C# Essentials - Herhalingsstructuren

Inhoudsopgave

1. Introductie

1.1. Wat is een loop?

Een loop is een **stuk code dat meerdere keren wordt uitgevoerd**. Dit kan handig zijn als je iets wil herhalen, zoals:

- Meerdere getallen afdrukken
- Spelrondes laten doorgaan tot iemand wint
- Herhaaldelijk vragen naar invoer tot deze geldig is

1.2. Waarom loops gebruiken?

Stel dat je iets 10 keer wil afdrukken. Je zou dan 10 keer dezelfde `Console.WriteLine()` moeten schrijven. Maar dat is:

- **Traag**
- **Foutgevoelig**
- **Moeilijk aanpasbaar**

Met een loop schrijf je die instructie maar één keer, en zeg je gewoon: “Herhaal dit 10 keer.”

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Hallo!");
}
```

1.3. Verschillende soorten loops

In C# heb je verschillende soorten loops, elk met hun eigen situatie:

Type	Gebruik
`for`	Als je vooraf weet **hoe vaak** je iets wil herhalen
`while`	Als je wil herhalen **zolang een voorwaarde waar is**
`do while`	Zoals `while`, maar de code wordt **minstens één keer** uitgevoerd
`foreach`	Als je alle elementen van een collection wil "doorlopen"

De volgende hoofdstukken leggen elk loop-type apart uit, met duidelijke voorbeelden en oefeningen.

1.4. Belangrijke termen

- **Iteratie**: één herhaling in een loop
- **Loop body**: de code binnen de accolades `{ }`
- **Voorwaarde**: bepaalt of de loop doorgaat of stopt

1.5. In het kort

- Een loop laat je toe om een stuk code meerdere keren uit te voeren.
- Je voorkomt herhaling van code en maakt je programma **korter, leesbaarder en flexibeler**.
- In C# gebruik je vooral `for`, `while` en `do while`.

2. De `for`-lus

2.1. Wat is een `for`-lus?

Een `for`-lus gebruik je wanneer je **op voorhand weet hoe vaak** je iets wil herhalen. Bijvoorbeeld: 10 keer iets afdrukken, een reeks getallen optellen, of door een lijst tellen met een teller.

2.2. Opbouw van een `for`-lus

```
for (beginwaarde; voorwaarde; stapgrootte)
{
    // Code die je wil herhalen
}
```

Betekenis van de onderdelen:

- `beginwaarde`: wordt 1 keer uitgevoerd bij de start
- `voorwaarde`: zolang deze waar is, blijft de loop doorgaan
- `stapgrootte`: wordt uitgevoerd na elke iteratie (meestal `i++`)

2.3. Voorbeeld 1 - optellen

```
Console.WriteLine("Optellen van 0 t.e.m. 9:");
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

Dit toont de getallen 0 tot en met 9 in de console. De lus stopt zodra `i < 10` niet meer waar is.

2.4. Voorbeeld 2 – aftellen

```
Console.WriteLine("Aftellen van 5 naar 0:");
for (int i = 5; i >= 0; i--)
{
    Console.WriteLine(i);
}
```

Hier telt de `for`-lus af van 5 naar 0 met telkens `i--`.

```
Aftellen van 5 naar 0:
5
4
3
2
1
0
```

2.5. Veelvoorkomende fouten

- `i <= 10` in plaats van `< 10` — opletten dat je niet per ongeluk één extra stap doet
- **Stapgrootte vergeten** — zorgt voor oneindige loop of foutmelding
- **Verkeerde voorwaarde** — kan maken dat de loop nooit start

2.6. Praktisch gebruik

Een `for`-lus wordt vaak gebruikt bij: - Aftellen of optellen in een vaste reeks - Herhalen van instructies **een exact aantal keren** - Simpele rekenkundige taken

2.7. In het kort

- Gebruik een `for`-lus als je weet **hoeveel stappen** je wil doorlopen
- De structuur bevat 3 delen: start, voorwaarde, stapgrootte
- Het is de meest gebruikte lus voor eenvoudige iteraties

3. De `while`-lus

3.1. Wat is een `while`-lus?

Een `while`-lus voert code uit **zolang een voorwaarde true is**. De voorwaarde wordt **voor elke iteratie gecontroleerd**.

