Arrays en klassen

Zie Scherp Scherper - Hoofdstuk 12

Tim Dams

Wat leer je in dit hoofdstuk?

Na dit hoofdstuk kan je:

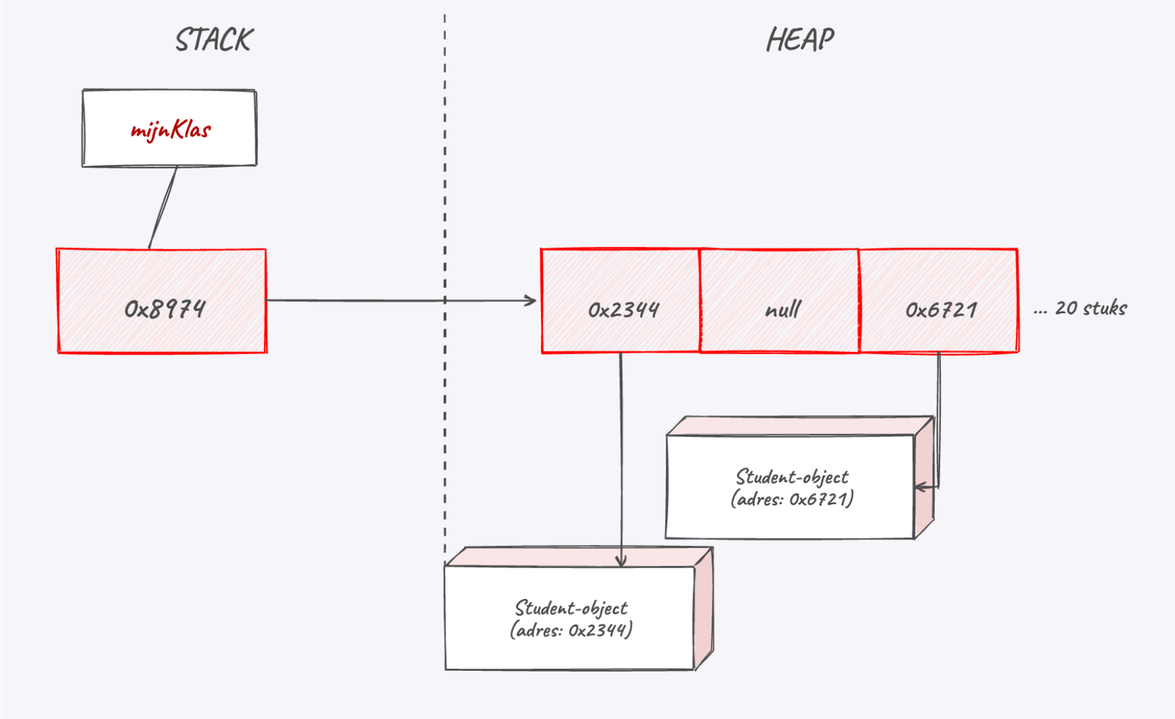
- een **array van objecten** aanmaken en de `null`-valkuil vermijden
- werken met de `List<>`-collectie als “array on steroids”
- een `foreach`-loop gebruiken (en zijn beperkingen kennen)
- het `var`-keyword correct inzetten
- de collecties `Queue`, `Stack` en `Dictionary` kiezen waar ze passen

Een array van objecten

```java
Student[] mijnKlas = new Student[20];
```

Belangrijk

`new` maakt enkel de **array** aan. Er staan nog **geen** objecten in: alle elementen zijn `null`.



Een lege array van referenties.

Objecten in de array plaatsen

```
for (int i = 0; i < mijnKlas.Length; i++)
{
    mijnKlas[i] = new Student();
}

mijnKlas[3].Naam = "Duke Peekaboo"; // via index + dot-operator
```

> ⚠ **Waarschuwing**
>
> Een nog niet aangemaakt element (`null`) benaderen geeft een **`NullReferenceException`**. Check eventueel met `mijnKlas[3]?.Naam`.

Array initializer syntax

Maak de objecten meteen aan; de compiler bepaalt de lengte:

```
Student[] mijnKlas = new Student[]
{
    new Student(),
    new Student(),
    jos              // een bestaand object mag ook
};
```

Opmerking

`jos` is nu via twee wegen bereikbaar: via `jos` én via `mijnKlas[2]`. Zolang er één referentie is, ruimt de GC het object niet op.

List: een array on steroids

```
1 List<int> getallen = new List<int>();
2 List<Pokemon> pokedex = new List<Pokemon>();
```

- Een **`List<>`** is een veiligere, krachtigere array
- Tussen `< >` staat het **type** dat de lijst bevat
- **Geen begingrootte** nodig: een list **groeit mee**

Elementen toevoegen

```
List<string> personages = new List<string>();
personages.Add("Reinhardt");
personages.Add("Mercy");

List<Pokemon> pokedex = new List<Pokemon>()
{
    new Pokemon() { Naam = "Pikachu", HP = 5 },
    new Pokemon() { Naam = "Bulbasaur", HP = 15 }
};
```

`Add()` voegt een element toe; object initializer syntax werkt ook hier.

