Overerving

Zie Scherp Scherper - Hoofdstuk 13

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- uitleggen wat **overerving** is en wanneer de **is-een**-relatie geldt
- een child-klasse maken met de `:`-notatie
- het `protected`-keyword en de regels rond `private` toepassen
- begrijpen hoe **constructors** bij overerving samenwerken (`base()`)
- methoden aanpassen met `virtual` en `override`
- de parent-implementatie hergebruiken met `base`

Inheritance: de I van A PIE

We deden al **a**bstractie en **e**ncapsulatie. Nu: **i**nheritance.

- Programmeurs zijn lui: dubbele code = dubbel zoveel bugs
- Hebben `Monster` en `Held` veel gemeen? Dan delen ze wellicht een **voorouder** (`Karakter`)

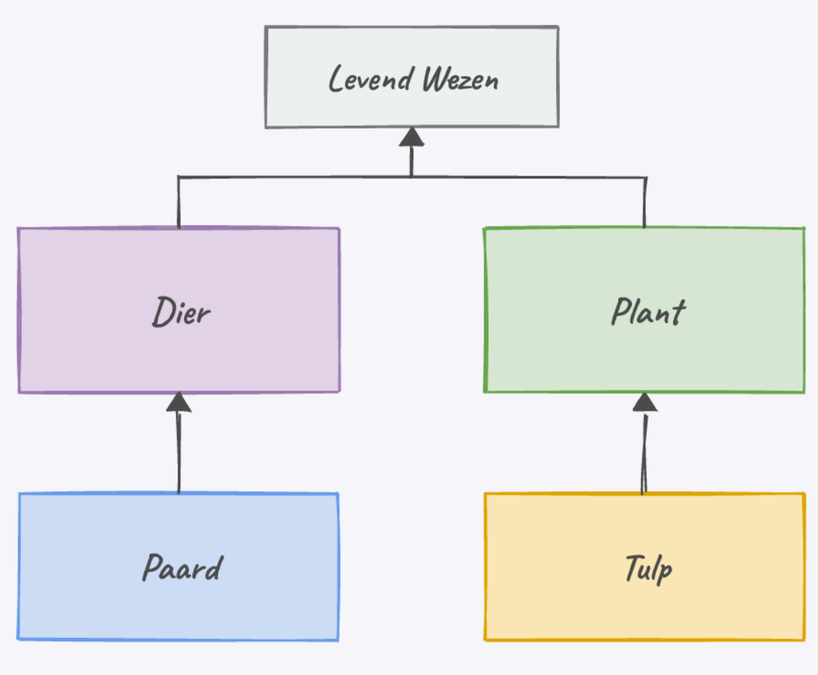
> De gemeenschappelijke code verhuist naar de **parent**; de child houdt enkel zijn **specialisatie**.

De is-een relatie

“x is een y” wijst op overerving:

- een paard **is een** dier
- een tulp **is een** plant

Kom je in een opgave *is*, *was*, *zijn* tegen, dan is overerving wellicht mogelijk.



Een is-een-hiërarchie.

Belangrijk

“x heeft een y” is géén overerving, maar associatie (volgend hoofdstuk).

Overerving in C

```cs
internal class Dier
{
    public void Eet() { /* ... */ }
}

internal class Paard : Dier
{
    public bool KanHinnikken { get; set; }
}
```

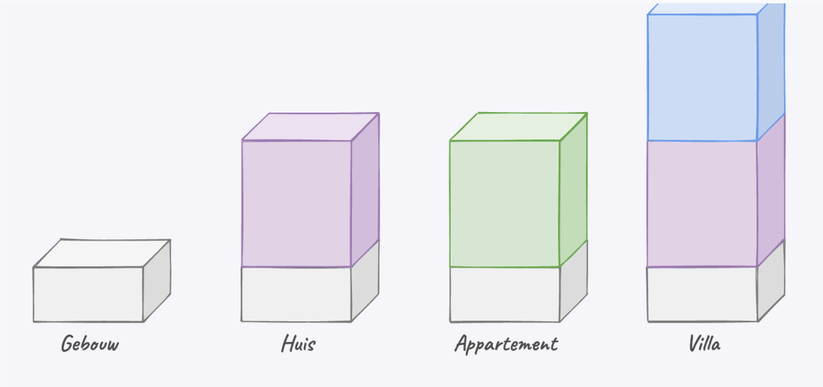
Een `Paard` kan **alles** wat een `Dier` kan, plus zijn eigen specialisatie:

```cs
Paard bpaard = new Paard();
bpaard.Eet();              // van Dier
bpaard.KanHinnikken = true; // van Paard
```

Alles wordt overgeërfd

De child erft **alles**: methoden, properties, instantievariabelen.

