameters ze verwachten.

Met behulp van named parameters kan je echter expliciet aangeven welke argument aan welke methode-parameter moet meegegeven worden.

Stel dat we een methode hebben met volgende signatuur:

```
static void PrintOrderDetails(string sellerName, int orderNum, string productName)
{
    Console.WriteLine($"Verkoper: {sellerName}");
    Console.WriteLine($"Ordernummer: {orderNum}");
    Console.WriteLine($"Product: {productName}");
}
```

Zonder named parameters zou een aanroep van deze methode als volgt kunnen zijn:

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

We kunnen named parameters aangeven door de naam van de parameter gevolg door een dubbel punt en de waarde. Als we dus bovenstaande methode willen aanroepen kan dat ook als volgt met named parameters:

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
```

of ook:

```
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Volgorde van named parameters belangrijk



Je mag ook een combinatie doen van named en gewone parameters, maar **dan is de volgorde belangrijk**: je moet je dan houden aan de volgorde van de methode-volgorde. Je verbetert hiermee de leesbaarheid van je code dus (maar krijgt niet het voordeel van een eigen volgorde te hanteren). Enkele voorbeelden:

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");
```

Enkele **NIET GELDIGE** voorbeelden:

```
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
```

## Optionele parameters

Soms wil je dat een methode een standaard waarde voor een parameter gebruikt indien de programmeur in z'n aanroep geen waarde meegaf. Dat kan met behulp van optionele of default parameters.

Je geeft aan dat een parameter optioneel is door deze een default waarde te geven in de methode-signatuur. Deze waarde zal dan gebruikt worden indien de parameter geen waarde van de aanroeper heeft gekregen.

**Optionele parameters worden steeds achteraan de parameterlijst van de methode geplaatst .**

In het volgende voorbeeld maken we een nieuwe methode aan en geven aan dat de laatste parameters optioneel zijn. `discountPercentage` past een "gewone" korting toe op de aankoop van het aantal items. `doubleDiscount` is een speciale extra korting die het percentage verdubbelt. We veronderstellen dat items meestal verkocht worden zonder korting:

```
static double ComputePrice(int numberOfItems, double itemPrice, double discountPercentage =
    if (not doubleDiscount) {
        return (numberOfItems * itemPrice) * (1.0 - discountPercentage / 100);
    }
    else {
        return (numberOfItems * itemPrice) * (1.0 - 2 * discountPercentage / 100);
    }
}
```

Volgende manieren zijn nu geldige manieren om de methode aan te roepen:

```
Console.WriteLine(ComputePrice(4,10.0)); // klassieke aanroep
Console.WriteLine(ComputePrice(4,10.0,5.0)); // 5% korting op de totaalprijs
Console.WriteLine(ComputePrice(4,10.0,5.0,true)); // 10% korting op de totaalprijs omdat de
```

Je mag enkel de optionele parameters van achter naar voor weglaten. Volgende aanroep is dus **niet** geldig:

```
Console.WriteLine(ComputePrice(4,10.0,true)); // derde argument moet een double zijn!
```

Door de argumenten te benoemen, kunnen we dit indien gewenst omzeilen. Volgende aanroep is wel geldig:

```
// dit heeft weinig zin want we verdubbelen 0% korting, maar voor C# werkt dit wel  
Console.WriteLine(ComputePrice(4,10.0,doubleDiscount: true));
```

## Method overloading


Method overloading wil zeggen dat je een **methode met dezelfde naam en returntype** meerdere keren definieert maar met andere parameters qua type en aantal. De compiler zal dan zelf bepalen welke methode moet aangeroepen worden gebaseerd op het aantal en type parameters dat je meegeeft.

Volgende methoden zijn overloaded:

```
static int ComputeArea(int lengte, int breedte)  
{  
    int opp = lengte*breedte;  
    return opp;  
}  
  
static int ComputeArea(int radius)  
{  
    int opp = (int)(Math.PI*radius*radius);  
    return opp;  
}
```

Afhankelijk van de aanroep zal dus de ene of andere uitgevoerd worden. Volgende code zal dus werken:

```
Console.WriteLine($"Rechthoek: {ComputeArea(5, 6)}");  
Console.WriteLine($"Circle: {ComputeArea(7)}");
```

 Method overloading is de reden waarom je in Visual Studio Code de documentatie van meerdere versies van een methode kan bekijken. Dit doe je door op de pijltjes naast de info over die methode te klikken.

# Oefeningen

## Oefeningen

Al deze oefeningen schrijf je in een klassen `Methodes` .

### Oefening H7-ReeksOperaties

#### Leerdoelen

- Methodes definiëren met parameters
- Oproepen van de methodes

Functionele analyse

Je stelt een aantal typische berekeningen voor met methodes.

#### Technische analyse

Voor deze oefening schrijf je **meerdere** methodes. Je plaatst er slechts één in je keuzemenu, namelijk `ReeksOperaties` . Deze voert alle andere methodes van deze oefening één voor één uit.

Enkel `ReeksOperaties` mag `Console.ReadLine` statements bevatten. De andere methodes **tonen** hun resultaat op het scherm door middel van `Console.WriteLine` statements **in de methode zelf** die het resultaat bepaalt. (Met andere woorden, alle methodes zijn `void` .)

De andere methodes zijn:

- `BerekenStraal` , die de straal van een cirkel kan berekenen waarvan je de diameter meegeeft (de diameter geef je mee als argument). De straal is de helft van de diameter.
- Idem voor `BerekenOmtrek` en `BerekenOppervlakte` . De omtrek is de diameter maal het getal `pi`, dat je voorstelt als `Math.Pi` . De oppervlakte is straal maal straal maal `pi`.
- Methode `Maximum` die het grootste van 2 getallen teruggeeft (beide getallen geef je mee als argument).
- Methode `IsEven` die toont of een getal even of oneven is.
- Methode `ToonOnEvenGetallen` die alle oneven getallen van 1 tot `n` toont waarbij `n` als argument wordt meegegeven.

Voorbeeldinteractie

