Gevorderde klasseconcepten

Zie Scherp Scherper - Hoofdstuk 11

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- een **constructor** schrijven om de beginstaat van een object in te stellen
- het verschil tussen een **default** en een **overloaded** constructor uitleggen
- constructors hergebruiken met `this()`
- objecten opzetten met **object initializer syntax** en `required`
- het keyword `static` gebruiken voor fields, methoden, properties en klassen
- uitleggen waarom een `static` deel niet bij de niet-static delen kan

Waar staan we?

Een klasse bestond tot nu uit:

- **instantievariabelen** (de toestand)
- **methoden** (het gedrag)
- **properties** (gecontroleerde toegang)

Daar komen nu bij:

- **constructors**, `static` en **object initializer syntax**

De new operator

```
1  Student frank = new Student();
```

`new` doet **drie** dingen:

1. een object aanmaken in de **heap**
2. de **constructor** aanroepen voor extra initialisatie
3. een **referentie** naar dat object teruggeven

> Vandaar de ronde haakjes bij `new Student()`: dat is de constructor-aanroep.

Soorten constructors

- **Geen** zelf geschreven constructor: je krijgt gratis een onzichtbare **default** constructor
- Enkel een **default** constructor (zonder parameters), maar zelf ingevuld
- Een of meer **overloaded** constructors met parameters, bv. `new Student("Jos")`

> ⚠ **Waarschuwing**
>
> Net als met aannemers: soms ruimen ze zelf op, soms niet. Schrijf je **zelf** een constructor, dan ben je de **gratis** default constructor kwijt.

Een default constructor

```
1  internal class Student
2  {
3      public Student()
4      {
5          UurVanInschrijven = DateTime.Now.Hour;
6      }
7      public int UurVanInschrijven { get; private set; }
8  }
```

- Altijd **`public`**, **geen** returntype, naam = **klassenaam**, geen parameters

> **Tip**
>
> Enkel een auto-property een startwaarde geven? Dan volstaat `public int X { get; set; } = 2;` zonder constructor.

Overloaded constructors

```
1  public Student(string bijnaamIn)
2  {
3      BijNaam = bijnaamIn;
4  }
```

Zo kan je `new Student("Lord Oakenwood")` schrijven.

> ⚠ **Waarschuwing**
>
> Vanaf nu werkt `new Student()` **niet** meer: *"Student does not contain a constructor that takes 0 arguments"*.
> Wil je de default constructor terug? Schrijf hem dan zelf.

this(): constructors hergebruiken

Vermijd dubbele code door één constructor de andere te laten aanroepen:

```
1  public Microfoon(string merkIn, bool isUitverkochtIn)
2  {
3      Merk = merkIn;
4      IsUitverkocht = isUitverkochtIn;
5  }
6
7  public Microfoon(string merkIn) : this(merkIn, false) { }
8
9  public Microfoon() : this("Onbekend", true) { }
```

De compiler kiest via **overload resolution** de juiste constructor.

Welke constructors voorzien?

Minstens genoeg om een object in een **geldige** staat te brengen:

```
1  public Breuk(int tellerIn, int noemerIn)
2  {
3      Teller = tellerIn;
4      Noemer = noemerIn;   // set weigert noemer 0
5  }
```

Tip

Door enkel deze constructor te voorzien, dwing je af dat niemand een `Breuk()` zonder noemer maakt (en zo een `DivideByZeroException` riskeert).

Object initializer syntax

Geef bij de creatie properties een beginwaarde, **zonder** een aparte constructor:

```
1  Meting m = new Meting { Temperatuur = 3.4, IsGeconfirmeerd = true };
```

- Werkt via de **default** constructor en daarna de **properties**
- Enkel voor properties met een van buitenaf bereikbare `set`

> **Opmerking**
>
> Intern: eerst de default constructor, dan `m.Temperatuur = 3.4;` enzovoort.

required properties

Wil je toch verplichten dat een property wordt ingesteld bij object initializer syntax? Gebruik **`required`**:

```
internal class Meting
{
    public double Temperatuur { get; set; }
    public required bool IsGeconfirmeerd { get; set; }
}
```

```
Meting m = new Meting { Temperatuur = 0.7 }; // FOUT: IsGeconfirmeerd ontbreekt
```

> **Belangrijk**
>
> `required` bestaat sinds C# 11 (.NET 7+).

static: het idee

> **`static`** geeft aan dat een element bij de **klasse zelf** hoort, niet bij een individueel object.

Gevolg: je kan het aanroepen **zonder** een object aan te maken, en de data wordt **gedeeld** over alle objecten.

`static` kan op fields, methoden, properties en zelfs de hele klasse.

static fields

```
internal class Mens
{
    private static int teller = 0;
    public Mens() { teller++; }
}
```

Eén exemplaar, **gedeeld** door alle objecten: maak je drie `Mens`-objecten, dan staat `teller` op 3.

