



// 01

Alles erft van System.Object

Elke klasse stamt af van de oer-klasse `System.Object`: je eigen klassen, `Random`, `Console` en zelfs `int` en `bool`. Schrijf je geen parent, dan erft je klasse **impliciet** van `System.Object`:

```
internal class Student : System.Object { }
```

// 02

Vier ingebouwde methoden

Elk object heeft ze meteen, geërfd van `System.Object`:

METHODE	DOET
<code>ToString()</code>	<code>virtual</code> een <code>string</code> die het object voorstelt
<code>Equals(o)</code>	<code>virtual</code> zijn twee objecten gelijk?
<code>GetHashCode()</code>	<code>virtual</code> een hash van het object
<code>GetType()</code>	niet het type (de klasse) van het object

`GetType()` kan je niet overriden. `stud1.GetType()` geeft `StudentManager.Student`, met `.Name` erachter enkel `Student`.

// 03

ToString() overriden

`Console.WriteLine(stud1)` roept stiekem `stud1.ToString()` aan. Standaard krijg je de klassenaam. In `Student`, met de autoprops `Voornaam` en `Geboortejaar`:

```
public override string ToString()
{
    return $"{Voornaam} ({Geboortejaar})";
}
```

UITVOER
Tim (1981)

// 04

Equals() en GetHashCode()

Jij beslist wanneer twee objecten gelijk zijn.

```
public override bool Equals(object o)
{
    // null of geen Student? Niet gelijk.
    if (o is not Student temp)
        return false;
    return Voornaam == temp.Voornaam
        && Geboortejaar == temp.Geboortejaar;
}
```

// LET OP

Een harde cast `(Student)o` zonder controle crasht op `null` of een ander type. En `return this == temp;` vergelijkt enkel de adressen: twee aparte objecten zijn dan nooit gelijk.

Gelijke objecten moeten dezelfde hashcode geven. Geef dezelfde instantievariabelen mee:

```
public override int GetHashCode()
{
    return GetHashCode.Combine(Voornaam,
                               Geboortejaar);
}
```

// LET OP

Override je `Equals`, override dan **ook** `GetHashCode`. Anders loopt het mis als je het object als sleutel in een `Dictionary` of `HashSet` gebruikt.

// 05

Abstracte klassen en methoden

Van een `abstract` klasse kan je **geen object** maken (`new Dier()` geeft een compileerfout), wel overerven. Een abstracte methode heeft geen body: elke child-klasse **moet** ze overriden.

```
internal abstract class Dier
{
    public string Naam { get; set; }
    public abstract string MaakGeluid();
}
internal class Paard : Dier
{
    public override string MaakGeluid()
    {
        return "Hinnikhinnik";
    }
}
```

- `virtual`: override **mag**. `abstract`: override **moet**.
- Een abstracte methode of property kan enkel in een abstracte klasse.
- `abstract` en `sealed` zijn tegenpolen: van `abstract` **moet** je overerven, van `sealed` **mag** het niet.

// 06

Abstracte properties

Je kiest meteen `{ get; }`, `{ set; }` of `{ get; set; }`:

```
public abstract int MaxLeeftijd { get; }
```

// LET OP

Dat is géén auto-property: er zit geen verborgen instantievariabele achter. De child-klasse schrijft zelf de `get`, bv. `get { return 100; }`.

// 07

Zelf een exception opwerpen

Een exception is een gewone klasse die erft van `Exception`. Met `throw` werp je er zelf een op:

```
throw new DivideByZeroException("BOEM!");
```

Opvangen doe je zoals altijd met `try` en `catch`.

// 08

Een eigen exception

Erf van `Exception` en voorzie de drie standaard-constructors:

```
internal class Timception : Exception
{
    public Timception() { }
    public Timception(string message)
        : base(message) { }
    public Timception(string message,
                      Exception inner)
        : base(message, inner) { }
}
```

Je boodschap lees je met `e.Message`, de verpakte fout met `e.InnerException`.

// LET OP

Werp je een opgevangen exception opnieuw op, schrijf dan `throw; .` Met `throw e;` wis je de stack trace en vind je de echte oorzaak moeilijk terug.