```

Geef de diameter van de cirkel: 12,4
De straal van de cirkel is 6,200
De omtrek van de cirkel is 38,956
De straal van de cirkel is 120,763
Geef twee getallen:
25,2
25,1999
Het grootste getal van 25,2 en 25,1999 is 25,2
Geef een geheel getal:
133
Getal 133 is een oneven getal
De reeks van oneven getallen van 1 tot 133 is:
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101 103 105 107 109 111 113 115 117 119 121 123 125 127 129 131 133

```

## Oefening H7-EmailadresGenerator

### Leerdoelen

- Methodes definiëren met return type en parameters
- Oproepen van de methodes

### Functionele analyse

Schrijf een programma dat op basis van je voornaam, naam en het feit of je al dan niet een student bent een AP e-mailadres genereert. Het e-mailadres moet uit allemaal kleine letters bestaan en moet een geldig e-mailadres zijn.

Je mag geen String methodes gebruiken maar je schrijft je eigen methodes. Als aanzet is de methode StringToLower al uitgewerkt.

**Uitbreiding:** kijk of voor al je lectoren en medestudenten het gegenereerde e-mailadres geldig is. Pas eventueel je code aan indien niet.

### Technische analyse

Werk volgend flowchart verder af en zet om in C#.



H07-EmailGeneratorTemplate.fprg 3KB

Binary

### Voorbeeldinteractie

```

Geef voornaam: Jan
Geef achternaam: Van den Poel
student <false/true>:false
email: jan.vandenpoel@ap.be

```

## Oefening: H7-ReeksOperatiesMetReturn

### Leerdoelen

- Methodes definiëren met parameters en returnwaarde
- Oproepen van de methodes
- Gebruiken van returnwaarden

## Functionele analyse

Je stelt een aantal typische berekeningen voor met methodes die geschikt zijn voor gebruik in andere programma's.

## Technische analyse

Voor deze oefening schrijf je **meerdere** methodes. Je plaatst er slechts één in je keuzemenu, namelijk `ReeksOperatiesMetReturn`. Deze voert alle andere methodes van deze oefening één voor één uit.

Enkel `ReeksOperaties` mag **`Console.ReadLine`** of **`Console.WriteLine`** statements bevatten. De andere methodes berekenen hun resultaat, zonder het te tonen op het scherm. Met andere woorden, hun return type is niet `void`.

De andere methodes zijn:

- `BerekenStraalMetReturn`, die de straal van een cirkel kan berekenen waarvan je de diameter meegeeft (de diameter geef je mee als argument). De straal is de helft van de diameter. Voor je deze toont, rond je ze in **`ReeksOperatiesMetReturn`** af tot één cijfer na de komma met `Math.Round(reedsBerekendeStraal,1)`.
- Idem voor `BerekenOmtrekMetReturn` en `BerekenOppervlakteMetReturn`. Ook deze waarden moet je, nadat ze uitgerekend zijn, afronden tot 1 cijfer na de komma.
- Methode `MaximumMetReturn` die het grootste van 2 getallen teruggeeft (beide getallen geef je mee als argument).
- Methode `IsEvenMetReturn` die bepaalt of een getal even of oneven is. De returnwaarde vertelt dus of iets waar of niet waar is.
- Methode `BepaalEvenGetallenMetReturn`. Deze geeft als antwoord een `List` van alle even getallen tot n. `ReeksOperatiesMetReturn` toont deze getallen dan, gescheiden door een komma.

## Voorbeeldinteractie

Zoals boven, met de vermelde aanpassingen (afronding tot 1 cijfer na de komma en getallen gescheiden door `,`)

## H7-driehoeken

### Functionele analyse

We schrijven een simpel tekenprogramma, dat driehoeken van een bepaalde afmeting, met een bepaald patroon kan tekenen.

## Technische analyse

Schrijf een methode `TekenDriehoek`. Deze methode verwacht twee zaken: een karakter om het patroon voor te stellen en de hoogte van de driehoek. Ze roept zelf **meermaals** een andere methode `TekenRegel` op, die één regel van de driehoek tekent. `TekenRegel` toont niets op het scherm via `Console.WriteLine!`

## Voorbeeldinteractie

```
Hoe hoog is de driehoek?
```

```
> 4
```

```
Welk karakter gebruiken we als patroon?
```

```
> #
```

```
#
```

```
##
```

```
###
```

```
####
```

```
Hoe hoog is de driehoek?
```

```
> 3
```

```
Welk karakter gebruiken we als patroon?
```

```
> .
```

```
.
```

```
..
```

```
...
```