List indexeren en overlopen

```
1  Console.WriteLine(personages[3]);
2  personages[2] = "Torbjorn";
3
4  for (int i = 0; i < personages.Count; i++)
5  {
6      Console.WriteLine(personages[i]);
7  }
```

Belangrijk

Een `List` werkt met **`Count`**, niet met `Length`.

Wat kan een List nog?

- `Clear()`, `Insert()`, `RemoveAt()`, `IndexOf()`, `Sort()`, …

> **Belangrijk**
>
> `IndexOf` en `Contains` vergelijken bij objecten de **referentie**. Twee aparte objecten met dezelfde inhoud gelden dus als **verschillend** (tot je `Equals` overridet, hoofdstuk 13).

> **Tip**
>
> **LINQ** (later) laat je in één leesbare regel filteren of sorteren (“alle Pokémon met meer dan 100 HP”). Voorlopig doen we dat nog met loops.

Pong met een List

```
List<Balletje> veelBalletjes = new List<Balletje>();
for (int i = 0; i < 100; i++)
{
    veelBalletjes.Add(new Balletje());
}

while (true)
{
    foreach (var bal in veelBalletjes) bal.Update();
    foreach (var bal in veelBalletjes) bal.TekenOpScherm();
    Thread.Sleep(50); Console.Clear();
}
```

Leesbaarder dan met arrays, en de lijst groeit en krimpt vanzelf.

foreach: de vierde loop

Wil je **alle** elementen overlopen zonder index?

```
1 double[] metingen = { 1.2, 0.89, 3.15, 0.1 };
2 foreach (double meting in metingen)
3 {
4     Console.WriteLine(meting);
5 }
```

De **iteration variable** (`meting`) krijgt elk element om de beurt. Geen teller nodig, en werkt ook op een `List`.

foreach: drie aandachtspunten

1. De iteration variable is **read-only**: je kan de array zelf niet wijzigen
2. Enkel als je **alle** elementen wil (anders een andere loop)
3. **Geen teller** beschikbaar; maak die zelf als je hem nodig hebt

> **Belangrijk**
>
> Read-only slaat op de **referentie**. Het object zelf mag je wél aanpassen: `student.Geboortejaar++;` mag, `student = new Student();` niet.

foreach: niet wijzigen tijdens iteratie

```
1  foreach (var student in deKlas)
2  {
3      if (student.Geboortejaar < 2000)
4          deKlas.Remove(student); // FOUT
5  }
```

> ⚠ **Waarschuwing**
>
> *"Collection was modified; enumeration operation may not execute."* Wil je tijdens het doorlopen verwijderen, gebruik dan een gewone `for`-loop **van achter naar voor**.

Het var-keyword

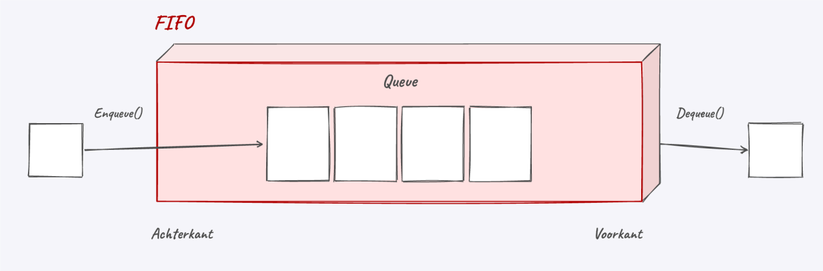
`var` laat de compiler het type **afleiden** uit de rechterkant:

```
1  var getal = 5;                 // int
2  var tekst = "Hi there";        // string
3  var ikke = new Leerkracht();   // Leerkracht
```

> **Tip**
>
> In C# is dit puur **lazy syntax**: het type ligt nog steeds vast (statically typed). Anders dan in JavaScript, waar `var` een echt variabel type betekent.

Queue: first in, first out

```
1  Queue<string> rij = new Queue<string>();
2  rij.Enqueue("Ik eerst");
3  rij.Enqueue("Ik tweede");
4  Console.WriteLine(rij.Dequeue()); // Ik eerst
```



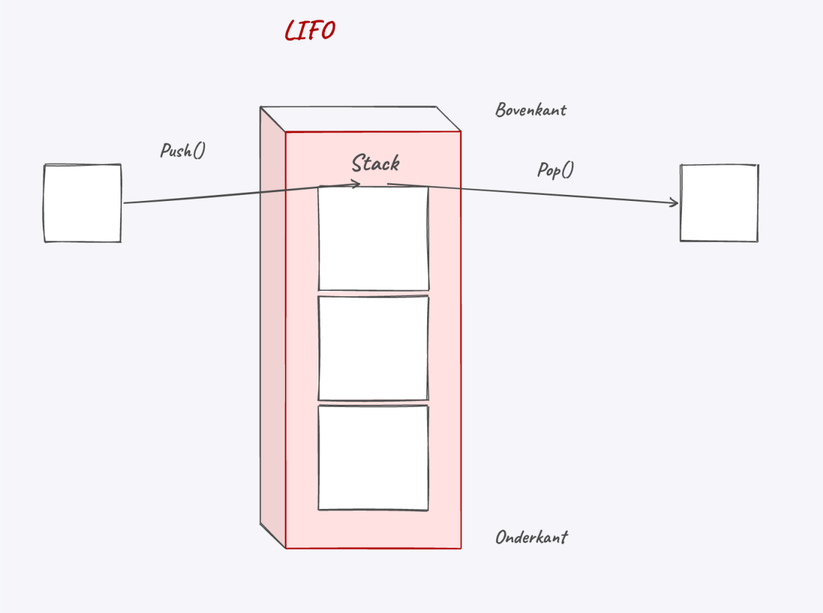
Een wachtrij (FIFO).

