



Geheugenmanagement en uitzonderingen



// 01

Stack en heap

Je programma krijgt twee geheugens: de kleine, snelle **stack** en de grote, tragere **heap**. Waar een variabele belandt, hangt af van haar type.

	VALUE TYPES	REFERENCE TYPES
voorbeelden	<code>int</code> , <code>double</code> , <code>bool</code> , <code>char</code> , <code>enums</code>	objecten, arrays, <code>string</code>
op de stack	de waarde zelf	een adres naar de heap
beginwaarde	<code>0</code> , <code>false</code> , ...	<code>null</code>
= kopieert	de waarde	het adres

Een object staat **in zijn geheel** in de heap, ook zijn instantievariabelen van een value type.

// 02

Kopiëren met =

```
Student a = new Student();
a.Naam = "Abba";
Student b = a;           // kopie van het adres
b.Naam = "Queen";
Console.WriteLine(a.Naam);
```

UITVOER
Queen

// LET OP

`b = a`; kopieert het object niet: `b` wijst naar hetzelfde object als `a`. Wees `b` al naar een ander object, dan ben je dat kwijt.

// 03

De Garbage Collector

De **GC** ruimt om de zoveel tijd de objecten in de heap op waar geen enkele referentie meer naar wijst.

```
Held supermand = new Held();
Held batmand = new Held();
batmand = supermand; // 2e held onbereikbaar
```

// LET OP

Roep `GC.Collect()` niet zelf aan. De GC kiest zelf het beste moment; zelf ingrijpen maakt je programma meestal trager.

// 04

Objecten en methoden

Een parameter van een value type is een **kopie**. Van een object krijgt de methode het **adres**: ze werkt op **hetzelfde** object, net als bij arrays.

```
public void VoegMetingToeEnVerwijder(
    Meting inMeting)
{
    Temperatuur += inMeting.Temperatuur;
    inMeting.Temperatuur = 0;
}
// m1 en m2 zijn twee metingen
m1.Temperatuur = 26;
m2.Temperatuur = 5;
m1.VoegMetingToeEnVerwijder(m2);
Console.WriteLine(m1.Temperatuur); // 31
Console.WriteLine(m2.Temperatuur); // 0
```

Een object kan ook returntype zijn: `public Mens PlantVoort()` geeft een `new Mens()` terug.

// 05

null

Een reference type dat (nog) nergens naar wijst, bevat `null`. Een methode mag bewust `null` teruggeven, bv. als ze niets vond. Test met `if (stud1 != null)`, of korter:

```
Student stud1 = null;
Console.WriteLine(stud1?.Naam); // lege lijn
```

// LET OP

`stud1.Naam` op een object dat `null` is, compileert, maar crasht met een `NullReferenceException`.

// 06

Namespaces en using

Een **namespace** voorkomt naamconflicten, zoals de gemeente bij een Kerkstraat: `MyEpicGame.Monster` en `NietZoEpicGame.Monster` zijn twee verschillende klassen. Met `using System.Diagnostics`; bovenaan je bestand zoekt C# ook in die namespace. Kent VS een klasse niet, klik dan op het **lampje**: het zet de juiste `using` erbij.

// 07

try en catch

Een **exception** is een uitzondering tijdens de uitvoering, zoals foute invoer of een deling door nul. Zonder opvang crasht je programma.

```
try
{
    string invoer = Console.ReadLine();
    int noemer = Convert.ToInt32(invoer);
    Console.WriteLine(100 / noemer);
}
catch (FormatException)
{
    Console.WriteLine("Geen geheel getal");
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Loopt er in `try` iets mis, dan springt je programma naar de eerste passende `catch`. Elke exception is een `Exception`, dus `catch (Exception e)` vangt alles op.

// LET OP

Specifiek eerst, `catch (Exception e)` als laatste. Omgekeerd compileert je code niet.

// 08

e, throw en finally

- `e.Message` is de foutmelding, `e.StackTrace` het spoor door je methoden. Toon dat niet zomaar aan je gebruiker.
- `throw new Exception("Oei");` werpt zelf een exception op. Je programma klimt terug door de aanroepen tot er een `catch` klaarstaat.
- `finally { }` na de catch-blokken loopt **altijd**, met of zonder fout.
- Een try/catch rond een hele methode-aanroep stopt bij de eerste fout. Staat hij in een loop, dan gaat de loop verder.

// LET OP

Een leeg `catch { }` slikt de fout stil in en verbergt je bugs.