



// 01

Heeft een: associatie

Gebruik je een object **in** een ander object, dan is dat een **associatie**. Die herken je aan een “**heeft een**”-relatie: een mango heeft een pit, een PC heeft een harde schijf. Overerving is een “**is een**”-relatie.

// LET OP

Een boek **is** geen pagina. Daar overerving gebruiken is een klassieke fout: een boek **heeft** pagina's.

// 02

Compositie of aggregatie

Het verschil zit in de **levensduur** van het innerlijke object.

- **Compositie**: het kan niet zonder het omliggende object. Verdwijnt het huis, dan verdwijnt de kamer.
- **Aggregatie**: beide bestaan los van elkaar. Een motor kan je uit een gesloopte auto redden.

In UML staat de ruit bij het omliggende object: **vol** bij compositie, **leeg** bij aggregatie. Een getal geeft de verhouding.



// 03

Het innerlijke object aanmaken

Een PC heeft een harde schijf, dus krijgt **PC** een instantievariabele van het type **HardeSchijf**. Maak dat object ook aan, anders blijft het **null**.

```
// in de klasse PC
private HardeSchijf schijf = new HardeSchijf();
```

Dat kan ook in de constructor, bijvoorbeeld enkel als de klant een voorgeïnstalleerde schijf wil. De PC maakt zijn eigen schijf en die verdwijnt mee: **compositie**. Bij **aggregatie** maak je de schijf **buiten** de PC en steek je ze erin via een property

```
public HardeSchijf HardeSchijf {get; set;};
```

```
HardeSchijf mijnSchijf = new HardeSchijf();
PC mijnPC = new PC();
mijnPC.HardeSchijf = mijnSchijf;
```

De referentie **mijnSchijf** blijft, dus de garbage collector ruimt de schijf niet op samen met **mijnPC**.

// 04

Referenties en null

Het innerlijke object staat **apart** in de heap. Het omliggende object bewaart enkel een referentie ernaar.



In de PC zelf gebruik je de schijf via de instantievariabele. Controleer eerst op **null**:

```
if (schijf != null)
    Console.WriteLine(schijf.MaxCapacity);
```

// LET OP

Spreek je een innerlijk object aan dat nooit aangemaakt werd, dan krijg je een **NullReferenceException**.

// 05

Heeft meerdere

Een boek heeft pagina's: een array of een **List** van het innerlijke type. Geef de lijst geen publieke **set** en voeg een eigen methode toe, zodat **Boek** een **blackbox** blijft.

```
public List<Pagina> AllePaginas
{
    get; private set;
} = new List<Pagina>();

public void InsertPagina(Pagina p, int positie)
{
    AllePaginas.Insert(positie, p);
}
```

Bij een array (**Pagina[]**) maak je elk element nog apart aan.

// LET OP

private set verbiedt enkel **zieScherper.AllePaginas = ...**. Maar **zieScherper.AllePaginas.Clear()**; werkt nog: de **get** geeft de echte lijst terug, geen kopie.

// 06

Associatie of overerving?

Je gebruikt veel vaker “heeft een” dan “is een”: **favor composition over inheritance**. Twijfel je of een “is een” klopt, kies dan “heeft een”. Te veel compositie kan een klasse wel minder herbruikbaar maken.

Een **Mens** heeft een **Hart** als property. Een transplantatie:

```
patient.Hart = null; // vorig hart weg
patient.Hart = donor.Hart;
```

// LET OP

Na **patient.Hart = donor.Hart**; verwijzen beide naar **hetzelfde Hart**-object in de heap. Er wordt niets gekopieerd.

// 07

Het this keyword

this is de referentie naar het **huidige object**. Je gebruikt het om:

- een gelijknamige instantievariabele of property te bereiken
- het object **zichzelf** als parameter te laten meegeven
- een andere constructor op te roepen:

```
public Werknemer() : this(1)
```

// LET OP

this werkt enkel in een instance-context. In een **static** methode bestaat er geen huidig object.

// 08

this in de praktijk

Bij een naamconflict wint de parameter:

```
public Monster(int Levens)
{
    Levens = 5; // de parameter
    this.Levens = 5; // de property
}
```

Een werknemer geeft zichzelf mee aan een externe methode:

```
public bool IsPromoveerbaar() // in Werknemer
{
    return Management.MagPromoveren(this);
}
```