Polymorfisme

Zie Scherp Scherper - Hoofdstuk 16

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- uitleggen wat **polymorfisme** is en hoe **late binding** werkt
- child-objecten in een variabele van het **parent-type** bewaren (**upcasting**)
- lijsten van een basistype vullen met allerlei child-objecten
- polymorfisme inzetten om code **eenvoudiger** te maken
- de `is`- en `as`-keywords en **pattern matching** gebruiken
- `Equals` netjes overriden

De P van A PIE

Polymorfisme = *poly* (meerdere) + *morfisme* (vormen): **meerdere vormen**.

> Het laat toe om child-objecten te behandelen als objecten van hun **parent**-klasse, en zorgt dat `virtual`/`override` echt werkt.

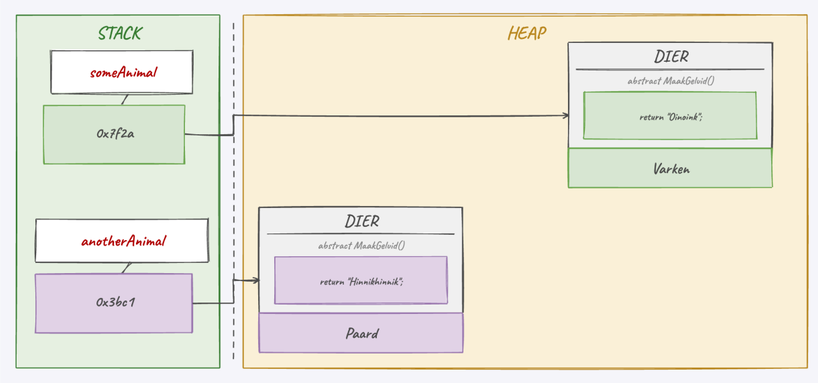
Is-een in actie

```
internal abstract class Dier
{
    public abstract string MaakGeluid();
}
internal class Paard : Dier
{
    public override string MaakGeluid() { return "Hinnikhinnik"; }
}
internal class Varken : Dier
{
    public override string MaakGeluid() { return "Oinkoink"; }
}
```

Child in een parent-variabele

```
1  Dier a = new Varken();
2  Dier b = new Paard();
3  Console.WriteLine(a.MaakGeluid()); // Oinkoink
4  Console.WriteLine(b.MaakGeluid()); // Hinnikhinni
```

De variabele is van het type `Dier`, maar elk roept zijn **eigen** `MaakGeluid` op.



Het type van de variabele bepaalt wat bereikbaar is.

Late binding

Waarom voert `a.MaakGeluid()` toch de **Varken**-versie uit?

- C# kiest pas **tijdens de uitvoer** welke `override` draait, op basis van het **echte object** in de heap
- Dat heet **late binding** (*dynamic dispatch*)
- Het werkt enkel omdat de methode **`virtual`** of **`abstract`** is

Upcasting

Een child toewijzen aan een parent-variabele (**upcasting**) gaat automatisch, tot zelfs:

```
1  System.Object mijnObject = new Varken();
```

> **Belangrijk**
>
> De compiler kijkt naar het **type van de variabele**, niet naar het echte object. Via `mijnObject` kan je dus enkel wat `System.Object` kent. Terug naar `Varken` (**downcasting**) doe je veilig met `is` en `as`.

Arrays en polymorfisme

Een lijst van het **basistype**, gevuld met child-objecten:

```
List<Dier> zoo = new List<Dier>();
zoo.Add(new Varken());
zoo.Add(new Paard());

foreach (var dier in zoo)
{
    Console.WriteLine(dier.MaakGeluid());
}
```

Objecten als één geheel behandelen, zonder dat ze exact hetzelfde type hoeven te zijn.

Met grote kracht…

Polymorfisme werkt enkel mooi als een child zich **echt** kan gedragen als zijn parent (het *Liskov substitution principle*).

Het klassieke tegenvoorbeeld: een `Vierkant` die overerft van `Rechthoek` klopt taalkundig, maar geeft brokkelige code. Kan elk child overal de plaats van zijn parent innemen?

In de praktijk: de drukke premier

```
internal class EersteMinister
{
    public MinisterVanMilieu Jansens { get; set; } = new MinisterVanMilieu();
    public MinisterBZ Ganzeweel { get; set; } = new MinisterBZ();

    public void Regeer()
    {
        Jansens.VerhoogBosSubsidies();
        Jansens.OpenOnderzoek();
        Ganzeweel.VervangAmbassadeur();
        // ... de premier moet ELK departement van binnen kennen
    }
}
```

De `EersteMinister` weet veel te veel: dat vloekt tegen **abstractie**.

De oplossing: een abstracte Minister

```
internal abstract class Minister
{
    public abstract void Adviseer();
}
internal class MinisterVanMilieu : Minister
{
    public override void Adviseer()
    {
        VerhoogBosSubsidies();
        OpenOnderzoek();
    }
}
```

Elke minister **moet** `Adviseer` overriden.

