

1.0.0


## H5: Loops

# Loops intro

Herhalingen (**loops**) creëer je wanneer bepaalde code een aantal keer moet herhaald worden. Hoe vaak de herhaling moet duren is afhankelijk van de conditie die je hebt bepaald. Deze conditie is een booleaanse expressie, net zoals de conditie van `if`.

In het vorige hoofdstuk leerden we hoe we met behulp van beslissingen onze code konden *branchen*, aftakken zodat andere code werd uitgevoerd afhankelijk van de staat van bepaalde variabelen of invoer van de gebruiker. Wat we nog niet konden was **terug naar boven** vertakken. Soms willen we dat een heel stuk code 2 of meerdere keren moet uitgevoerd worden zo lang als of totdat aan een bepaalde conditie wordt voldaan. "Voer volgende code uit tot dat de gebruiker een bepaalde waarde invoert." Dit zal bijna net hetzelfde werken als bij `if`, maar nadat je code is uitgevoerd, wordt de conditie (opnieuw) gecontroleerd om na te gaan of de code (opnieuw) moet worden uitgevoerd.

**Door herhalende code met loops te schrijven maken we onze code korter en bijgevolg ook minder foutgevoelig en beter onderhoudbaar.**

 Van zodra je dezelfde lijn(en) code onder elkaar in je code ziet staan (door bijvoorbeeld te copy pasten) is de kans zéér groot dat je dit korter kunt schrijven met loops.

## Soorten loops

Er zijn verschillende soorten loops:

- **Definite of counted loop**: een loop waar het aantal iteraties vooraf van gekend is. (Bijvoorbeeld 100 keer of een aantal keer gelijk aan de waarde van de variabele `x`).
- **Indefinite of sentinel loop**: een loop waarvan op voorhand niet kan gezegd worden hoe vaak deze zal uitgevoerd worden. Input van de gebruiker of een interne test zal bepalen wanneer de loop stopt (bv. "Voer getallen in, voer -1 in om te stoppen")
- **Oneindige loop**: een loop die nooit stopt. Soms gewenst, vaak een bug.

## Loops in C#

Er zijn 3 manieren om zogenaamde loops te maken in C#:

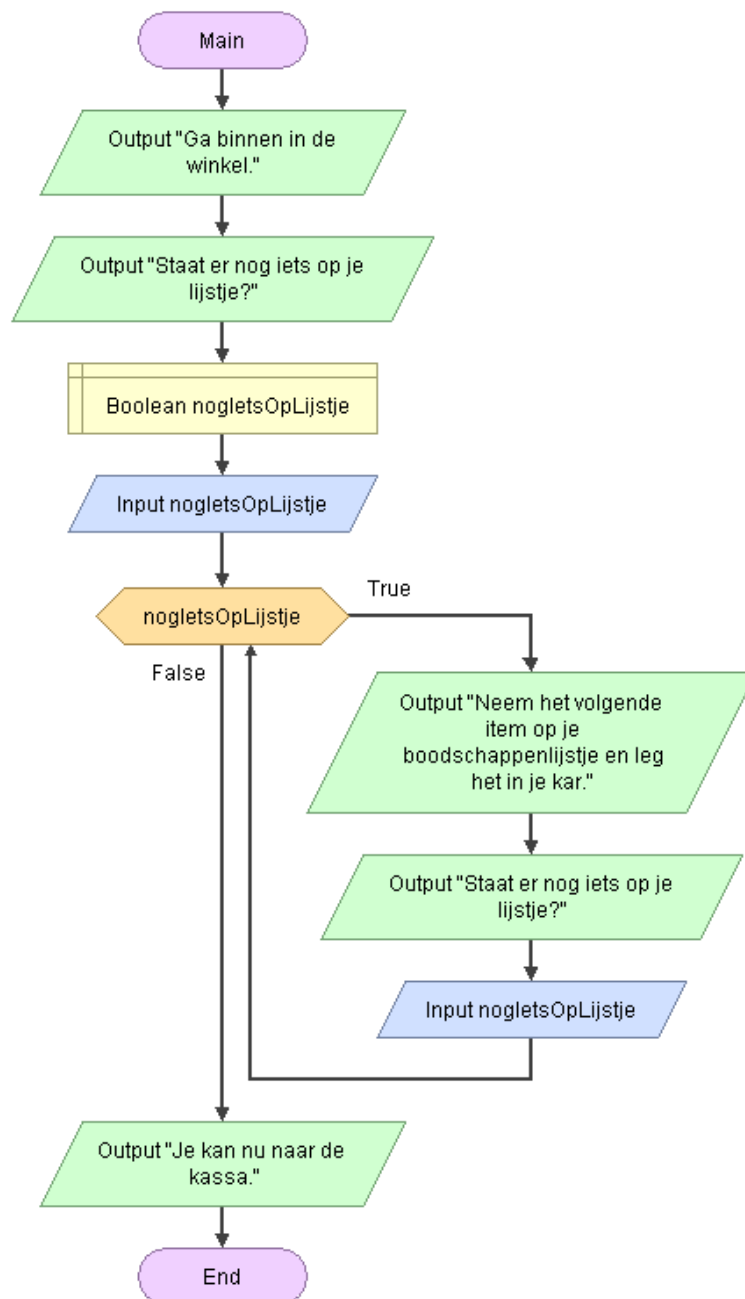
- **while**: zal 0 of meerdere keren uitgevoerd worden
- **do while**: zal minimaal 1 keer uitgevoerd worden
- **for**: een alternatieve iets compactere manier om loops te beschrijven

# While en Do While

## While

### Voorbeeld werking

Volgend Flowgorithm programma demonstreert hoe een `while` lus wordt toegepast wanneer je systematisch boodschappen afwerkt.



Een while wordt aangegeven door een booleaanse expressie waaruit een True-tak vertrekt die er daarna weer in terugkomt. Hier: de oranje zeshoek.

Omdat de `True` -tak terug leidt naar de controle van de voorwaarde, blijven we dezelfde stappen herhalen tot niet meer aan de voorwaarde voldaan is. Dit is het basisconcept achter een `while` lus. Het woord "while" betekent dan ook "zolang als".

Dit is anders dan bij `if` : het woordje "if" betekent gewoonweg "als" en houdt dus niet in dat we dezelfde stappen zullen herhalen. Bij `while` blijven we dezelfde stappen uitvoeren zo lang dat nodig is.

## Syntax

De syntax van een while loop is eenvoudig:

```
while (booleaanse expressie)
{
    // C# die zal uitgevoerd worden zolang de booleaanse expressie waar is
}
```

Waarbij, net als bij een `if` statement, de conditie uitgedrukt wordt als een booleaanse expressie.

Zolang de conditie `true` is zal de code binnen de accolades uitgevoerd worden. Indien dus de conditie reeds vanaf het begin `false` is dan zal de code binnen de `while` -loop niet worden uitgevoerd.

Telkens wanneer het programma aan het einde van het `while` codeblock komt springt het terug naar de conditie bovenaan en zal de test wederom uitvoeren. Is deze weer `true` dan wordt de code weer uitgevoerd. Van zodra de test `false` is zal de code voorbij het codeblock springen en na het `while` codeblok doorgaan.