> ⚠ **Waarschuwing**
>
> Gebruik `static` met mate: het druist vaak in tegen OOP. Het past bij zaken die bij de **klasse** horen (een teller), niet bij een individueel object (een geboortejaar).

static methoden en klassen

Daarom kan je `Math.Pow(3, 2)` zonder een `Math`-object te maken:

```
1  internal static class EpicLibrary
2  {
3      public static void ToonInfo() { Console.WriteLine("Ik ben ik"); }
4      public static int TelOp(int a, int b) { return a + b; }
5  }
6
7  EpicLibrary.ToonInfo();
8  int som = EpicLibrary.TelOp(3, 5);
```

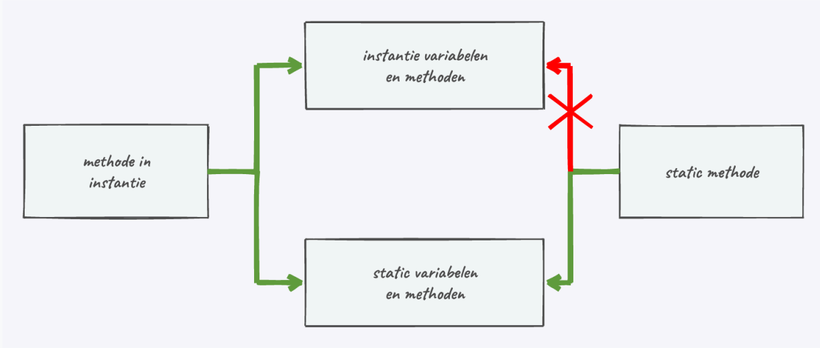
Belangrijk

Een `static class` mag **enkel** static members bevatten. De compiler dwingt dat af: je kan er toch nooit een object van maken.

static versus non-static

Een `static` methode kan **enkel** andere `static` leden aanspreken: ze hoort bij geen enkel object.

```
1  private int gewicht = 50;
2  private static void Verminder()
3  {
4      gewicht--; // FOUT
5  }
```



Wat mag wel, wat niet.

> **Tip**
>
> Daarom moet in `Program.cs` alles `static` zijn: `Main` is zelf `static`.

static properties

Alle balletjes delen dezelfde speelveldgrenzen:

```
static public int Breedte { get; set; }
static public int Hoogte { get; set; }
```

```
Balletje.Hoogte = Console.WindowHeight;
Balletje.Breedte = Console.WindowWidth;
```

Elk balletje gebruikt in `Update()` gewoon `Balletje.Breedte`, zonder de grenzen zelf te kennen.

static en Random

Een `Random` per object is een **slecht idee**: objecten die snel na elkaar ontstaan, krijgen dezelfde seed en dus dezelfde getallen.

```
1  internal class Dobbelsteen
2  {
3      static Random gen = new Random(); // gedeeld
4      public int Werp() { return gen.Next(1, 7); }
5  }
```

> **Tip**
>
> Eén gedeelde, `static` generator lost dat netjes op.

Conclusie

- De **constructor** zet een object meteen in een geldige beginstaat (via `new`)
- Schrijf je zelf een constructor, dan ben je de **gratis** default constructor kwijt
- `this()` hergebruikt constructors; **object initializer syntax** en `required` vullen properties in
- `static` hoort bij de **klasse**, niet bij een object: gedeelde data en hulpmethoden
- Een `static` deel kan enkel andere `static` delen aanspreken

> **Tip**
>
> Volgende halte: **arrays en collecties van objecten** (`List`, `foreach`, `Dictionary`).

Meer weten

Kennisclips

- Constructors
- Overloaded constructors en this
- Object initialize Syntax
- static keyword
- Nugets en libraries
- Eigen class library maken

Oefeningen H11 · 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: constructor in andere talen

In **Python** heet de constructor altijd `__init__` (eerste parameter `self`):

```python
class Auto:
    def __init__(self, merk):
        self.merk = merk
```

In **JavaScript** heet hij gewoon `constructor`:

```javascript
class Auto {
    constructor(merk) { this.merk = merk; }
}
```

Zie verder: meerdere constructors

C#, **Java** en **C++** laten meerdere constructors toe (overloading). **Python** kent dat niet: één `__init__`, en je lost het op met default-parameterwaarden:

```
class Student:
    def __init__(self, bijnaam, is_werkstudent=False):
        self.bijnaam = bijnaam
        self.is_werkstudent = is_werkstudent
```

Zowel `Student("Jos")` als `Student("Jos", True)` werkt: één constructor die zich als meerdere gedraagt.

Zie verder: static in andere talen

In **Java** werkt `static` zo goed als identiek aan C#:

```java
class Fiets {
    static int aantalFietsen = 0;
    static int getAantal() { return aantalFietsen; }
}
// Fiets.getAantal();
```

In **Python** is een variabele rechtstreeks in de klasse (zonder `self`) gedeeld, en `@staticmethod` maakt een methode die je op de klasse aanroept:

```python
class Fiets:
    aantal_fietsen = 0
    @staticmethod
    def beschrijf(): print("Een fiets")
# Fiets.beschrijf()
```

In alle gevallen hoort dit bij de *klasse*, niet bij een los object.

Samenvattende poster

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