Een child-object is dus opgebouwd uit een **parent-deel** plus zijn eigen **specialisatie**.



Object = parent + specialisatie.

> **Belangrijk**
>
> `private` blijft `private`: de child erft het wel, maar mag er niet rechtstreeks aan.

protected

Wil je dat children er wél bij kunnen, maar de buitenwereld niet? Gebruik **`protected`**:

```
internal class Dier
{
    protected int geboortejaar;       // zichtbaar in child
}
internal class Paard : Dier
{
    public void MaakOuder() { geboortejaar++; } // werkt nu
}
```

> **Tip**
>
> Nog netter: een property met **`protected set`**, zodat de instantievariabele beschermd blijft.

Geen multiple inheritance

In C# erft een klasse van **maximaal één** parent.

- Dat vermijdt het **diamond problem**: erf je van twee parents met dezelfde methode, welke geldt dan?
- Meerdere “rollen” combineren doe je later met **interfaces** (hoofdstuk 16)

> **Opmerking**
>
> Wil je net verhinderen dat iemand van je klasse erft? Zet er `sealed` voor.

Constructors bij overerving

Maak je een child-object, dan lopen de constructors in volgorde:

1. eerst de **verste voorouder** (bovenaan de keten)
2. dan elke tussenliggende parent
3. ten slotte de **klasse zelf**

> **Tip**
>
> Logisch: de child heeft de **fundering** van zijn parent nodig voor hij zelf iets kan doen.

base()

Achter elke child-constructor staat impliciet **`: base()`** (de default constructor van de parent). Heeft de parent enkel een overloaded constructor, dan moet je die expliciet aanroepen:

```
1  internal class Soldaat
2  {
3      public Soldaat(bool kanSchieten) { /* ... */ }
4  }
5  internal class VeldArts : Soldaat
6  {
7      public VeldArts() : base(true) { }
8  }
```

Parameters doorgeven via base

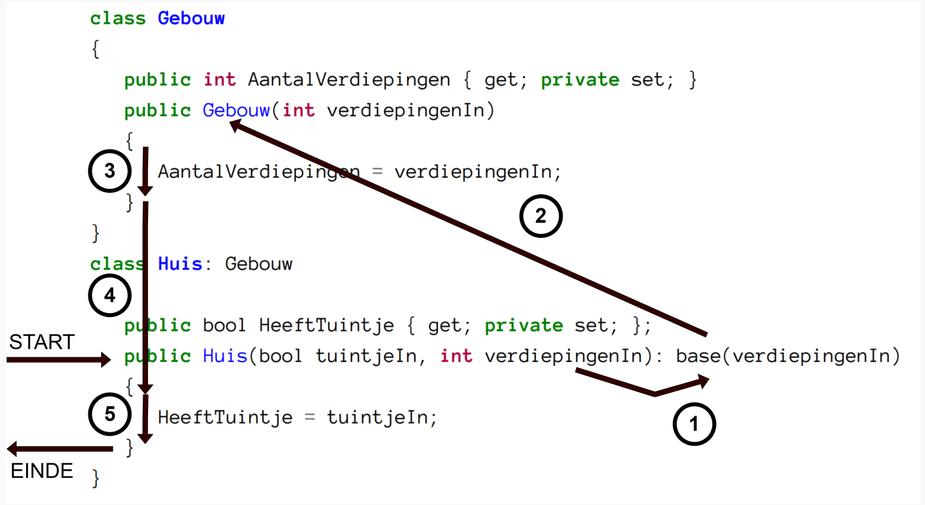
```
internal class Gebouw
{
    public int AantalVerdiepingen { get; private set; }
    public Gebouw(int verdiepingenIn) { AantalVerdiepingen = verdiepingenIn; }
}
internal class Huis : Gebouw
{
    public bool HeeftTuintje { get; private set; }
    public Huis(bool heeftTuin, int aantalVer) : base(aantalVer)
    {
        HeeftTuintje = heeftTuin;
    }
}
```

`new Huis(true, 1)`: de child houdt `heeftTuin` zelf, en sluist `aantalVer` door naar de parent.