De code voor ons boodschappenlijstje ziet er dan ook zo uit:

```
public static void Main(string[] args)
{
    Console.WriteLine("Ga binnen in de winkel.");
    Console.WriteLine("Staat er nog iets op je lijstje?");
    boolean nogIetsOpLijstje;

    nogIetsOpLijstje = (boolean) readValue();
    while (nogIetsOpLijstje)
    {
        Console.WriteLine("Neem het volgende item op je boodschappenlijstje en leg het in je kar.");
        Console.WriteLine("Staat er nog iets op je lijstje?");
        nogIetsOpLijstje = (boolean) readValue();
    }
    Console.WriteLine("Je kan nu naar de kassa.");
}
```

Een tweede voorbeeld van een eenvoudige while loop:

```
int myCount = 0;

while (myCount < 100)
{
    myCount++;
    Console.WriteLine(myCount);
}
```

Zolang `myCount` kleiner is dan 100 ( `myCount < 100` ) zal `myCount` met 1 verhoogd worden en zal de huidige waarde van `myCount` getoond worden. We krijgen met dit programma dus alle getallen van 1 tot en met 100 op het scherm onder elkaar te zien.

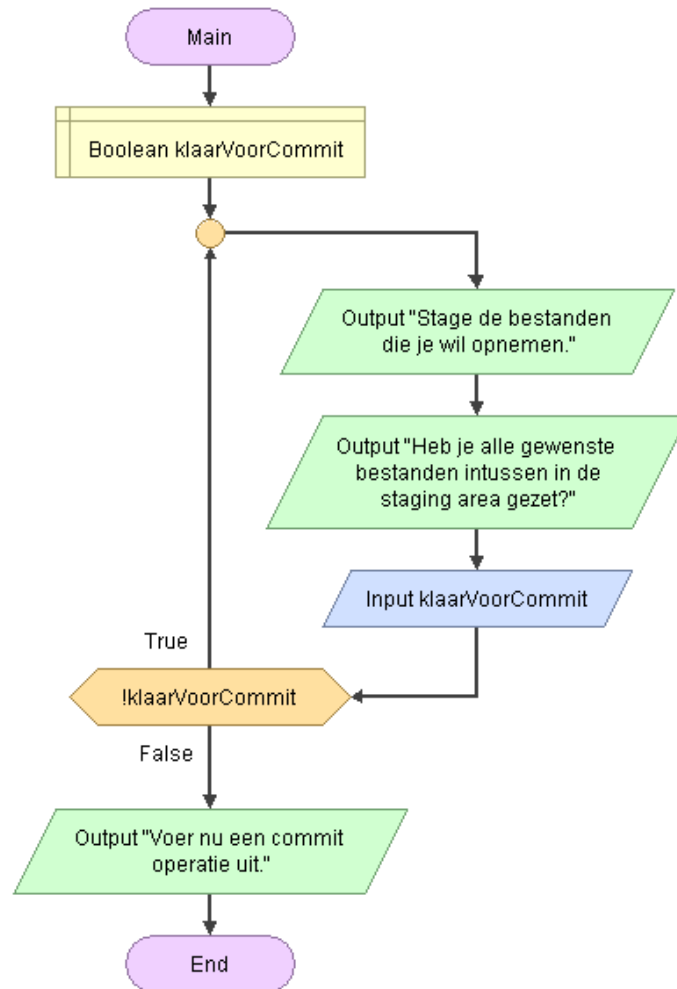
Daar de test gebeurt aan het begin van de loop wil dit zeggen dat het getal 100 nog wel getoond zal worden. **Begrijp je waarom?** Test dit zelf!

## Do while

In tegenstelling tot een while loop, zal een do-while loop sowieso **minstens 1 keer uitgevoerd worden**.

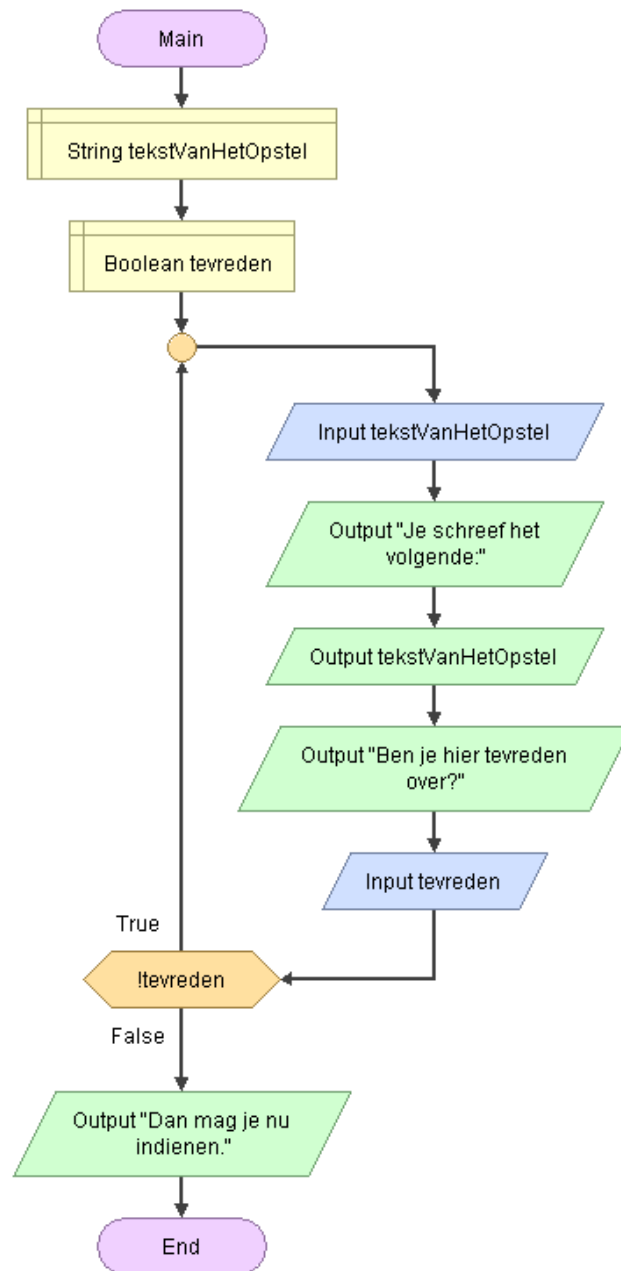
### Voorbeelden werking

Volgende flowchart helpt je bepalen of je klaar bent om een commit operatie uit te voeren in Git. Zoals je in andere vakken leert, moet je in Git eerst de gewenste bestanden in de staging area zetten. Dat kan soms in één commando gebeuren en soms in meerdere commando's, maar je moet altijd wel minstens één keer stagen.



Een do-while wordt aangegeven door een booleaanse expressie die na een reeks instructies wordt geëvalueerd en waarbij de True-tak terug tot diezelfde reeks instructies leidt.