3.2. Basisvorm

```
while (voorwaarde)
{
    // code die je wil herhalen
}
```

Als de voorwaarde vanaf het begin `false` is, dan wordt de lus **nooit** uitgevoerd.

3.3. Voorbeeld

```
int number = 0;

while (number < 5)
{
    Console.WriteLine($"Waarde: {number}");
    number++;
}
```

Zodra `number` gelijk is aan 5, stopt de lus.

```
Waarde: 0
Waarde: 1
Waarde: 2
Waarde: 3
Waarde: 4
```

3.4. Typische toepassingen

Een `while`-lus gebruik je als:

- je moet **blijven vragen naar input** tot deze geldig is
- je programma **moet blijven lopen** zolang iets nog actief is
- je niet op voorhand weet hoe vaak de herhaling nodig is

3.5. Veelvoorkomende fouten

- De **voorwaarde verandert niet** → oneindige lus
- De **initiële waarde** klopt niet → lus start niet
- De **lus bevat geen inhoud** → niets zichtbaar

Voorbeeld van een oneindige lus:

```
int number = 0;

while (number < 5)
{
    Console.WriteLine("Hallo!");
    // Oeps! number++ vergeten
}
```

Dit blijft oneindig doorgaan.

3.6. In het kort

- Een `while`-lus herhaalt code **zolang een voorwaarde true is**
- De voorwaarde wordt **voor elke herhaling** gecontroleerd
- Geschikt voor situaties waar je het **aantal herhalingen niet op voorhand weet**
- Let op dat de **voorwaarde op een bepaald moment false wordt**, anders loopt de lus eindeloos

4. De `do while`-lus

4.1. Wat is een `do while`?

Een `do while`-lus werkt bijna zoals een `while`, met één belangrijk verschil: - **De voorwaarde wordt pas gecontroleerd na de eerste uitvoering**

Je bent dus zeker dat de code **minstens één keer** wordt uitgevoerd.

4.2. Basisvorm

```
do
{
    // code die je wil herhalen
}
while (voorwaarde);
```

Let op het puntkomma `;` na de `while` — die hoort er bij!

4.3. Voorbeeld

```
int number = 1;

do
{
    number *= 2;
    Console.WriteLine(number);
}
while (number < 100);
```

Deze code vermenigvuldigt `number` telkens met 2, tot de waarde 100 of meer wordt.

```
2
4
8
16
32
64
128
```

4.4. Belangrijk verschil met `while`

```
int x = 200;

while (x < 100)
{
    Console.WriteLine("while wordt niet uitgevoerd");
}

do
{
```

```
    Console.WriteLine("do while wordt WEL uitgevoerd");
}
while (x < 100);
```

- De `while`-lus wordt overgeslagen omdat de voorwaarde vanaf het begin false is.
- De `do while`-lus voert de code één keer uit voordat de voorwaarde wordt gecontroleerd.

4.5. Wanneer gebruik je `do while`?

- Als je zeker weet dat de actie **minstens één keer** moet worden uitgevoerd
- Bij menu's of inputdialogen die altijd getoond moeten worden
- Als je **eerst iets wil doen en pas daarna beslissen of je verdergaat**

4.6. In het kort

- Een `do while`-lus is bijna hetzelfde als een `while`-lus
- Het verschil: de code wordt **eerst uitgevoerd, dan pas gecontroleerd**
- Altijd goed voor situaties waarbij de actie **minstens één keer** moet gebeuren

5. Geneste loops (nesting)

5.1. Wat is nesting?

Een geneste loop is een **lus binnen een lus**. De binnenste lus wordt telkens **volledig uitgevoerd** bij elke iteratie van de buitenste lus.