`Enqueue` zet achteraan bij, `Dequeue` haalt vooraan af. Net als de rij aan de kassa.

Stack: last in, first out

```
1  Stack<string> stapel = new Stack<string>();
2  stapel.Push("Ik eerst");
3  stapel.Push("Ik laatste");
4  Console.WriteLine(stapel.Pop()); // Ik laatste
```

`Push` legt bovenop, `Pop` neemt bovenaf.
Net als een stapel papier.

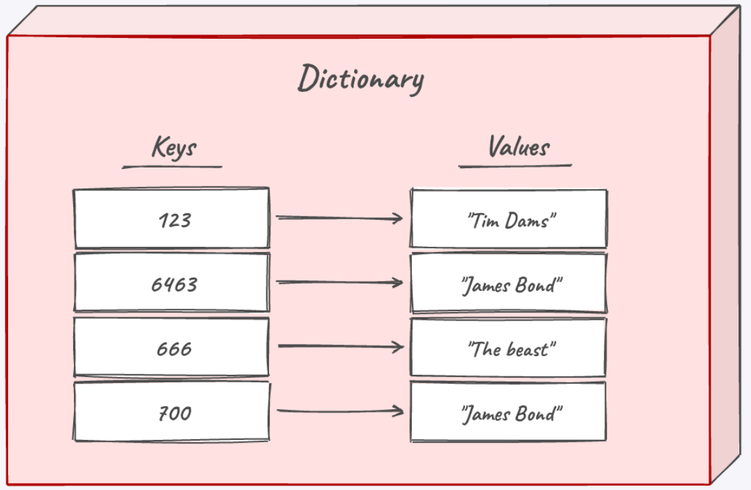


Een stapel (LIFO).

Dictionary: sleutel en waarde

Elk element heeft een **unieke key** en een **value**:

```
1  Dictionary<int, string> klanten = new Dictionary<
2  klanten.Add(123, "Tim Dams");
3  klanten.Add(666, "The beast");
4
5  Console.WriteLine(klanten[123]); // Tim Dams
```



Sleutel-waarde-paren.

> **Tip**
>
> Opzoeken via de key gaat **razendsnel**, ongeacht de grootte. Veel sneller dan zoeken in een `List`.

Dictionary doorlopen

```
1  foreach (var item in klanten)
2  {
3      Console.WriteLine($"{item.Key}\t: {item.Value}");
4  }
```

Elk item heeft een `.Key` en een `.Value` van de types die je bij de declaratie koos.

> **Belangrijk**
>
> Gebruik voorlopig een eenvoudig type (`int`, `string`) als key. Een eigen klasse als key vereist `Equals` en `GetHashCode` (hoofdstuk 13).

Conclusie

- Een **array van objecten** start vol `null`: maak elk object eerst met `new`
- Een `List<>` groeit mee, werkt met `Count` en vol handige methoden
- `foreach` overloopt alle elementen leesbaar, maar de iteration variable is **read-only**
- `var` laat de compiler het type afleiden (C# blijft statically typed)
- `Queue` (FIFO), `Stack` (LIFO) en `Dictionary` (key-value) voor speciale gevallen

> **Tip**
>
> Volgende halte: **overerving**, waar klassen eigenschappen en gedrag van elkaar erven.

Meer weten

Kennisclips

- Arrays van objecten
- var keyword
- De foreach loop
- List<> klasse
- Werken met Dictionary<>

Oefeningen H12 · 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: List in andere talen

In **Python** is een `list` ingebouwd en typeloos:

```
1  namen = ["Tim", "An"]
2  namen.append("Jan")
```

In **Java** lijkt het sterk op C#: `ArrayList<String>` met `.add(...)`.

Zie verder: foreach in andere talen

In **Python** schrijf je net hetzelfde idee, zonder type en accolades:

```python
for meting in metingen:
    print(meting)
```

In **JavaScript** heet dit `for...of`:

```javascript
for (const meting of metingen) {
    console.log(meting);
}
```

Zie verder: Dictionary in andere talen

In **Python** heet dit een `dict`, ingebouwd met een handige `{}`-syntax:

```python
klanten = {123: "Tim Dams", 6463: "James Bond"}
print(klanten[123])
```

In **JavaScript** gebruik je een `Map` met `set` en `get`:

```javascript
const klanten = new Map();
klanten.set(123, "Tim Dams");
console.log(klanten.get(123));
```

Samenvattende poster

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