Geheugen- en codebeheer

Zie Scherp Scherper - Hoofdstuk 10

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

- uitleggen hoe het geheugen verdeeld is in **stack** en **heap**
- het verschil tussen **value types** en **reference types** doorgronden
- de rol van de **Garbage Collector** beschrijven
- werken met `null` en de `NullReferenceException` vermijden
- `namespace` en `using` situeren
- robuuste code schrijven met `try`, `catch` en `finally`

Twee soorten geheugen

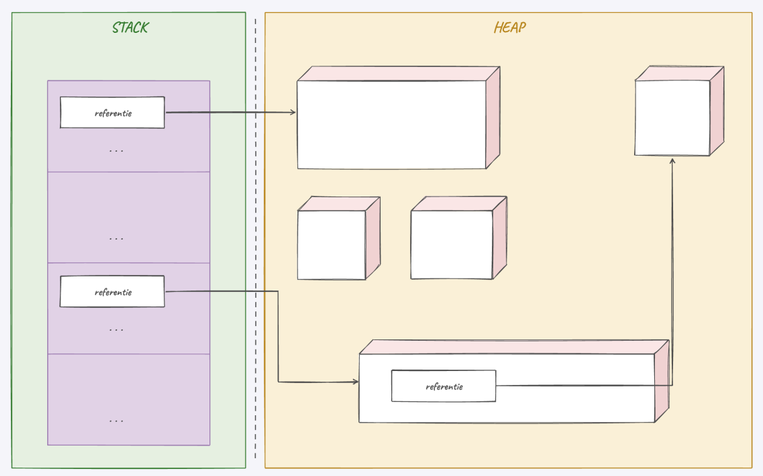
Een C#-programma krijgt twee geheugens:

- de kleine, snelle **stack**
- het grote, tragere **heap**

	Value types	**Reference types**
Inhoud	de waarde zelf	een verwijzing naar de data
Locatie	**stack**	**heap**
Default	`0`, `false`, …	`null`
`=` kopieert	de waarde	het **adres**

Waarom twee geheugens?

- De **stack** is bij het compileren al **exact berekenbaar** (snel, vaste grootte)
- De **heap** is voor data waarvan de grootte pas **tijdens de uitvoer** bekend is (arrays, objecten)



Stack en heap.

> De heap is de grote speelplaats; de stack het kleine, snelle bankje ernaast.

Value types: in de stack

= kopieert de **waarde**. Een methode krijgt een **kopie**:

```
void VerhoogParameter(int a)
{
    a++;
    Console.WriteLine($"In methode {a}");
}

int getal = 5;
VerhoogParameter(getal);
Console.WriteLine($"Na methode {getal}");
```

```
In methode 6
Na methode 5
```

Reference types: in de heap

```
1  Student stud = new Student();
```

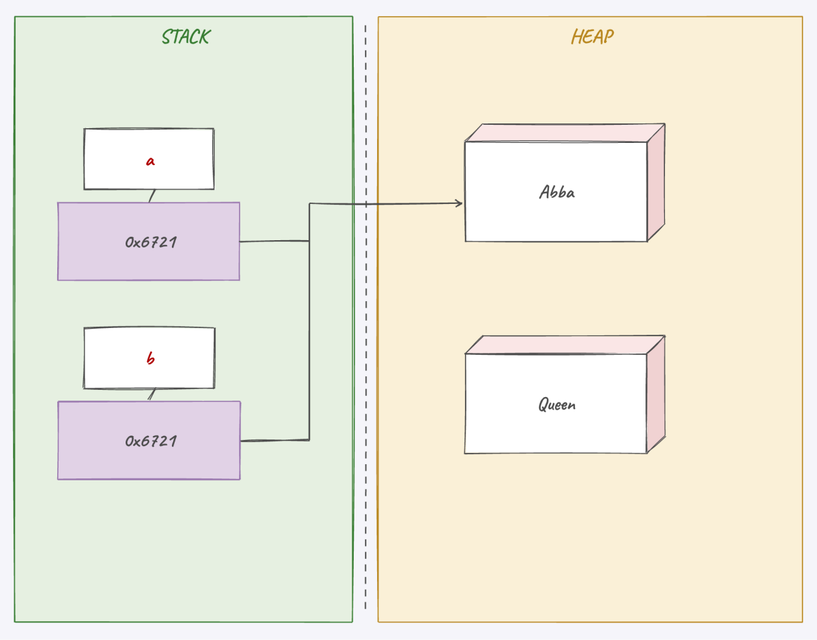
1. `new Student()` maakt het object in de **heap** en geeft het adres terug
2. de variabele `stud` staat in de **stack**
3. dat adres wordt in `stud` bewaard

> **Opmerking**
>
> Zonder `new` is `stud` nog `null`: een lege doos, zonder verwijzing.

De alias-val

```
1  Student a = new Student("Abba");
2  Student b = new Student("Queen");
3  b = a;
4  Console.WriteLine(a.Naam); // Abba
```

`b = a` kopieert het **adres**, niet het object.
Beide wijzen nu naar hetzelfde object;
“Queen” is onbereikbaar.