De volgorde in beeld

`new Huis(true, 5)`:

1. constructor `Huis` start
2. roept eerst `base(5)` aan
3. `Gebouw` zet `AantalVerdiepingen`
4. terug in `Huis`: `HeeftTuintje = true`



Constructor-flow bij overerving.

virtual en override

Standaard mag een child de werking van de parent **niet** zomaar wijzigen. De parent moet dat **toelaten** met **`virtual`**; de child gebruikt **`override`**:

```
internal class Vliegtuig
{
    public virtual void Vlieg()
    {
        Console.WriteLine("Het vliegtuig vliegt door de wolken.");
    }
}
internal class Raket : Vliegtuig
{
    public override void Vlieg()
    {
        Console.WriteLine("De raket verdwijnt in de ruimte.");
    }
}
```

virtual en override: regels

- `virtual` mag op alles behalve `private` (en niet op `static`)
- `virtual` is een **mogelijkheid**, geen verplichting om te overriden
- De signatuur van de `override` moet **identiek** zijn aan die van de parent

> ⚠ **Waarschuwing**
>
> Vergeet je `override` (en schrijf je gewoon `public void Vlieg()`), dan krijg je een **`new`**-waarschuwing: C# denkt dat je de methode wil **verbergen** (*hiding*), niet **overschrijven**.

base in een methode

Wil je de parent-versie **hergebruiken** en uitbreiden? Roep ze aan met **`base`**:

```
internal class Frituur : Restaurant
{
    public override void PoetsAlles()
    {
        base.PoetsAlles();  // basiskost van Restaurant
        kosten += 500;      // extra voor de frituur
    }
}
```

Zo blijft de basislogica op **één** plek staan: verandert de basisprijs, dan klopt alles nog.

Nog een base-voorbeeld

```
internal class ModerneMens : Oermens
{
    public bool IsJager { get; set; }

    public override int VoorzieVoedsel()
    {
        if (IsJager)
            return base.VoorzieVoedsel() + 20;
        return 100;
    }
}
```

Een jager bouwt voort op de techniek van de `Oermens` (`base`) en doet er 20 kg bovenop.

Pong met overerving

Maak `Update` in `Balletje` **`virtual`** en bouw een nieuw soort balletje:

```
internal class CentreerBalletje : Balletje
{
    public override void Update()
    {
        if (X + VX >= Console.WindowWidth || X + VX < 0)
        {
            X = Console.WindowWidth / 2;
            Y = Console.WindowHeight / 2;
        }
        base.Update();
    }
}
```

`Balletje` blijft onaangeroerd; de nieuwe functionaliteit zit in de child.

Conclusie

- **Overerving** geldt bij een echte **is-een**-relatie; de child specialiseert de parent
- De child erft **alles**, maar `private` blijft afgeschermd; `protected` opent het voor children
- Constructors lopen van **parent naar child**; met `base()` kies je de juiste parent-constructor
- `virtual` + `override` laten een child de werking aanpassen
- Met `base` hergebruik je de parent-implementatie binnen een override

> **Tip**
>
> Volgende halte: **gevorderde overerving**: `System.Object`, `abstract` en eigen exceptions.

Meer weten

Kennisclips

- Overerving overzicht
- Constructors bij overerving
- virtual en override
- base keyword

Oefeningen H13 · 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: overervings-syntax

Hetzelfde idee, andere notatie. In **Python** zet je de parent tussen haakjes:

```
1 class Paard(Dier):
2     pass
```

Java gebruikt `extends` (`class Paard extends Dier`), **C++** schrijft `class Paard : public Dier`.

Zie verder: virtual en override

C# vereist expliciet `virtual` op de parent én `override` op de child. In **Python** is élke methode automatisch overschrijfbaar, zonder keywords:

```
class Raket(Vliegtuig):
    def vlieg(self):
        print("De raket verdwijnt in de ruimte.")
```

Ook **Java** maakt elke (niet-`final`) methode standaard overschrijfbaar. C# kiest bewust het omgekeerde.

Zie verder: base versus super

Wat C# `base` noemt, heet in **Python** en **Java** net `super`:

```python
class Frituur(Restaurant):
    def poets_alles(self):
        super().poets_alles()   # in C#: base.PoetsAlles()
        self.kosten += 500
```

Samenvattende poster

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