Object georiënteerd programmeren

Zie Scherp Scherper - Hoofdstuk 9

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- uitleggen waarom **OOP** code modulair en onderhoudsvriendelijk maakt
- het verschil tussen een **klasse** en een **object** beschrijven
- **abstractie** en **encapsulatie** situeren (het black-box-principe)
- een klasse maken met **methoden** en **instantievariabelen**
- **access modifiers** (`public`/`private`) correct gebruiken
- objecten hun staat laten beschermen via **properties** (full en auto)
- werken met de ingebouwde **`DateTime`**-klasse

Van gestructureerd naar OOP

Tot nu toe programmeerden we **gestructureerd** (procedureel): loops, methoden, beslissingen.

- Werkt prima, maar grote projecten verzanden in **spaghetti**
- **OOP** bouwt hierop voort, maar maakt code **modulair, leesbaar en onderhoudsvriendelijk**

> Alles draait voortaan rond **klassen en objecten**.

Pong zonder OOP

```
int balX = 20; int balY = 20;
int vX = 2;    int vY = 1;
while (true) { /* posities updaten, tekenen ... */ }
```

> ⚠ **Waarschuwing**
>
> Twee balletjes? Dan moet je **bijna elke lijn verdubbelen** (`bal2X`, `v2X`, …). Bij honderd balletjes wordt het een ramp.

Pong met OOP: de klasse

```
internal class Balletje
{
    public int X { get; set; }
    public int Y { get; set; }
    public int VX { get; set; }
    public int VY { get; set; }

    public void Update() { /* botst tegen de randen */ }
    public void TekenOpScherm()
    {
        Console.SetCursorPosition(X, Y);
        Console.Write("O");
    }
}
```

We stoppen de **data** en het **gedrag** van een balletje samen in één klasse.

Pong met OOP: de Main

```
Balletje bal1 = new Balletje();
bal1.X = 20; bal1.Y = 20; bal1.VX = 2; bal1.VY = 1;

Balletje bal2 = new Balletje();
bal2.X = 10; bal2.Y = 8; bal2.VX = 2; bal2.VY = -1;

while (true)
{
    bal1.Update();  bal2.Update();
    bal1.TekenOpScherm();  bal2.TekenOpScherm();
    Thread.Sleep(50); Console.Clear();
}
```

Tip

Honderd balletjes? Een array van `Balletje` en twee loopjes. Zo simpel.

De kracht van OOP

> De **logica zit in de objecten zelf**. Elk object is verantwoordelijk voor zijn eigen gedrag, op basis van zijn eigen interne toestand.

In de `Main` zeg je gewoon: “update jezelf” en “teken jezelf”.

Even doorbijten

Duizend mammoeten en sabeltandtijgers! Ik doe een voorspelling: van alle hoofdstukken wordt **dit** het hoofdstuk waar je het meest je tanden op stukbijt.

Hou vol, geef niet te snel op, en kom hier geregeld terug.

Klasse versus object

- Een **klasse** is een **blauwdruk** die het gedrag en de toestand van objecten beschrijft
- Een **object** is een **instantie** van die klasse, met een eigen toestand en identiteit

> **Tip**
>
> Het **grondplan** van een huis is de klasse. De tien huizen die je ermee bouwt zijn de objecten: elk met eigen bakstenen en rolluiken.

Toestand, gedrag, identiteit

Een object heeft:

- **Gedrag**: beschreven door de **methoden** van de klasse
- **Toestand**: bepaald door zijn **instantievariabelen** en **properties**
- **Identiteit**: een unieke naam zodat andere objecten ermee kunnen werken

> **Tip**
>
> Zoek in een opgave naar de **zelfstandige naamwoorden**: dat zijn vaak je klassen.

Abstractie en encapsulatie

Het **black-box**-principe, denk aan een auto:

- **Encapsulatie**: alle onderdelen die samenhoren bundel je tot één geheel
- **Abstractie**: naar buiten toe vereenvoudig je de complexiteit tot een **interface** (stuur, pedalen, dashboard)

> Hoe minder de buitenwereld moet weten om met een object te werken, hoe beter.

