



// 01

Meerdere vormen

De **P** uit A PIE: *poly* (meerdere) + *morfisme* (vormen). Een child-object mag in een variabele van het **parent-type**.

```
internal abstract class Dier
{
    public abstract string MaakGeluid();
}
```

Varken en Paard erven van Dier en overrideen MaakGeluid.

```
Dier someAnimal = new Varken();
Dier anotherAnimal = new Paard();
Console.WriteLine(someAnimal.MaakGeluid());
Console.WriteLine(anotherAnimal.MaakGeluid());
```

```
UITVOER
Oinkoink
Hinnikhinnik
```

// 02

Late binding

- Het **type van de variabele** bepaalt wat je mag aanroepen: enkel wat in Dier staat.
- Het **echte object** in de heap bepaalt welke **override** draait, pas tijdens de uitvoer: **late binding** (*dynamic dispatch*).
- Dat werkt enkel bij een **virtual** of **abstract** parent-methode.

// LET OP

Zonder **virtual** of **abstract** op de parent-methode draait gewoon de parent-versie. Met **override** in de child compileert het dan zelfs niet.

// 03

Upcasting en downcasting

Een child in een parent-variabele steken heet **upcasting** en gaat automatisch. Alles erft van **System.Object**, dus ook dit mag:

```
System.Object mijnObject = new Varken();
mijnObject.MaakGeluid(); // compileerfout
```

Via **mijnObject** kan je enkel wat **System.Object** kent (**ToString**, **Equals**, ...). Terug naar **Varken** (**downcasting**) moet expliciet: veilig met **is** en **as**.

// 04

Lijsten van het basistype

Een **List<Dier>** vul je met objecten van **Dier** én van alle child-klassen. Eén lus verwerkt ze allemaal, zonder type-checks.

```
List<Dier> zoo = new List<Dier>();
zoo.Add(new Varken());
zoo.Add(new Paard());
foreach (Dier dier in zoo)
{
    Console.WriteLine(dier.MaakGeluid());
}
```

Liskov substitution principle: dat werkt enkel als elk child zich overal kan gedragen als zijn parent. Een **Vierkant** die overerft van **Rechthoek** klopt taakkundig, maar geeft brokkelige code.

// 05

De eerste minister

Elke minister erft van de abstracte klasse **Minister** en moet **public abstract void Adviseer();** overrideen. De **EersteMinister** hoeft geen enkel departement nog van binnen te kennen:

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

// LET OP

Geen **if (minister is MinisterBZ)** per type: zo bouw je met de hand na wat polymorfisme gratis doet, en vergeet je ergens een nieuwe minister.

// 06

is en as

is test of een object van een type is en geeft een **bool**. **as** zet om, en geeft **null** als dat niet lukt in plaats van een uitzondering.

```
Varken varken = dier as Varken;
if (varken != null) { /* Varken-zaken */ }
```

	VARKEN IN HEAP	PAARD IN HEAP
dier is Varken	true	false
dier as Varken	de referentie	null
(Varken)dier	de referentie	crasht

Object daarna gebruiken: **as**. Enkel het type weten: **is**. Niet op value types zoals **int**.

// LET OP

Test en gebruik dezelfde variabele. **eersteObject is Student** testen en **tweedeObject as Student** gebruiken kan alsnog **null** geven.

// 07

Pattern matching

Controle en omzetting in één stap: zet een variabelenaam achter het type. Null-controles schrijf je modern als **is null** en **is not null**.

```
if (dier is Varken v)
{
    Console.WriteLine(v.MaakGeluid());
}
```

// 08

Equals en GetHashCode

= vergelijkt bij objecten de **referenties**. Met een eigen **Equals** vergelijk je op **inhoud**:

```
public override bool Equals(object o)
{
    if (o is Student temp)
        return Voornaam == temp.Voornaam
            && Geboortejaar == temp.Geboortejaar;
    return false;
}
```

// LET OP

Override je **Equals**, override dan ook **GetHashCode**, met **return GetHashCode.Combine(Voornaam, Geboortejaar);**. Anders werkt je object niet correct als key in een **Dictionary** of **HashSet**.