



// 01

Een array aanmaken

Een reeks waarden van **hetzelfde datatype** in één variabele. Zodra de array bestaat, ligt haar **lengte vast**.

```
int[] punten;           // nog geen array
punten = new int[] {8, 12, 5}; // new nodig
int[] cijfers = {8, 12, 5}; // bij declaratie
int[] scores = new int[3]; // 0, 0, 0
```

Een nieuwe array staat vol met de **defaultwaarde**: **0** bij getallen, **false** bij **bool**, **null** (nog niets) bij **string**.

// 02

Index, Length en for

Met `naam[index]` lees of schrijf je één element. De index start bij **0**: lengte 5 betekent de indices 0 tot en met 4.

```
int[] getallen = {10, 20, 30};
getallen[0] = 15; // schrijven
for (int i = 0; i < getallen.Length; i++)
{
    Console.WriteLine(getallen[i]); // lezen
}
```

// LET OP

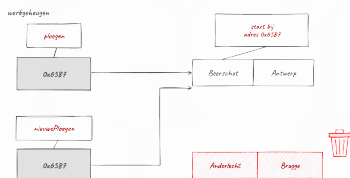
`getallen[5]` bij lengte 3 of `i <= getallen.Length` compileert, maar crasht: `IndexOutOfRangeException`.

// 03

Referenties en kopiëren

Een array-variabele bewaart niet de array zelf, maar haar **adres** in het geheugen: arrays zijn **reference types**.

```
string[] ploegen = {"Beerschot", "Antwerp"};
string[] nieuwePloegen = {"Anderlecht", "Brugge"};
nieuwePloegen = ploegen; // kopieert het adres
```



Een echte kopie: elk element apart met een `for`, of `Array.Copy`:

```
string[] kopie = new string[ploegen.Length];
Array.Copy(ploegen, kopie, ploegen.Length);
```

// LET OP

`nieuwePloegen = ploegen`; maakt geen kopie maar een alias: beide variabelen wijzen naar dezelfde array.

// 04

System.Array

METHODE	DOET
<code>Array.Sort(arr)</code>	sorteert de array zelf
<code>Array.Reverse(arr)</code>	keert de volgorde om
<code>Array.Clear(arr, 0, 2)</code>	2 elementen vanaf index 0 op default
<code>Array.IndexOf(arr, x)</code>	index van de eerste <code>x</code> , of <code>-1</code>
<code>Array.BinarySearch(arr, x)</code>	snel zoeken, negatief: niet gevonden

// LET OP

`BinarySearch` werkt enkel op een **gesorteerde** array. Anders krijg je geen fout, maar soms een fout antwoord.

// 05

Zelf zoeken en tellen

Maak de variabele voor je resultaat aan **vóór** de loop.

```
int[] punten = {12, 8, 17, 15, 4};
int som = 0;
for (int i = 0; i < punten.Length; i++)
{
    som += punten[i];
}
Console.WriteLine((double)som / punten.Length);
```

UITVOER
11,2

- **Grootste zoeken**: start met `punten[0]`, niet met `0`
- **Zoeken en stoppen**: `break` zodra je het hebt, `-1` voor "niet gevonden"
- **Synchrone arrays**: `prijzen[i]` hoort bij `producten[i]`

// 06

Strings en arrays

Een string lees je als een array van `char`. Schrijven (`zin[0] = 'i';`) kan niet: een string ligt vast.

```
string zin = "Ik ben Tim";
char eerste = zin[0]; // 'I'
bool cijfer = char.IsDigit(zin[1]); // false
string[] woorden = zin.Split(' '); // 3 woorden
Console.WriteLine(string.Join("-", woorden));
```

UITVOER

Ik-ben-Tim

// LET OP

Een string-methode geeft een **nieuwe** string terug. `boek.Trim()`; alleen doet niets: schrijf `boek = boek.Trim()`;

// 07

Arrays en methoden

Als parameter of returntype schrijf je `int[]`, zonder grootte. In de methode vraag je `Length` op.

```
static void PasAan(int[] inarr)
{
    inarr[0] = 0;
}
```

Met `params int[] getallen` als laatste parameter mag je `Som(1, 2, 3)` of `Som()` schrijven. `Main(string[] args)` krijgt de opstartparameters: test eerst `args.Length`.

// LET OP

Geef je een array mee aan een methode, dan werk je op het origineel: wat `PasAan` aanpast, blijft na de aanroep bestaan.

// 08

Meerdimensionale arrays

Een rooster, zoals een spelbord of een doolhof.

```
int[,] regen = new int[4, 7]; // 4 rijen
regen[2, 4] = 12; // [rij, kolom]
int rijen = regen.GetLength(0); // 4
int kolommen = regen.GetLength(1); // 7
```

Overlopen doe je met twee geneste `for`-loops: buiten de rijen, binnen de kolommen. `regen.Length` is 28, alle vakjes samen. Een **jagged array** `double[][]` is een array van arrays met elk een eigen lengte (`tickets[0][1]`): vooral herkennen.