Een tweede voorbeeld: een opstel schrijven. Dat moet je altijd nog eens nakijken. Als het niet goed is, moet je je tekst aanpassen en opnieuw controleren. Pas wanneer je helemaal tevreden bent, mag je inzenden.



De syntax van een do-while is eveneens verraderlijk eenvoudig:

```
do{
    // C# die uitgevoerd zal worden zolang de booleaanse expressie waar is
} while (booleaanse expressie);
```

⚠ Merk op dat achteraan de conditie een puntkomma na het ronde haakje staat. **Dit is een véél voorkomende fout. Bij een while is dit niet zo!** Daar de test van een do-while achteraan de code van de loop gebeurt is het logisch dat een do-while dus minstens 1 keer wordt uitgevoerd. Het volgende eenvoudige aftelprogramma toont de werking van de do-while loop.



Voor het opstel ziet dit er bijvoorbeeld zo uit:

```
public static void Main(string[] args)
{
    string tekstVanHetOpstel;
    boolean tevreden;

    do
    {
        tekstVanHetOpstel = Console.ReadLine();
        Console.WriteLine("Je schreef het volgende:");
        Console.WriteLine(tekstVanHetOpstel);
        Console.WriteLine("Ben je hier tevreden over?");
        tevreden = (boolean) readValue();
    }
    while (!tevreden);
    Console.WriteLine("Dan mag je nu indienen.");
}
```

Je kan dit ook doen met een gewone while (je kan elke do while herschrijven met een gewone while), maar dan kost dat meer code. Dat verhoogt dan weer de kans op fouten.

## Complexe condities

Uiteraard mag de conditie waaraan een loop moet voldoen complexer zijn door middel van de relationele operatoren.

Volgende `while` bijvoorbeeld zal de gebruiker aanraden in bed te blijven zolang deze ziek of moe is:

```
while(gebruikerVoeltZichZiek || gebruikerVoeltZichMoe)
{
    Console.WriteLine("Blijf in bed!");
    Console.ReadLine();
}
```

## Oneindige loops

Indien de loop-conditie nooit `false` wordt, dan heb je een oneindige loop gemaakt. Soms is dit gewenst gedrag (bijvoorbeeld in een programma dat constant je scherm moet updaten, zoals in een game), soms is dit een bug.

Volgende twee voorbeelden tonen dit:

- Een bewust oneindige loop:

```
while(true)

{
    // code die nooit hoort te stoppen of die zelf het programma kan afsluiten
}
```

- Een bug die een oneindige loop veroorzaakt:

```
int teller = 0;
while(teller<10)
{
    Console.WriteLine(teller);
}
```

## Scope van variabelen in loops

Let er op dat de **scope** van variabelen bij loops zeer belangrijk is. Indien je een variabele binnen de loop definieert dan zal deze steeds terug "verdwijnen" wanneer de cyclus van de loop is afgewerkt. Latere declaraties voor variabelen met dezelfde naam hebben niets meer te maken met de oorspronkelijke variabele.

Volgende code toont bijvoorbeeld **foutief** hoe je de som van de eerste 10 getallen (1+2+3+...+10) zou maken:

```
int teller= 1;
while(teller <= 10)
{
    int som = 0;
    som = som+teller;
    teller++;
}
Console.WriteLine(som); //deze lijn zal fout genereren
```

De **correcte** manier om dit op te lossen is te beseffen dat de variabele som enkel binnen de accolades van de while-loop gekend is. Op de koop toe wordt deze steeds terug op 0 gezet en er kan dus geen som van alle teller-waarden bijgehouden worden:

```
int teller= 1;
int som=0;
while(teller <= 10)
{
    som = som+teller;
    teller++
}
Console.WriteLine(som);
```

Scope is eerder al in het algemeen behandeld, maar zeker in loops worden er veel fouten tegen gemaakt. Daarom beklemtonen we het hier nog eens.



# For

## For

Het gebeurt heel vaak dat we een teller bijhouden die aangeeft hoe vaak een lus al doorlopen is. Wanneer de teller een bepaalde waarde bereikt moet de loop afgesloten worden.

Bijvoorbeeld volgende code om alle even getallen van 0 tot 10 te tonen:

```
int k = 0;
while(k <= 10)
{
    Console.WriteLine(k);
    k = k + 2;
}
```

Met een `for`-loop kunnen we deze veel voorkomende code-constructie verkort schrijven.

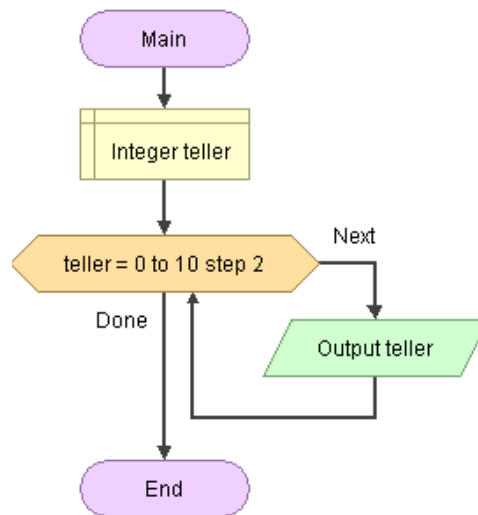
## For syntax

De syntax van een `for`-loop is de volgende:

```
for (setup; test; update)
{
    // alles tussen deze accolades is de "body" van de lus
}
```

- **setup**: In het setup gedeelte voeren we code uit die klaar moet zijn voor de lus voor het eerst begint. Dat betekent in de praktijk bijna altijd dat we een **loop variabele** aanmaken waarmee we bijhouden hoe vaak we al door de lus zijn gegaan. Variabelen die je hier aanmaakt hebben dezelfde scope als de **body** van de lus. Wanneer de `for` eindigt, bestaan ze met andere woorden niet meer.
- **test**: Hier plaatsen we een booleaanse expressie die nagaat of de lus mag worden uitgevoerd. Deze heeft dus niet dezelfde functie als de booleaanse expressie van een `while`. Als ze `false` oplevert, wordt de body van de lus niet meer uitgevoerd. Dit betekent in de praktijk heel vaak dat we nagaan of de loop variabele al een zekere waarde heeft bereikt.
- **update**: Hier plaatsen we wat er moet gebeuren telkens de loop body is afgewerkt. Meestal zullen we hier de loop variabele verhogen of verlagen.

In Flowgorithm (dit stemt helaas niet 100% overeen met C# omwille van een klein verschil in scope):



"step 2" betekent dat we na elke cyclus de teller met 2 verhogen

Gebruiken we deze kennis nu, dan kunnen we de eerder vermelde code om de even getallen van 0 tot en met 10 tonen als volgt:

```

for (int teller = 0; teller <= 10; teller += 2)
{
    Console.WriteLine(teller);
}
  
```

Voor de indexvariabele kiest men meestal niet `teller`, maar `i`, maar dat is niet noodzakelijk. In de setup wordt dus een variabele op een start-waarde gezet. De test zal aan de start van iedere loop kijken of de voorwaarde nog waar is, indien dat het geval is dan wordt een nieuwe loop gestart en wordt `i` met een bepaalde waarde, zoals in update aangegeven, verhoogd.