Tijd voor taart: A PIE

De vier pijlers van OOP:

- **A**bstractie
- **P**olymorfisme (hoofdstuk 16)
- **I**nheritance / overerving (hoofdstuk 13)
- **E**ncapsulatie

> **Opmerking**
>
> Steve Jobs over objecten: *"They encapsulate complexity, and the interfaces to that complexity are high level."*

Een klasse maken

```
1  internal class Auto
2  {
3
4  }
```

- Een klassenaam begint met een **hoofdletter**
- Zet elke klasse in een **apart `.cs`-bestand** (Project - Add class…)
- VS zet er de access modifier **`internal`** voor

Objecten aanmaken met new

```
1 Auto mijnEersteAuto = new Auto();
2 Auto mijnAndereAuto = new Auto();
```

- Objecten maak je met het **`new`** keyword (zoals eerder bij `Random`)
- `new` maakt het object in het geheugen en geeft het **adres** terug

> **Tip**
>
> Klassen zijn dus gewoon een **nieuw, krachtiger datatype**: variabelen die meerdere waarden én methoden bundelen.

new en null

Zonder `new` bestaat je object niet: de variabele is dan **`null`** (een lege doos).

```
1  Auto auto;             // null
2  Console.WriteLine(auto); // use of unassigned local variable
```

> ⚠ **Waarschuwing**
>
> Een `null`-object tóch gebruiken geeft een **`NullReferenceException`**: dé klassieker bij beginnende OOP'ers. Je vergat `new`.

Object-methoden

```cs
internal class Mens
{
    public void Praat()
    {
        Console.WriteLine("Ik ben een mens!");
    }
}
```

```cs
Mens joske = new Mens();
joske.Praat();   // via de dot-operator
```

- **Geen `static`** voor methoden in zo'n klasse
- **`public`** maakt de methode bruikbaar voor de buitenwereld

Access modifiers

- **`private`**: enkel zichtbaar binnen de klasse zelf (de **standaard** als je niets schrijft)
- **`public`**: iedereen kan eraan
- **`internal`**: enkel binnen het huidige project; **`protected`**: voor overerving (later)

Belangrijk

Zet **altijd expliciet** een access modifier. Niets schrijven betekent `private`, maar dat expliciet maken leest veel duidelijker.

Waarom private?

Interne hulpcode scherm je af zodat anderen ze niet per ongeluk aanroepen.
Binnen de klasse mag je ze wél gebruiken:

```
1  public void Praat()
2  {
3      Console.WriteLine("Ik ben een mens!");
4      VertelGeheim();         // mag: zelfde klasse
5  }
6  private void VertelGeheim()
7  {
8      Console.WriteLine("Ik ben verliefd op Anneke");
9  }
```

Instantievariabelen

Elke instantie houdt zijn **eigen** toestand bij in een **instantievariabele**:

```
internal class Mens
{
    private int geboorteJaar = 1970;

    public void Praat()
    {
        Console.WriteLine($"Ik ben geboren in {geboorteJaar}.");
    }
}
```

Verjong je het ene object, dan blijft het andere ongemoeid: elk object heeft zijn **eigen** `geboorteJaar`.

Houd instantievariabelen privé

Ik zag je denken: “zal ik die instantievariabele eens `public` maken?” **Niet doen. Simpel.**

Een `public` instantievariabele laat de buitenwereld om het even welke waarde toekennen (`adil.geboortejaar = -12000;`) en breekt zo je klasse.

Belangrijk

Toegang regel je via **properties** en **methoden**, nooit via een publieke instantievariabele.

Gecontroleerde toegang

Een methode kan illegale waarden tegenhouden:

```
1 public void VeranderGeboortejaar(int nieuw)
2 {
3     if (nieuw >= 1900)
4         geboorteJaar = nieuw;
5 }
```

Een object **beschermt zichzelf**: de buitenwereld kan zijn werking niet zomaar om zeep helpen.

Properties: het idee

Darth Vader is een `SithLord` met een geheime naam en een hoeveelheid energie (nooit onder 0):

```
internal class SithLord
{
    private int energie;
    private string sithName;
}
```

> **Tip**
>
> **Properties** zijn de moderne C#-manier om de interne staat **gecontroleerd** in en uit te lezen, in plaats van losse methoden.