Regeren wordt eenvoudig

```
internal class EersteMinister
{
    public List<Minister> AlleMinisters { get; set; } = new List<Minister>();

    public void Regeer()
    {
        foreach (Minister minister in AlleMinisters)
        {
            minister.Adviseer();
        }
    }
}
```

De premier hoeft geen enkel departement nog van binnen te kennen. Wie zei dat regeren moeilijk was?

Het is-keyword

`is` test of een object van een bepaald type is, en geeft een **`bool`**:

```
1  List<Voertuig> alleMiddelen = new List<Voertuig>();
2  alleMiddelen.Add(new Auto());
3
4  foreach (var middel in alleMiddelen)
5  {
6      if (middel is Auto)
7      {
8          // doe enkel iets met de auto's
9      }
10 }
```

Het as-keyword

`as` **probeert** te converteren; lukt het niet, dan krijg je **`null`** in plaats van een crash:

```
1 Mens jos = fritz as Mens;
2 if (jos != null)
3 {
4     // doe Mens-zaken
5 }
```

> **Belangrijk**
>
> `as` werkt enkel op **reference types** (objecten en interfaces), niet op value types als `int`.

> **Tip**
>
> **Vuistregel:** wil je het object daarna *gebruiken*, kies `as`; wil je enkel *weten* of iets van een type is, kies `is`.

Pattern matching

Sinds C# 7 doe je de check én de omzetting in **één** stap:

```
1  if (mijnAuto is Voertuig v)
2  {
3      // v is hier meteen beschikbaar als Voertuig
4  }
```

> **Tip**
>
> In dezelfde geest: `if (jos is not null)` is de modernere variant van `!= null`. Je ziet deze vorm overal in moderne C#.

Equals netjes overriden

Met `is` schrijf je een veilige `Equals` (geen crash bij een verkeerd type):

```
1  public override bool Equals(object o)
2  {
3      if (o is Student temp)
4          return Geboortejaar == temp.Geboortejaar && Voornaam == temp.Voornaam;
5      return false;
6  }
```

Belangrijk

Wie `Equals` overridet, override **ook** `GetHashCode` (met `HashCode.Combine`), anders haperen `Dictionary` en `HashSet`.

== of .Equals()?

- `==` vergelijkt bij objecten standaard de **referenties**: twee aparte objecten met dezelfde inhoud zijn dus niet gelijk
- Door `Equals` te overriden bepaal jij gelijkheid op basis van de **inhoud**

> **Tip**
>
> Sinds C# 9 regelen `record`-types dit automatisch: `public record Student(string Voornaam, int Geboortejaar);` genereert zelf `Equals`, `GetHashCode` én `ToString`. We doen het hier met de hand om te begrijpen wat eronder zit.

Conclusie

- **Polymorfisme** behandelt child-objecten als hun parent; **late binding** kiest de juiste `override`
- **Upcasting** gaat vanzelf; voor **downcasting** gebruik je `is` en `as`
- Een `List<Basistype>` met child-objecten maakt code flexibel en kort (de premier!)
- `is` toetst een type, `as` zet veilig om (`null` bij mislukking), **pattern matching** doet beide
- Override `Equals` voor vergelijking op inhoud, en vergeet `GetHashCode` niet

> **Tip**
>
> Volgende halte: **interfaces**, contracten die klassen beloven na te komen.

Meer weten

Kennisclips

- Polymorfisme
- Polymorfisme in de praktijk met presidenten
- Is en as keywords
- Objecten vergelijken met Equals: polymorfisme in actie

Oefeningen H16 · 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: polymorfisme in Python

C# heeft een gemeenschappelijke parent plus `virtual`/`override` nodig. **Python** doet het via *duck typing*: heeft het object de methode, dan werkt het gewoon, zonder gedeelde basisklasse.

```
def laat_spreken(dier):
    print(dier.maak_geluid())   # elk object met die methode werkt

laat_spreken(Paard())   # Hinnikhinnik
laat_spreken(Varken())  # Oinkoink
```

Bij Python merk je pas tijdens het uitvoeren of een object de methode echt heeft; C# controleert dat al bij het compileren.

Zie verder: type-check in andere talen

Dezelfde vraag, andere notatie. In **Python** is het een functie:

```python
if isinstance(mijn_auto, Voertuig):
    print("mijn_auto is een Voertuig")
```

In **JavaScript** een operator:

```javascript
if (mijnAuto instanceof Voertuig) {
    console.log("mijnAuto is een Voertuig");
}
```

Zie verder: gelijkheid in andere talen

Zelf bepalen wanneer objecten gelijk zijn, doe je via een vaste methode.
Python override `__eq__` (zit achter `==`):

```python
def __eq__(self, other):
    return self.voornaam == other.voornaam
```

Java lijkt op C#: override `equals` (en `hashCode`):

```java
@Override
public boolean equals(Object o) { ... }
```

In **JavaScript** kan `===` niet overschreven worden: daar schrijf je zelf een functie `zijnGelijk(a, b)`.

Samenvattende poster

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