⚠ Je kan niets met een `for` dat je met een `while` niet kan. Je kan niets met een `while` dat je met een `for` niet kan. Als je dit niet inzielt, heb je lussen nog niet begrepen en moet je dit hoofdstuk vanaf het begin opnieuw lezen.

P.S.: wat voor lus is hier net gebruikt?!

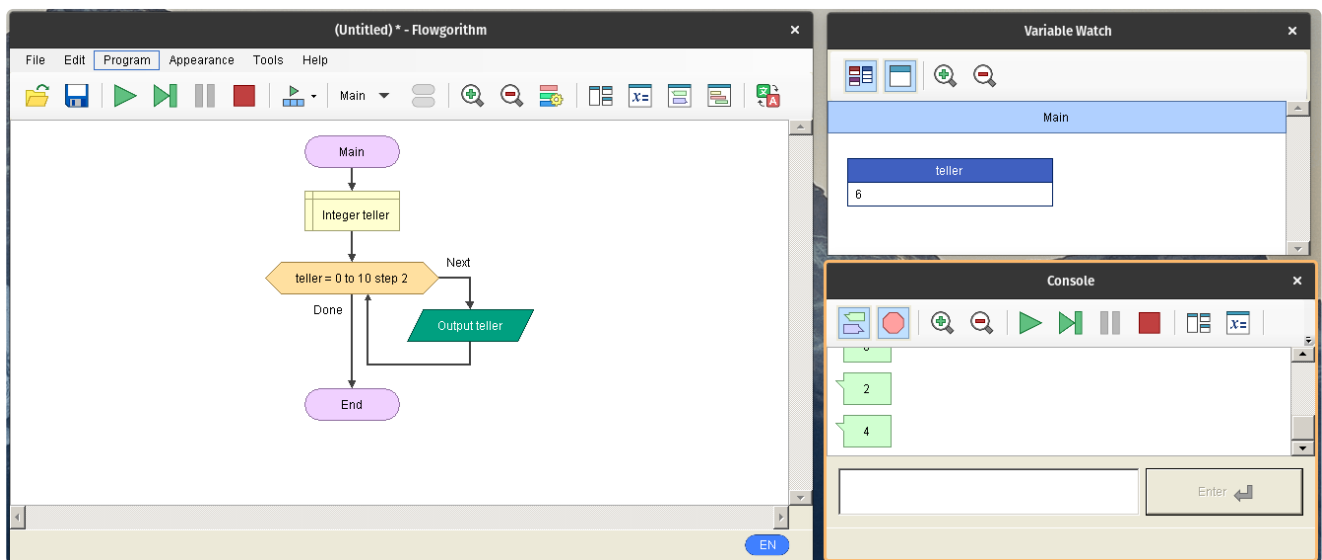
❗ Waarom stemt de Flowgorithm hierboven niet perfect overeen met de C#-code? Omdat in de C#-code `teller` **gedeclareerd wordt als onderdeel van de `for`**. In de code die we uit de Flowgorithm hierboven kunnen genereren, wordt `teller` niet gedeclareerd als onderdeel van de lus en blijft deze dus bestaan nadat de lus is afgewerkt. Dit is een klein detail, maar we vinden het belangrijk uit te leggen waarom deze code niet gelijkwaardig is.

# Debuggen

## Debuggen

**Debugging is een essentiële skill.** Bekijk onderstaande clip en zorg dat je de werking van elk gedemonstreerd onderdeel begrijpt. Gebruik voortaan altijd de debugger als je programma wel uitvoert, maar verkeerde resultaten produceert vooraleer je de lector aanspreekt.

Intussen kan je hopelijk stap voor stap een Flowgorithm programma in je hoofd uitvoeren. Als je de software gedownload hebt, kan je dit ook op je machine doen door telkens F6 te gebruiken. Je kan ook op elk punt in het programma kijken wat de waarden van je variabelen zijn via de knop met het symbooltje `x =`.



Flowgorithm programma, gepauzeerd middenin een for-lus, met weergave van de teller.

Voor C#-programma's is er een gelijkaardige tool in Visual Studio Code: de debugger. De naam is wat misleidend: deze zal geen bugs verwijderen uit je programma, maar zal je wel helpen stap voor stap door je programma te gaan. Als je programma bugs vertoont, is de kans groot dat je op deze manier de oorzaak kan vinden.

Dit wordt geïllustreerd door volgende code:

```

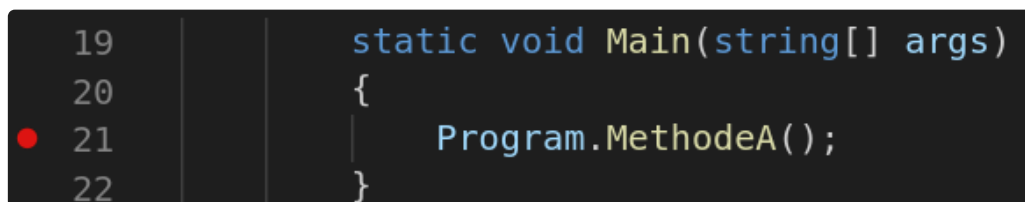
using System;

namespace Demonstratie_debugger
{
    class Program
    {
        public static void MethodeB() {
            int getal = 2;
            double kommagetal = 4.9;
            Console.WriteLine("B");
        }

        public static void MethodeA() {
            Console.WriteLine("A1");
            MethodeB();
            int getal1 = 56;
            int getal2 = 13;
            Console.WriteLine("A2");
        }
        static void Main(string[] args)
        {
            Program.MethodeA();
        }
    }
}

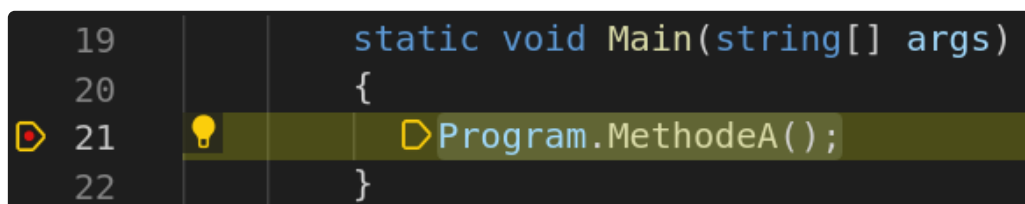
```

Eerst zet je ergens in deze code een **breakpoint**. Dit ziet er uit als een rood bolletje naast de code waar je programma moet "stilstaan". Je doet dit door op de plaats van het bolletje te klikken.



breakpoint

Dan start je je programma op via "Run -> Start debugging" (of via F5). De regel waarop je code gepauzeerd is, wordt aangeduid.



In dit geval is er maar één instructie: "voer MethodeA uit". We kunnen dit als één geheel doen via "step over" of we kunnen ook de stappen binnenin MethodeA bekijken via "step into".



step over: "voer de instructie als één geheel uit"    step into: "bekijk de instructie zelf als een reeks kleinere instructies"

In dit geval kunnen we in detail kijken hoe `MethodeA` werkt via "step into". We kunnen "step into" niet altijd uitvoeren: we hebben hiervoor toegang nodig tot de broncode van de instructie die we verder willen bekijken.

