



// 01

Constructor en new

`new` maakt een object aan op de **heap**, roept de **constructor** aan en geeft een referentie naar het object terug.

```
public Student() // default constructor
{
    UurVanInschrijven = DateTime.Now.Hour;
}
```

- Zelfde **naam als de klasse**, **public**, **geen returntype** (ook geen **void**)
- Loopt één keer: bij de geboorte van het object, met `new`
- Enkel een startwaarde voor een auto-property? Dat kan ook zonder constructor: `{ get; private set; } = 2;`

// 02

Default en overloaded

Schrijf je geen constructor, dan krijg je een **gratis default constructor**: alle instantievariabelen en properties krijgen hun defaultwaarde. Een **overloaded constructor** heeft parameters, net als bij method overloading.

```
public Student() // default
{
    BijNaam = "Geen";
}
public Student(string bijnaamIn) // overloaded
{
    BijNaam = bijnaamIn;
}
```

Aanroepen: `new Student()` of `new Student("Lord Oakenwood")`.

// LET OP

Schrijf je zelf een constructor, dan ben je de gratis default constructor kwijt en compileert `new Student()` niet meer (*does not contain a constructor that takes 0 arguments*). Nodig? Schrijf hem zelf bij.

// 03

Een geldig object

Heeft **Breuk** enkel deze constructor, dan kan niemand nog een breuk zonder noemer maken, zoals `new Breuk()`.

```
public Breuk(int tellerIn, int noemerIn)
{
    Teller = tellerIn;
    Noemer = noemerIn; // via de property
}
```

Gebruik in een constructor de **properties**, niet de instantievariabele erachter: dan doet de controle in de `set` meteen haar werk (de `set` van `Noemer` weigert 0).

// 04

Constructors hergebruiken: this(...)

Met `: this(...)` geeft een constructor zijn werk door aan een andere. Zo stelt enkel `Microfoon(string, bool)` zelf `Merk` en `IsUitverkocht` in, zonder dubbele code:

```
public Microfoon(string merkIn)
: this(merkIn, false) { }
public Microfoon()
: this("Onbekend", true) { }
```

De compiler kiest welke, op aantal en type van de parameters. Die loopt eerst, daarna de body van de constructor zelf.

// 05

Object initializer syntax

Geef **properties** een beginwaarde, zonder extra constructor:

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

Eerst loopt de default constructor, dan krijgen de properties hun waarde. De `()` na `new Meting` mag weg. Niet elke property hoeft erin, tenzij er **required** voor staat (C# 11, .NET 7):

```
public required bool IsGeconfirmeerd { get; set; }
```

// LET OP

Werkt enkel met properties waarvan de `set` van buitenaf bereikbaar is. Een `private set` kan je er niet mee instellen.

// 06

static: bij de klasse

Een **static** element hoort bij de **klasse zelf**, niet bij één object. Er is één exemplaar, en alle objecten delen het.

```
internal class Fiets
{
    private static int aantalFietsen = 0;
    public Fiets()
    {
        aantalFietsen++;
    }
}
```

Maak je drie fietsen, dan staat `aantalFietsen` op 3. Zonder **static** krijgt elke fiets een eigen teller, die op 1 blijft staan. Eén `Random` voor alle objecten:

```
static Random gen = new Random();
```

// 07

Static methoden, properties, klassen

Een static lid roep je op de **klassenaam** aan, zonder object: `Math.Pow(3, 2)`, `EpicLibrary.TelOp(3, 5)`. Een **static class** mag enkel static leden bevatten; `new Math()` kan niet.

```
internal static class EpicLibrary
{
    public static int TelOp(int a, int b)
    {
        return a + b;
    }
}
```

Een gewone klasse mag mengen: `DateTime.Now` is static.

// 08

Static tegenover non-static

VANUIT	INSTANTIELEDEN	STATIC LEDEN
een gewone methode	✓	✓
een static methode	✗	✓

De methoden in `Program` zijn static omdat `Main` dat is. En **static** is geen snelheidstruc: het gaat erom of iets bij de klasse of bij één object hoort.

// LET OP

`gewicht--`; in een **static** methode, met `gewicht` een instantievariabele: *An object reference is required for the non-static field, method, or property.* Van welk object?