5.2. Voorbeeld – 10x10 vierkant

```
Console.WriteLine("Print een 10x10 vierkant:");

for (int i = 0; i < 10; i++) // buitenste lus - rijen
{
    for (int j = 0; j < 10; j++) // binnenste lus - kolommen
    {
        Console.Write("* ");
    }
    Console.WriteLine(); // nieuwe regel
}
```

Uitvoer:

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

5.3. Hoe werkt dit?

- De **buitenste** `for` bepaalt hoeveel rijen er zijn.
- De **binnenste** `for` print telkens 10 sterretjes.
- Na elke binnenste loop volgt een `Console.WriteLine()` om een nieuwe regel te beginnen.

5.4. Wanneer gebruik je geneste loops?

- Bij het afdrukken van tabellen, rasters of patronen
- Bij games met een bord (bv. schaak, sudoku)
- Als je met meerdere niveaus van data werkt (bv. een lijst in een lijst)

5.5. Let op leesbaarheid

Geneste loops kunnen snel onoverzichtelijk worden. Gebruik duidelijke variabelen (`i`, `j`, of betere namen als `row`, `col`) en zorg voor juiste inspringing.

5.6. In het kort

- Een geneste loop is een lus binnen een lus
- De binnenste loop wordt telkens volledig uitgevoerd bij elke iteratie van de buitenste
- Handig voor tabellen, patronen en dubbele structuren
- Hou de code overzichtelijk en vermijd té diepe nesting

6. Valkuilen bij loops

6.1. Oneindige lus

6.1.1. Wat is een oneindige lus

Een loop die **nooit stopt**, noemen we een oneindige lus. Dit gebeurt als de **voorwaarde altijd `true` blijft** of als de **waarde nooit aangepast wordt**.

```
int i = 0;

while (i < 5)
{
    Console.WriteLine(i);
    // i++ vergeten → loopt eindeloos!
}
```

Of:

```
for (int i = 10; i >= 10; i++)
{
    Console.WriteLine(i);
}
// Voorwaarde blijft altijd true → eindeloze lus
```

6.1.2. Hoe voorkom je dit?

- Controleer altijd of de **voorwaarde ooit false kan worden**
- Zorg dat de **lusvariabele aangepast wordt** binnen de loop
- Test loops met kleine waarden om fouten sneller op te merken

6.1.3. Hoe stop je een oneindige lus?

In een console-applicatie:

- Druk op `Ctrl + C` om de applicatie geforceerd te stoppen

In Visual Studio:

- Klik op de **"Stop Debugging"**-knop
- Of gebruik **Shift + F5**

6.2. Andere valkuilen

- **Voorwaarde altijd false** → loop wordt nooit uitgevoerd
- **Verkeerde richting in `for`** → bv. `i++` in plaats van `i--`
- **Stapgrootte verkeerd** → bv. `i += 2` gebruiken waar je elke stap wil

- **Onnodig diepe nesting** → moeilijk leesbaar en foutgevoelig

6.3. In het kort

- Oneindige loops zijn vaak het gevolg van een fout in de voorwaarde of update
- Ze kunnen je programma vast laten lopen
- Test altijd je loopcondities grondig en gebruik `breakpoints` of `debugging` bij twijfel

7. Break en Continue

7.1. break

`break` betekent: **"Stop onmiddellijk met de loop."** Het programma springt dan naar de **eerste regel ná de lus**. Alle resterende herhalingen worden niet meer uitgevoerd.

Je gebruikt `break` meestal als:

- je een bepaald resultaat gevonden hebt,
- of als het niet meer zinvol is om verder te zoeken.

7.1.1. Voorbeeld

```
static void Main(string[] args)
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 5)
        {
            break;
        }

        Console.WriteLine(i);
    }
}
```

Resultaat:

```
1
2
3
4
```

De `for`-lus stopt zodra `i == 5`, door het gebruik van `break`. De rest van de iteraties wordt overgeslagen.

7.2. continue

`continue` betekent: **"Sla de rest van de huidige herhaling over."** Het programma springt dan onmiddellijk terug naar **het begin van de lus**, om te kijken of er nog een volgende herhaling moet komen.

Je gebruikt `continue` meestal als:

- je een bepaalde voorwaarde wil overslaan,
- maar de rest van de iteraties nog wil blijven uitvoeren.