Door enkele stappen uit te voeren, kunnen we pauzeren op de toekenning van `getal2`:

```
public static void MethodeA() {  
    Console.WriteLine("A1");  
    MethodeB();  
    int getal1 = 56;  
    int getal2 = 13;  
    Console.WriteLine("A2");  
}
```

Op dit punt in de code, kunnen we onder "variables" bekijken welke waarden onze variabelen op dit moment hebben (merk op: de toekenning van 13 is nog niet gebeurd, we staan stil vlak ervoor):

```
▼ VARIABLES  
  ▼ Locals  
    getal1 [int]: 56  
    getal2 [int]: 0
```

En via de "call stack" zien we dat we niet gewoon in `MethodeA` zitten, maar dat `MethodeA` is opgeroepen door `Main`:

```
Demonstratie_debugger.dll!Demonstratie_debugger.Program.MethodeA() Line 16 Program...  
Demonstratie_debugger.dll!Demonstratie_debugger.Program.Main(string[] args) Line 21
```

Onderaan staan de methodes die al het langst aan het uitvoeren zijn. Met andere woorden: `Main` heeft `MethodeA` opgeroepen.



Er komen geen oefeningen specifiek rond debuggen. Dat gaat niet, want debuggen helpt je alleen informatie verzamelen. Toch is het een zéér, zéér belangrijke skill. Debuggen kan je helpen inzien **wat je programma doet**. Er worden geen rechtstreekse punten op gegeven, maar als je goed debugt, kan je opdrachten afwerken die je anders niet zou kunnen maken. Gedraagt je programma zich tijdens de labosessies niet zoals je verwacht? **Zet een breakpoint en kijk wat er aan de hand is!**



# Oefeningen

Al deze oefeningen maak je in een klasse `Loops`

## Oefeningen WHILE en DO WHILE

### Oefening: H5-CountDown

#### Leerdoelen

- flowchart omzetten naar code
- gebruik van een `while`-lus

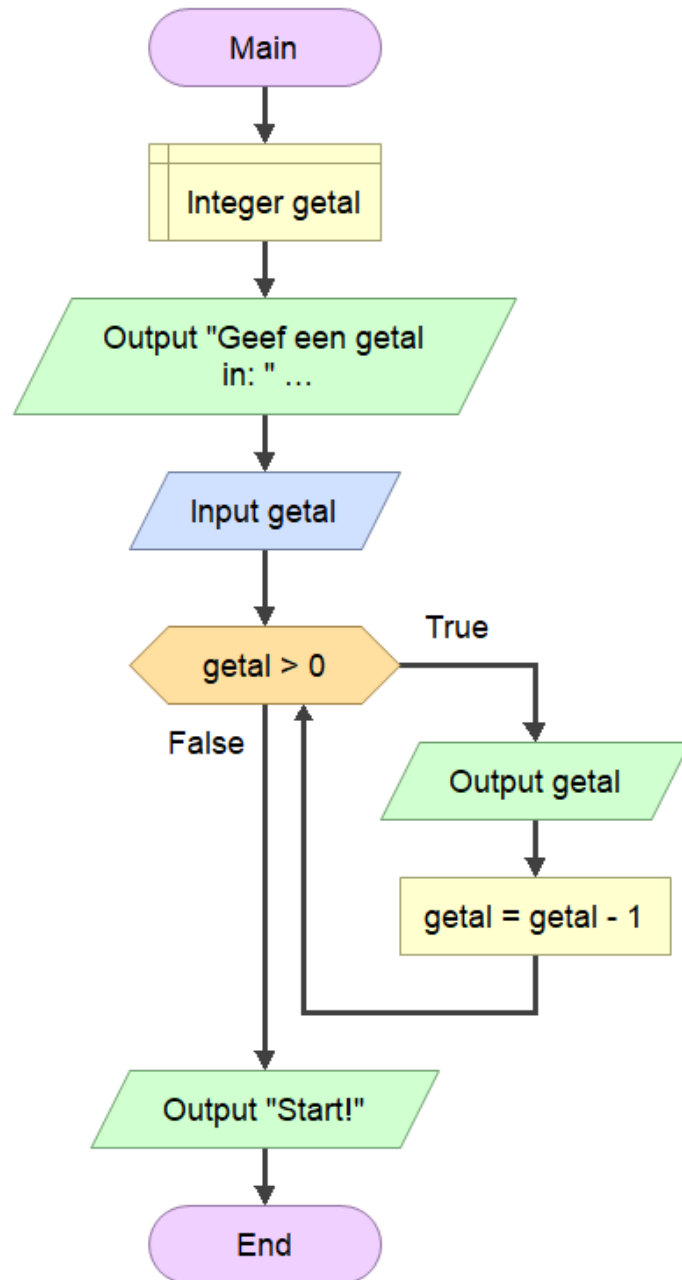
#### Functionele analyse

Je vraagt de gebruiker een positief geheel getal en vervolgens ga je aftellen. Alle getallen vanaf dat getal tot 1 worden getoond onder elkaar. Na het tonen van het laatste getal toon je "Start!".

#### Technische analyse

Maak een methode met de naam `CountDown`

Zet volgende flowchart om in code:



Voorbeeldinteractie(s)

```
Geef een getal in: 10
10
9
8
7
6
5
4
3
2
1
Start!
```

### Testscenario's

Voer een negatief getal in.

