



// 01

De is-een-relatie

Een **child-klasse** erft van een **parent-klasse** en is er een specialisatie van: ze kan alles wat de parent kan, plus haar eigen code. De gemeenschappelijke code staat maar één keer, in de parent.

- Een paard **is een** dier, een tulp **is een** plant
- Staat er *is*, *was* of *zijn* in een opgave, dan past overerving wellicht

// LET OP

Gelijkaardige code alleen is geen reden om te overerven: er moet een realistische “is een”-relatie zijn. “X **heeft een** Y” is geen overerving maar associatie.

// 02

Overerving in C#

Een dubbele punt achter de klassenaam: `Paard` erft van `Dier`.

```
internal class Dier
{
    public void Eet() { /* ... */ }
}
internal class Paard : Dier
{
    public bool KanHinnikken { get; set; }
}
```

Een `Paard` kan `Eet()` en heeft `KanHinnikken`. Een `Dier` kan enkel `Eet()`: `aDier.KanHinnikken` geeft een fout.

// 03

private en protected

De child erft **alles**: methoden, properties en instantievariabelen. Ook wat `private` is, maar daar mag haar code niet aan.

MODIFIER	BEREIKBAAR IN
<code>private</code>	enkel de klasse zelf
<code>protected</code>	de klasse én haar child-klassen
<code>public</code>	overall

```
internal class Dier
{
    protected int geboortejaar; // private: fout
}
internal class Paard : Dier
{
    public void MaakOuder() { geboortejaar++; }
}
```

Veiliger: een property met `protected set`, zoals `{ get; protected set; }.`

// 04

Eén parent, en sealed

Een klasse erft van **hoogstens één** parent: geen *multiple inheritance*. Zo vermijdt C# het *diamond problem* (twee parents met dezelfde methode: welke geldt dan?). Van een `sealed` klasse mag niemand erven:

```
internal sealed class DoNotInheritMe { }
internal class Child : DoNotInheritMe { } // fout
```

// 05

Constructors en base(...)

Bij een nieuw child-object draait eerst de constructor van de **verste voorouder**, dan die van elke parent daaronder, en als laatste die van de klasse zelf.

Achter elke child-constructor staat onzichtbaar `base()`, de default constructor van de parent. Met `base(...)` kies je zelf een constructor van de parent en geef je parameters door:

```
internal class Gebouw
{
    public Gebouw(int verdiepingenIn)
    { /* ... */ }
}
internal class Huis : Gebouw
{
    public Huis(bool heeftTuin, int aantalVer)
        : base(aantalVer)
    { /* ... */ }
}
```

`new Huis(true, 5)`: eerst krijgt `Gebouw` de 5 en loopt zijn constructor, daarna die van `Huis`.

// LET OP

Heeft de parent enkel een constructor met parameters, dan is er geen `base()` om impliciet aan te roepen. Zonder eigen `base(...)`, zoals `public VeldArts() : base(true)`, compileert de child niet.

// 06

virtual en override

De parent zet `virtual` op wat een child mag aanpassen, de child schrijft `override`. De rest van de signatuur blijft identiek.

```
internal class Vliegtuig
{
    public virtual void Vlieg() { /* ... */ }
}
internal class Raket : Vliegtuig
{
    public override void Vlieg() { /* ... */ }
}
```

- `virtual` is een mogelijkheid, geen verplichting
- Niet op `private` of `static` methoden
- `override` zonder `virtual` in de parent: compileerfout

// LET OP

Vergeet je `override`, dan compileert het wel, met een waarschuwing over `new`: dat is *hiding*. Via een variabele van het parent-type (`Vliegtuig v = new Raket();`) draait dan de `Vlieg` van de parent.

// 07

base.Methode()

In een override roep je met `base` de versie van de parent alsnog op en bouw je erop verder, waar in de methode je maar wil.

```
internal class Frituur : Restaurant
{
    public override void PoetsAlles()
    {
        base.PoetsAlles(); // basiskost: 1000
        kosten += 500;     // ontsmetting
    }
}
```