7.2.1. Voorbeeld

```
static void Main(string[] args)
{
    for (int i = 1; i <= 5; i++)
    {
        if (i == 3)
        {
            continue;
        }

        Console.WriteLine(i);
    }
}
```

Resultaat:

```
1
2
4
5
```

De waarde `3` wordt overgeslagen omdat `continue` het printen verhindert voor die specifieke iteratie.

7.3. Samenvatting

Instructie	Betekenis
`break`	Stopt de loop volledig en springt naar de eerste code ná de loop
`continue`	Slaat de rest van de huidige herhaling over en start de volgende

7.4. Oefening

Wat is de uitvoer van onderstaande code?

```
static void Main(string[] args)
{
    for (int i = 0; i < 6; i++)
    {
        if (i % 2 == 0)
        {
            continue;
        }

        if (i == 5)
        {
            break;
```

```
        }

        Console.WriteLine(i);
    }
}
```

Oplossing

```
1
3
```

`i = 0, 2, 4` worden overgeslagen door `continue`. Bij `i = 5` breekt de loop af door `break`.

7.5. Praktische voorbeelden

7.5.1. Eerste getal groter dan 20 (`break`)

We zoeken het **eerste getal groter dan 20**. Zodra we het vinden, stoppen we met zoeken via `break`.

```
static void Main(string[] args)
{
    for (int i = 10; i <= 30; i += 3)
    {
        if (i > 20)
        {
            Console.WriteLine($"Eerste getal boven 20: {i}");
            break;
        }
    }
}
```

Resultaat:

```
Eerste getal boven 20: 22
```

7.5.2. Getallen kleiner dan 10 overslaan (`continue`)

We willen enkel getallen tonen die **10 of meer** zijn. De rest wordt overgeslagen met `continue`.

```
static void Main(string[] args)
{
    for (int i = 5; i <= 15; i++)
    {
        if (i < 10)
        {
            continue;
        }

        Console.WriteLine($"Geldig getal: {i}");
```

```
    }
}
```

Resultaat:

```
Geldig getal: 10
Geldig getal: 11
Geldig getal: 12
Geldig getal: 13
Geldig getal: 14
Geldig getal: 15
```

8. Scope van variabelen

8.1. Herhaling: wat is scope?

Scope betekent het bereik waarin een variabele bestaat. Je hebt eerder geleerd dat een variabele enkel geldig is **binnen het blok** `{ }` waarin ze werd aangemaakt.

8.2. Scope in een `for`-lus

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(i); // OK
    }

    // Console.WriteLine(i); // FOUT: i bestaat enkel binnen de for-lus
}
```

De variabele `i` werd **binnen** de haakjes van de `for`-lus gedeclareerd. Ze is dus enkel beschikbaar in dat blok.

8.3. Scope in een `while`-lus

```
static void Main(string[] args)
{
    int counter = 0;

    while (counter < 3)
    {
        string message = $"Teller is {counter}";
        Console.WriteLine(message);
        counter++;
    }

    // Console.WriteLine(message); // FOUT: message bestaat enkel in de while-
lus
}
```

Variabelen zoals `message` die je binnen de `while` aanmaakt, verdwijnen zodra je buiten de accolades gaat.

8.4. Scope uitbreiden door buiten de lus te declareren

Soms wil je een variabele ook **na** de loop nog gebruiken. Dan moet je ze **voor** de loop declareren.

```
static void Main(string[] args)
{
```

```
    int sum = 0;

    for (int i = 1; i <= 3; i++)
    {
        sum += i;
    }

    Console.WriteLine($"Totaal: {sum}"); // OK
}
```

8.5. Samenvatting

Regel	Uitleg
Variabelen in een lus bestaan enkel binnen die lus	Je krijgt een fout als je ze daarbuiten probeert te gebruiken
Als je een variabele ook buiten de lus nodig hebt	Declareer ze vóór de lus begint
Elke lus (`for`, `while`, `do`) heeft zijn eigen scope-blok `{ }`	Die bepaalt waar variabelen geldig zijn