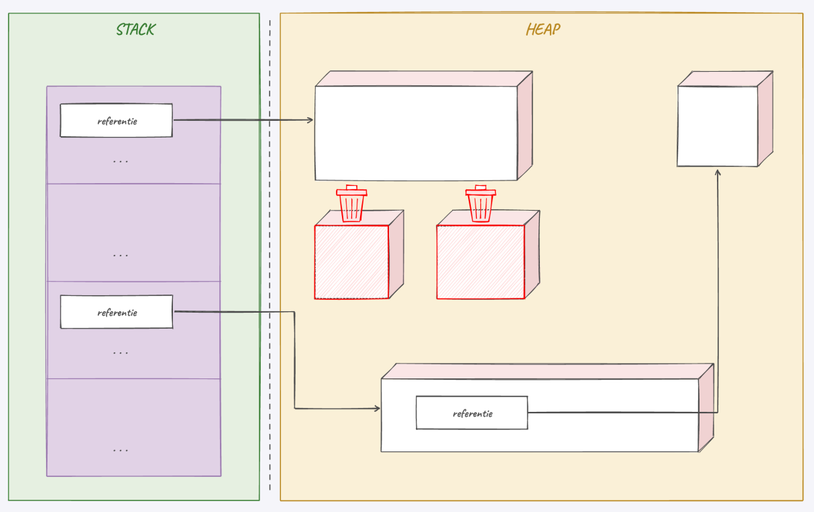


Beide variabelen wijzen naar hetzelfde object.

De Garbage Collector

Objecten in de heap waar **geen referentie** meer naar wijst, worden door de **GC** opgeruimd:

```
1  Held superman = new Held();
2  Held batman = new Held();
3  batman = superman; // 'batman'-object is nu wees
```



De GC ruimt onbereikbare objecten op.

> ⚠ **Waarschuwing**
>
> Roep **nooit** zelf `GC.Collect()` op. De GC kiest zelf het beste moment; jij maakt het meestal net trager.

string: een buitenbeentje

`string` is een **reference type**, maar gedraagt zich vaak als een value type omdat het **immutable** is:

```
1  string naam = "tim";
2  naam = naam.ToUpper();   // nieuwe tekst "TIM"
```

> **Tip**
>
> Methoden op een string geven steeds een **nieuwe** string terug. Vergeet je de toewijzing (`naam.ToUpper();`), dan verandert er niets.

Objecten in methoden

Objecten gaan **by reference** naar een methode. Aanpassingen werken op het **origineel**:

```
1  public void VoegMetingToeEnVerwijder(Meting inMeting)
2  {
3      Temperatuur += inMeting.Temperatuur;
4      inMeting.Temperatuur = 0;   // past het meegegeven object aan!
5  }
```

> **Opmerking**
>
> Een methode mag ook een object **teruggeven** (`return new Meting();`), bijvoorbeeld `PlantVoort()` die een nieuw `Mens`-object oplevert.

null en de NullReferenceException

```
1  Student stud1 = null;
2  Console.WriteLine(stud1.Naam); // crash
```

> ⚠ **Waarschuwing**
>
> *“Object reference not set to an instance of an object”*: de beruchte **`NullReferenceException`**. Je gebruikt een object dat nog niet met `new` is gemaakt.

null netjes afhandelen

```
1 if (stud1 != null)
2 {
3     Console.WriteLine(stud1.Naam);
4 }
5
6 Console.WriteLine(stud1?.Naam); // korter: enkel als niet null
```

- Het `?.` (null-conditional) voert de code enkel uit als het object niet `null` is
- Een methode mag bewust `null` **teruggeven** (bv. “niet gevonden”)

> **Opmerking**
>
> Groene kronkellijntjes rond `null`? Dat is de *nullable reference types*-functie. Die behandelen we bewust nog niet; je mag ze voorlopig negeren.

Namespaces

Een **`namespace`** voorkomt naamconflicten tussen klassen:

```
1  namespace MyEpicGame
2  {
3      internal class Monster { }
4  }
```

De volledige naam is `MyEpicGame.Monster`, dus een `Monster` uit een ander project (`NietZoEpicGame.Monster`) botst er niet mee.

De namespace-politie

De politie, uw vriend! Vraag je “waar is de Kerkstraat?”, dan vragen wij eerst: **in welke gemeente?**

Een namespace is net die gemeentenaam: ze laat toe een klasse (de straat) zonder verwarring te identificeren.

using

```
1  using System.Diagnostics;
```

- Zo zeg je: “zoek je een klasse en vind je ze niet, kijk dan ook in deze bibliotheek”
- Het VS-**lampje** vindt een ontbrekende `using` voor je
- Via **NuGet** voeg je extra bibliotheken toe (zoek eens `Colorful.Console`)

Exception handling: waarom?