Full property

```
1  private int energie;
2
3  public int Energie
4  {
5      get { return energie; }
6      set { energie = value; }
7  }
```

```
1  Vader.Energie = 20;                  // set
2  Console.WriteLine(Vader.Energie);    // get
```

- Een property is altijd **`public`** en begint met een **hoofdletter**
- In de `set` zit de waarde in het keyword **`value`**

Full property met controle

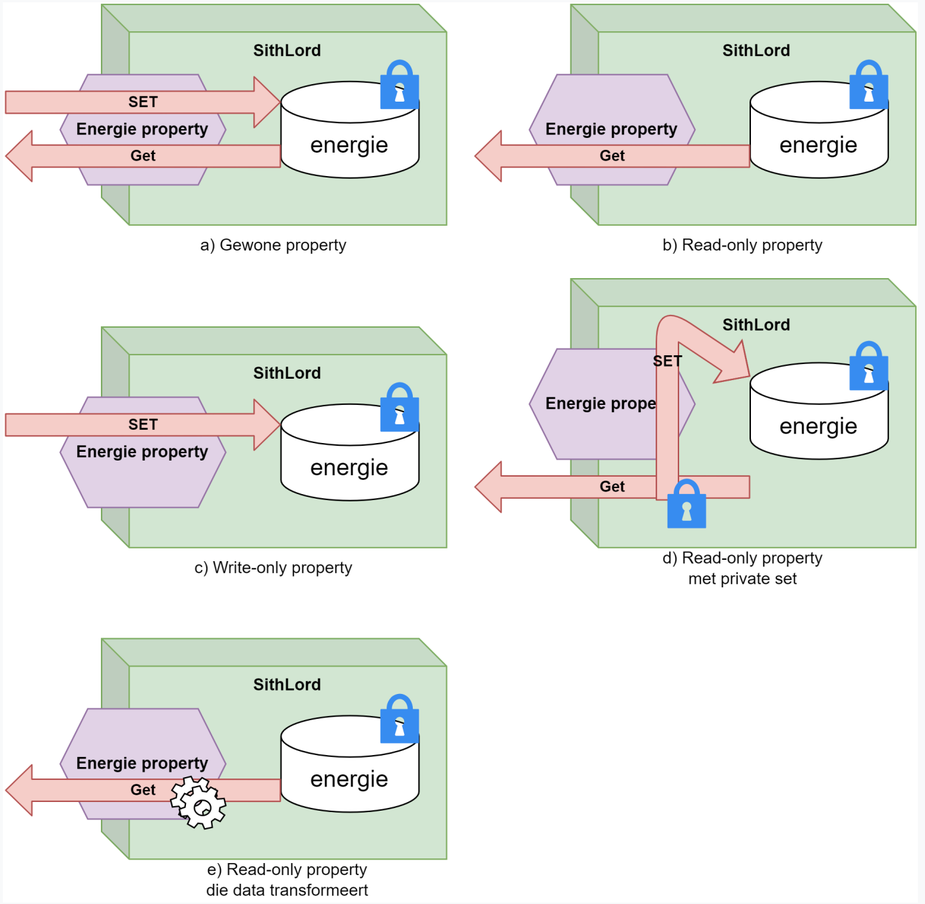
De property treedt op als **wachter**:

```
1 public int Energie
2 {
3     get { return energie; }
4     set
5     {
6         if (value >= 0)
7             energie = value;
8     }
9 }
```

`palpatine.Energie = -1;` heeft nu **geen effect**: negatieve energie wordt geweigerd.

Property-variaties

- **Write-only**: geen `get` (bv. een pincode)
- **Read-only**: geen `set`
- **Read-only met `private set`**: enkel intern aanpasbaar
- **Transformerend**: berekent data in de `get`



De full-property-varianten.

> **Tip**

Transformerende property

Een read-only property die data **berekent** (geen eigen instantievariabele):

```
1  internal class Persoon
2  {
3      public string Voornaam { get; set; }
4      public string Achternaam { get; set; }
5
6      public string VolledigeNaam
7      {
8          get { return $"{Voornaam} {Achternaam}"; }
9      }
10 }
```

Methode of property?

Tip

- Gaat het om een **actie of gedrag** (iets doen, tonen, berekenen dat tijd kost)? Gebruik een **methode**.
- Gaat het om een **eigenschap** met data die snel verkregen wordt? Gebruik een **property**.

Auto-properties

Geen extra controle nodig? Dan volstaat een **auto-property**, zonder zichtbare instantievariabele:

```
internal class Persoon
{
    public string Voornaam { get; set; }
    public int Geboortejaar { get; set; } = 2002;   // beginwaarde
    public string Naam { get; private set; }          // read-only naar buiten
}
```

Tip

Begin vaak met auto-properties; vervang ze door full properties zodra je controle of transformatie nodig hebt. (`prop` + tab + tab in VS.)