## Oefening: H5-Wachtwoord

### Leerdoelen

- Gebruik van een `do while`-lus

### Functionele analyse

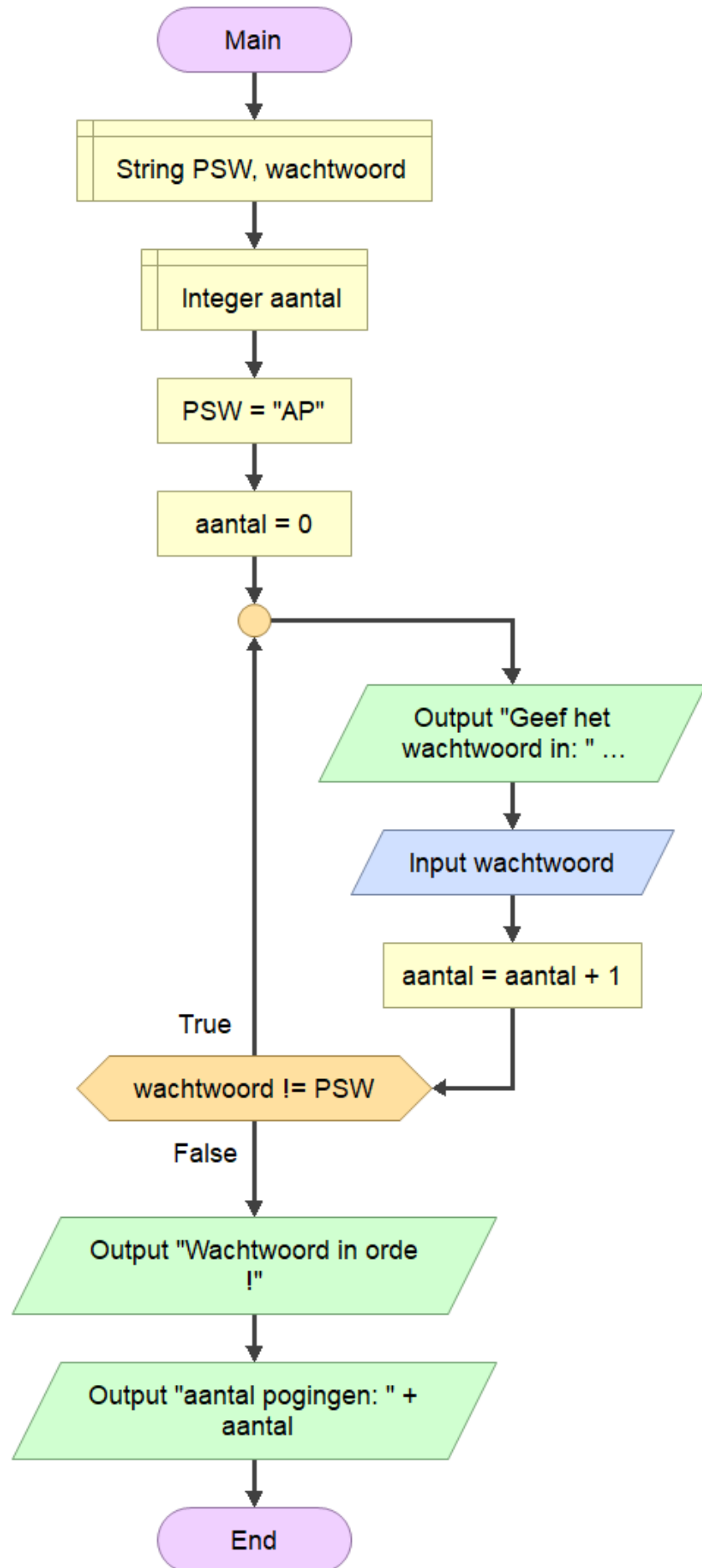
De gebruiker wordt gevraagd om een wachtwoord in te geven totdat het juiste antwoord ("AP") wordt gegeven. Toon daarna "Wachtwoord in orde!".

Tel ook het aantal pogingen dat de gebruiker nodig had om het juiste antwoord te geven en toon het aantal pogingen.

### Technische analyse

Maak een methode met de naam `Wachtwoord`

Zet volgende flowchart om in code:



### Voorbeeldinteractie(s)

```
Geef het wachtwoord in: IT
Geef het wachtwoord in: ap
Geef het wachtwoord in: AP
Wachtwoord in orde!
aantal pogingen: 3
```

### Testscenario's

- Duw meteen op ENTER

## Oefening: H5-Gemiddelde

### Leerdoelen

- Gebruik van een `while` of `do while` lus.

### Functionele analyse

Bereken het gemiddelde van een aantal ingegeven getallen. De invoer van de getallen stopt met ingeven van de waarde 0.

### Technische analyse

Maak een methode met de naam `Gemiddelde`. Het gemiddelde bereken je door de ingegeven getallen te delen door het aantal ingegeven getallen. De 0 die de reeks getallen stopt, wordt niet meegeteld als ingegeven getal.

Werk een oplossing uit met gebruik van een `while` lus of een `do while` lus. Zorg ervoor dat je gemiddelde ook de cijfers na de komma bevat.

### Voorbeeldinteractie(s)

```
Geef het volgende getal in (stoppen met 0) 15
Geef het volgende getal in (stoppen met 0) 8
Geef het volgende getal in (stoppen met 0) 11
Geef het volgende getal in (stoppen met 0) 0
Het gemiddelde: 11,33333333333334
```

### Testscenario's

- Test met negatieve waarden
- Geef dadelijk een 0 in. Wat wordt er dan getoond als waarde van het gemiddelde en wat betekent dit?

## Oefening: H5-Feestje

### Leerdoelen

- Gebruik van een `while` met samengestelde booleaanse expressie

### Functionele analyse

Je organiseert een feestje en met een programma ga je de inschrijvingen noteren. Je kan echter maximaal 20 personen inschrijven én je kan elk moment beslissen dat je geen volgende persoon meer wil inschrijven. Op het einde toon je alle namen van de ingeschrevenen.

### Technische analyse

Maak een methode met de naam `Feestje`

Werk een oplossing uit met gebruik van een while lus.

### Voorbeeldinteractie(s)

```
Wil je een volgende persoon inschrijven? (ja of nee) ja
Geef de naam: John
Wil je een volgende persoon inschrijven? (ja of nee) ja
Geef de naam: Paul
Wil je een volgende persoon inschrijven? (ja of nee) ja
Geef de naam: George
Wil je een volgende persoon inschrijven? (ja of nee) ja
Geef de naam: Ringo
Wil je een volgende persoon inschrijven? (ja of nee) nee
Lijst van aanwezigen: John Paul George Ringo
Er zijn 4 personen aanwezig.
```

### Testscenario's

- Geef dadelijk "nee" in.
- Probeer meer dan 20 personen in te schrijven.

## Oefening: H5-AantalDigits

### Leerdoelen

- Gebruik van een `do while`

### Functionele analyse

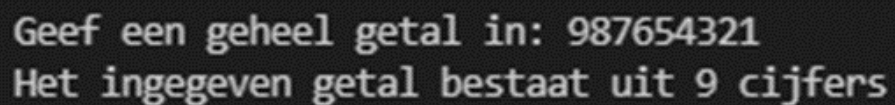
Schrijf een programma dat het aantal cijfers in een getal telt (het getal 12348 heeft bijvoorbeeld 5 cijfers). Het is de bedoeling dat je dit doet met een loop, dus **niet door het getal als tekst te behandelen**.

### Technische analyse

Maak een methode met de naam `AantalCijfers`

Het is de bedoeling dat je het aantal digits telt met een `do while` loop, dus niet door het getal als tekst te behandelen.