Uitzonderingen liggen vaak **buiten** je controle:

- data die er niet is (verplaatst bestand, wegvallende wifi)
- foute invoer van de gebruiker
- programmeerfouten (deling door nul, een `null`-object)

> **Tip**
>
> Een onafgehandelde exception toont een **joekel van een foutmelding** en sluit je programma af. Exception handling laat je netjes reageren, zonder een woud aan `if`-checks.

try en catch

```
1  try
2  {
3      string input = Console.ReadLine();
4      int getal = Convert.ToInt32(input);
5  }
6  catch
7  {
8      Console.WriteLine("Verkeerde invoer!");
9  }
```

- In het **`try`**-blok staat code die kan mislukken
- Het **`catch`**-blok vangt de uitzondering op en laat je programma netjes verder werken

throw

Je kan zelf een uitzondering **opwerpen**:

```
1  throw new Exception("Wow, dit loopt fout");
```

> **Opmerking**
>
> Bij een `throw` klimt de uitvoering omhoog door de methode-aanroepen (*stack unwinding*) tot er een passend `catch`-blok klaarstaat. Daarom telt de **plaats** van je `try`/`catch`.

Meerdere catch-blokken

```
try { /* ... */ }
catch (FormatException e)
{
    Console.WriteLine("Verkeerd invoerformaat");
}
catch (Exception e)
{
    Console.WriteLine("Exception opgetreden");
}
```

Exceptions vormen een **hiërarchie**: `FormatException` is óók een `Exception`.

> ⚠ **Waarschuwing**
>
> **Specifiek eerst, algemeen laatst.** Zet je `catch (Exception)` bovenaan, dan compileert je code niet: het zou alles al opvangen.

De exception-parameter

```
1 catch (Exception e)
2 {
3     Console.WriteLine(e.Message);
4     Console.WriteLine(e.StackTrace);
5 }
```

`Message`, `StackTrace`, `TargetSite`, … geven info over de fout.

Belangrijk

Stuur deze info **niet zomaar naar de gebruiker**: ze kan gevoelige details bevatten die kwaadwillige gebruikers misbruiken.

Tip

Een **leeg** `catch`-blok is een anti-pattern: je verbergt bugs in plaats van ze op te lossen.

finally

```
try { /* ...kan mislukken */ }
catch (Exception ex) { Console.WriteLine(ex.Message); }
finally
{
    // loopt ALTIJD, bv. een bestand netjes sluiten
}
```

Het **`finally`**-blok draait altijd: of er nu een exception was of niet. Samen vormen `try`, `catch` en `finally` de drie-eenheid van exception handling.

Waar plaats je try/catch?

- **Rond de hele methode-aanroep**: bij de eerste fout stopt alles
- **Rond één stap in een loop**: een foute url wordt overgeslagen, de rest gaat door

```
1 for (int i = 0; i < urls.Length; i++)
2 {
3     try { /* download urls[i] */ }
4     catch (Exception ex) { Console.WriteLine(ex.Message); }
5 }
```

De plaats van je `try`/`catch` bepaalt dus hoe je programma op fouten reageert.

Conclusie

- Het geheugen splitst in een snelle **stack** (value types) en een flexibele **heap** (reference types)
- `=` kopieert bij objecten het **adres**, niet het object: pas op voor de alias-val
- De **Garbage Collector** ruimt onbereikbare objecten op; vergeet `new` niet (`null`!)
- `namespace` en `using` ordenen je klassen en voorkomen naamconflicten
- `try`/`catch`/`finally` vangt uitzonderingen op: specifiek eerst, algemeen laatst

> **Tip**
>
> Volgende halte: **gevorderde klasseconcepten**, met constructors en `static`.

Meer weten

Kennisclips

- Stack vs heap
- Referenties en null
- Objecten in methoden als parameter of returnwaarde
- Namespaces en using
- Werken met exception
- Stack Overflow Exception

Oefeningen H10 · 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: geheugen zelf beheren

C# heeft een GC. In **C++** beheer je geheugen zelf met `new` en `delete`:

```
1  int* p = new int(5);   // reserveren
2  delete p;              // zelf weer vrijgeven
```

Vergeet je `delete`, dan lek je geheugen (*memory leak*). De C#-GC neemt dat risico van je over.

Zie verder: “geen waarde”

Python noemt `null` gewoon `None`:

```python
student = None
if student is None:
    print("Nog geen student.")
```

JavaScript heeft er zelfs twee: `null` (bewust leeg) én `undefined` (nooit toegekend):

```javascript
let student = null;   // bewust leeg
let leeftijd;         // undefined
```

Zie verder: fouten in andere talen

Python kent ook exceptions, maar `catch` heet er `except`:

```python
try:
    getal = int(input())
except ValueError:
    print("Verkeerde invoer!")
```

De taal **C** heeft *geen* exceptions: je checkt zelf na elke aanroep een foutcode:

```c
int getal;
if (scanf("%d", &getal) != 1) {   // 1 = gelukt
    printf("Verkeerde invoer!\n");
}
```

Samenvattende poster

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