DateTime: een ingebouwde klasse

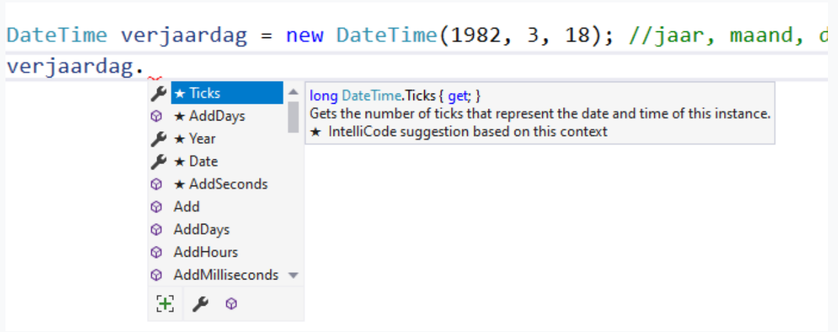
```
1  DateTime nu = DateTime.Now;                  // huidige datum en tijd
2  DateTime dag = new DateTime(2017, 4, 21);    // jaar, maand, dag
```

- `DateTime.Now` is een **static property**
- `new DateTime(...)` gebruikt een **constructor** (hoofdstuk 11) om het object meteen in te stellen

DateTime: methoden en properties

```
1  DateTime einde = trouwMoment.AddDays(35);
2  Console.WriteLine(einde.Month);      // 5
3  Console.WriteLine(einde.DayOfWeek);  // Friday
```

- `AddDays`, `AddHours`, `AddYears`, … geven telkens een **nieuw** `DateTime`-object terug
- Properties zoals `Month` en `DayOfWeek` zijn **read-only**



DateTime via de dot-operator.

DateTime: TimeSpan

Twee **`DateTime`**-objecten aftrekken geeft een **`TimeSpan`**:

```
1  DateTime vandaag = DateTime.Today;
2  DateTime geboorte = new DateTime(2009, 6, 17);
3  TimeSpan verschil = vandaag - geboorte;
4  Console.WriteLine($"{verschil.TotalDays} dagen oud.");
```

Optellen van twee datums kan niet, aftrekken wel.

Terugblik op een stevig hoofdstuk

We doorliepen vier fasen:

1. **Pong** toonde waarom OOP helpt
2. de **theorie** (klasse vs object) verwarde even
3. de **praktijk**: methoden en properties als kern van een klasse
4. `DateTime` als voorproefje van een krachtige klasse

Conclusie

- **OOP** bundelt data en gedrag in **klassen**; objecten zijn de concrete **instanties** (met `new`)
- Een object beheert zijn eigen **toestand**; **instantievariabelen** blijven **privé**
- **Properties** geven gecontroleerde toegang: full (met `get`/`set`/`value`) of auto
- Vergeet je `new`, dan is je object `null` en krijg je een **`NullReferenceException`**
- **`DateTime`** toont hoe krachtig een goed ontworpen klasse is

> **Tip**
>
> Volgende halte: **geheugenmanagement** en **uitzonderingen** (exceptions).

Meer weten

Kennisclips

- OOP, klassen en objecten: what’s it all about?
- Klassen en objecten in C#
- Methoden en access modifiers
- Properties: nut en full properties
- Auto-properties
- De DateTime klasse

Oefeningen H9 · 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: moet alles in een klasse?

Dat *alle* code in een klasse moet, deelt C# met **Java**. **Python** en **JavaScript** zijn losser: daar mag je gewoon losse code en functies schrijven.

```
# een volledig geldig programma, geen klasse in zicht
print("Hallo wereld")
```

OOP is daar een keuze, geen verplichting. In **C** bestaat het begrip klasse zelfs niet. C# zit aan het strenge eind.

Zie verder: properties elders

Java kent geen properties: daar schrijf je met de hand een aparte *getter* en *setter*:

```
1  sith.setEnergie(20);
2  System.out.println(sith.getEnergie());
```

Python zit dichter bij C# met de `@property`-decorator: methoden die je van buiten gebruikt alsof het een gewone variabele is. In beide gevallen verberg je de instantievariabele en regel je de toegang.

Samenvattende poster

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