### Voorbeeldinteractie(s)



```
Geef een geheel getal in: 987654321
Het ingegeven getal bestaat uit 9 cijfers
```

## Oefening: H5-SomEvenGetallen

### Leerdoelen

- Gebruik van een `while` met een geneste `if`.
- Flowchart omzetten in code

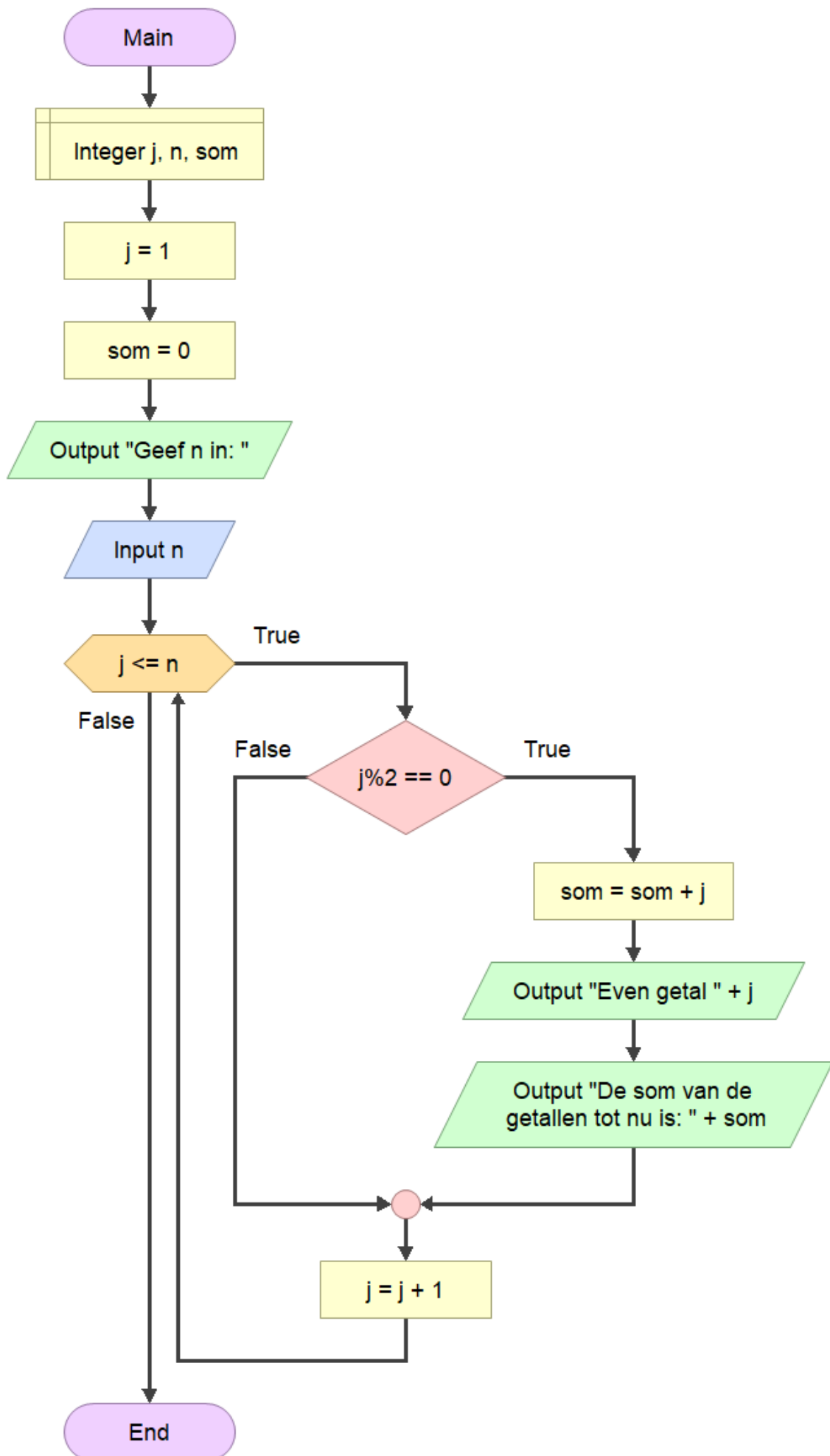
### Functionele analyse

Schrijf een programma dat de som maakt van alle even getallen van 1 tot een waarde `n`. Toon niet enkel de eindsom maar ook alle even getallen met de tussentijdse sommen.

### Technische analyse

Maak een methode met de naam `SomEvenGetallen`

Zet volgende flowchart om in code:





## Voorbeeldinteractie(s)

```
Geef n in: 20
Even getal 2. De som van de getallen tot nu is: 2
Even getal 4. De som van de getallen tot nu is: 6
Even getal 6. De som van de getallen tot nu is: 12
Even getal 8. De som van de getallen tot nu is: 20
Even getal 10. De som van de getallen tot nu is: 30
Even getal 12. De som van de getallen tot nu is: 42
Even getal 14. De som van de getallen tot nu is: 56
Even getal 16. De som van de getallen tot nu is: 72
Even getal 18. De som van de getallen tot nu is: 90
Even getal 20. De som van de getallen tot nu is: 110
```

## Testscenario's

- Druk dadelijk op ENTER.
- Geef 0 in.
- Geef 1 in.

## Oefening: H5-Factoren

### Leerdoelen

- Gebruik van een `while` met geneste `if`'s.

### Functionele analyse

Schrijf een programma een getal  $n$  ontbindt in `factoren`. Factoren zijn de getallen waardoor je  $n$  kan delen zonder rest (van bijvoorbeeld het getal 100 zijn de factoren 1,2,4,5,10,20,25,50,100).

### Technische analyse

Maak een methode met de naam `Factoren`

Gebruik een `while` loop. Gebruik een extra `if` om ervoor te zorgen dat er op het einde van de factoren geen komma teveel staat.

## Voorbeeldinteractie(s)

```
Geef een getal (groter dan 1):
12
Factoren zijn: 1, 2, 3, 4, 6, 12
```

## Testscenario's

- Geef 0 in.
- Geef 1 in.
- Geef een negatief getal in.
- Geef een priemgetal in.

## Oefening: H5-RNA

### Leerdoelen

- Gebruik van een `do while` met geneste `if`.

### Functionele analyse

DNA heeft steeds een RNA-complement (DNA is het gevolg van RNA transcriptie). Schrijf een programma dat een ingevoerde DNA-string omzet naar een RNA-string. De gebruiker voert steeds 1 DNA-nucleotide (m.a.w. één letter) in per keer en duwt op enter, de RNA string wordt steeds groter. De omzetting is als volgt:

- G wordt C
- C wordt G
- T wordt A
- A wordt U
- “stop” dan stopt de gebruiker met letters ingeven en wordt het resultaat getoond.
- Andere letters of meer dan één letter worden genegeerd

### Technische analyse

Noem de methode voor deze oefening `RNATranscriptie`.

Voorbeeldinteractie(s)

```
Geef de letter in (stoppen met 'stop')
G
Geef de letter in (stoppen met 'stop')
C
Geef de letter in (stoppen met 'stop')
T
Geef de letter in (stoppen met 'stop')
A
Geef de letter in (stoppen met 'stop')
jkjfhskj
Geef de letter in (stoppen met 'stop')
G
Geef de letter in (stoppen met 'stop')
stop
De RNA string is: CGAUC
```

### Testscenario's

- Geef andere letters in dan G, T, C of A.
- Geef meerdere letters in.
- Geef een getal in.
- Druk op ENTER.

## Oefening: H5-boekhouder

### Leerdoelen

- Gebruik van een `do while` met geneste `if`
- Gebruik van een oneindige lus

### Functionele analyse

Maak een 'boekhoudprogramma': de gebruiker kan continu positieve en negatieve getallen invoeren. Dit programma houdt volgende zaken bij:

- de totale balans
- de som van de positieve getallen
- de som van de negatieve getallen
- het gemiddelde

### Technische analyse

Maak een methode met de naam `Boekhouder`

Voor de eerste drie zaken kom je toe met een variabele. Voor de laatste is dit lastiger, omdat elk nieuw getal een kleiner effect heeft op het gemiddelde dan het vorige. Je houdt beter een teller bij met het aantal ingevoerde getallen. Dan is het gemiddelde de totale balans gedeeld door het aantal ingevoerde getallen.

voorbeeldinteractie(s)

```
Geef een getal
7
De balans is 7
De som van de positieve getallen is 7
De som van de negatieve getallen is 0
Het gemiddelde is 7
Geef een getal
-3
De balans is 4
De som van de positieve getallen is 7
De som van de negatieve getallen is -3
Het gemiddelde is 2
Geef een getal
11
De balans is 15
De som van de positieve getallen is 18
De som van de negatieve getallen is -3
Het gemiddelde is 5
Geef een getal
```

(Dit programma kan blijven verder lopen zo lang je wil.)

### Testscenario's

- Druk dadelijke op ENTER.

## Oefeningen FOR

### Oefening: H5-VanMin100Tot100

#### Leerdoelen

- Gebruiken van een `for`-lus
- Aanpassen 'update'
- Flowchart omzetten in code

#### Functionele analyse

Basis: Toon alle natuurlijke getallen van -100 tot 100.

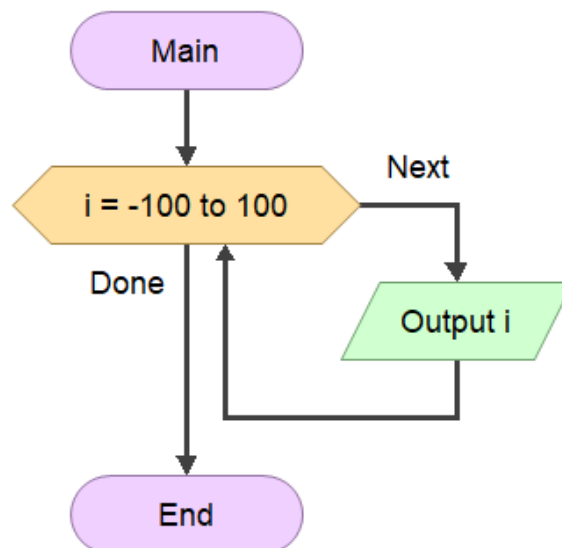
Uitbreiding: Toon alle even natuurlijke getallen van -100 tot 100.

### Technische analyse

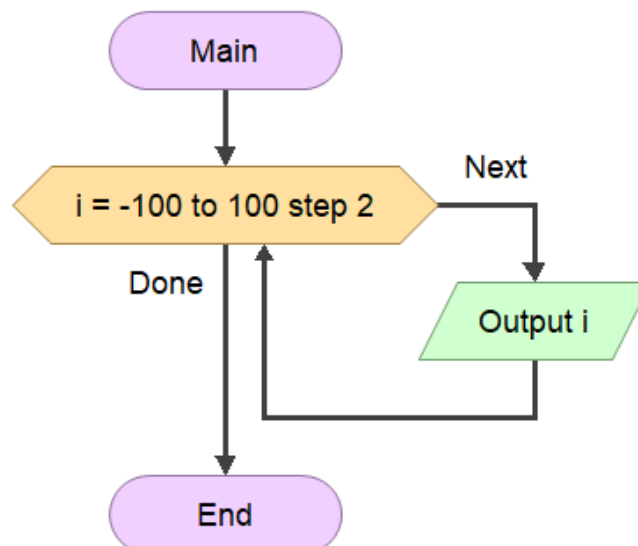
Maak een methode met de naam `VanMin100Tot100`.

Zet volgende flowcharts om in code: (als er geen update vermeld is, wordt de wachtervariabele verhoogd met 1)

**Basis:**



**Uitbreiding:**



**voorbeeldinteractie(s)**

**Basis**

```
-100
-99
-98
-97
-96
-95
```

...

```
96
97
98
99
100
```

Uitbreiding:

```
-100
-98
-96
-94
-92
```

...

```
96
98
100
```

## Oefening: H5-EenTafel

### Leerdoelen

- Gebruiken van een `for`-lus

### Functionele analyse

Vraag aan de gebruiker van welk getal de tafel van vermenigvuldiging tot 10 moet getoond worden.  
Toon elke vermenigvuldiging onder elkaar.

## Technische analyse

Maak een methode met de naam `EenTafel` .

voorbeeldinteractie(s)

```
Van welk getal wil je de tafel van vermenigvuldiging zien  
5  
1 x 5 is 5  
2 x 5 is 10  
3 x 5 is 15  
4 x 5 is 20  
5 x 5 is 25  
6 x 5 is 30  
7 x 5 is 35  
8 x 5 is 40  
9 x 5 is 45  
10 x 5 is 50
```

Testscenario's

Geef 0 in.

## Oefening: H5-Vee1vouden6En8

Leerdoelen

- Gebruiken van een `for` -lus met een geneste `if` met een samengestelde booleaanse expressie

## Functionele analyse

Toon alle getallen van 1 tot en met 100 die een veelvoud zijn 6 en die een veelvoud zijn van 8.

## Technische analyse

Maak een methode met de naam `Vee1vouden6En8` .

voorbeeldinteractie(s)

6  
8  
12  
16  
18  
24  
30  
32  
36  
40  
42  
48  
54  
56  
60  
64  
66  
72  
78  
80  
84  
88  
90  
96

## Oefening: H5-Priemchecker

### Leerdoelen

- Gebruiken van een `for`-lus met geneste `if`.

### Functionele analyse



Je krijgt een getal van de gebruiker. Je moet nagaan of dit een priemgetal is, d.w.z. of het precies 2 gehele delers heeft.

### Technische analyse

Maak een methode met de naam `PriemChecker`.

Elk geheel getal vanaf 2 heeft minstens 2 gehele delers: 1 en zichzelf. Als dat de enige delers van het gegeven getal zijn, is het priem. Je kan dus nagaan of een getal een priemgetal is door alle getallen vanaf 2 tot het getal zelf te overlopen en na te gaan of deze delers zijn van het getal.

Je mag veronderstellen dat de gebruiker minstens 2 intypt.

### voorbeeldinteractie(s)

```
Geef getal in:  
8  
8 is geen priemgetal
```

```
Geef getal in:  
11  
11 is een priemgetal
```

### Testscenario's

Test met een negatief getal.

## Oefening: H5-PriemGenerator

### Leerdoelen

- Gebruiken van een `while` lus met een geneste `for` lus

### Functionele analyse

Je toont de priemgetallen tussen een laagste waarde en een hoogste waarde die door de gebruiker worden ingegeven.

### Technische analyse

Maak een methode met de naam `PriemGenerator`.

Je kan een deel van de code van de vorige oefening gebruiken

### voorbeeldinteractie(s)

```
Priemgetallen van (laagste getal):  
2  
tot en met (hoogste getal):  
10  
2 3 5 7
```

### Testscenario's

Test met een negatieve laagste waarde.