C# Essentials - Verzamelingen

Inhoudsopgave

1. Introductie

1.1. Waarom collections?

Stel je voor dat je een lijst van 100 studenten wil opslaan. Zou je dan 100 aparte variabelen moeten aanmaken?

```
string student1 = "Alice";
string student2 = "Bob";
// ...
string student100 = "Zoe";
```

Dat is natuurlijk niet werkbaar. In zo'n geval gebruik je een **collection**: een soort "verzameling" van meerdere waarden onder één naam.

1.2. Arrays, Lists en Dictionaries

C# kent verschillende types van collecties. Dit zijn de drie bekendste:

- `array` - vaste grootte, snel en eenvoudig
- `List<T>` - flexibel en krachtig, meest gebruikte type
- `Dictionary<TKey, TValue>` - handig als je wil zoeken op een **sleutel**

Een array gebruik je bijvoorbeeld als je weet dat je altijd 10 scores nodig hebt. Een List is beter als je items wil toevoegen of verwijderen. Een Dictionary gebruik je wanneer je snel toegang wil tot data via een sleutel, zoals een studentennummer.

1.3. Voorbeeld

Een array van integers:

```
int[] scores = new int[] { 10, 8, 7, 9 };
```

Een lijst van namen:

```
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");
```

Een dictionary met studentnummer als sleutel:

```
Dictionary<int, string> students = new Dictionary<int, string>();
students.Add(123, "Alice");
students.Add(456, "Bob");
```

1.4. Samenvatting

Type	Uitleg	Kan je elementen toevoegen?
`array`	Vast aantal elementen, indexgebaseerd	❌
`List<T>`	Dynamisch aantal, eenvoudig in gebruik	✅
`Dictionary<K,V>`	Gegevens opzoeken via een sleutel	✅

1.5. Andere collecties

In deze cursus focussen we op de drie belangrijkste types: `array`, `List<T>` en `Dictionary<K,V>`. Maar er bestaan ook andere collecties zoals:

- `HashSet<T>` - een lijst zonder dubbele waarden
- `Queue<T>` - elementen in volgorde van aankomst (FIFO - first in first out)
- `Stack<T>` - laatste toegevoegd is eerste eruit (LIFO - last in first out)
- `ObservableCollection<T>` - Voor WPF-binding (meldt wijzigingen aan UI)
- `ReadOnlyCollection` - Alleen-lezen-wrapper rond een lijst

Deze komen niet aan bod in deze cursus, maar als je later verder programmeert, ga je ze zeker tegenkomen.

2. Arrays

2.1. Wat is een array?

- Een array is een verzameling van **vaste grootte** die meerdere waarden van hetzelfde type bevat.
- Elke waarde van een array wordt ook wel een **element** genoemd.
- De elementen worden automatisch genummerd vanaf 0.

2.2. Een array aanmaken

Om een array aan te maken, gebruik je vierkante haken (`[]`). Dat is de manier waarop je in C# zegt: "ik wil meerdere elementen van dit type opslaan".

Bijvoorbeeld:

```
int[] scores;   // Declaratie van een nieuwe array
```

Hier geef je aan dat `scores` geen gewoon getal is, maar een **reeks van int-waarden**. Je kan je dat voorstellen als een rij vakjes in het geheugen:

```
+-----+-----+-----+-----+
|  ?  |  ?  |  ?  |  ?  |
+-----+-----+-----+-----+
   0     1     2     3
```

- Elk vakje in de array bevat een waarde van hetzelfde type (in dit geval: `int`).
- Elk vakje heeft een nummer: dat is de *index*. **De eerste index is altijd `0`.**

Om zo'n array effectief aan te maken en plaats te reserveren in het geheugen, gebruik je het sleutelwoord `new`:

```
scores = new int[4];
// maakt plaats voor 4 gehele getallen
```

Je kan dit ook in één lijn schrijven:

```
int[] scores = new int[4];
```

2.2.1. Oefening: Juist of fout?

Bepaal voor elke lijn of deze correct is in C#. Noteer voor jezelf of ze **juist** of **fout** is.

```
1. int[] numbers = new int[3];
2. string[] names = { "Alice", "Bob" };
3. bool values = new bool[] { true, false };
4. double[] data = new double[] { 1.0, 2.5, 3 };
5. int[] items = new int[] { "één", "twee" };
6. var cities = new string[2];
7. int[] scores = new int[] { };
8. int[5] result = new int[];
```

 Oplossing

Lijn	/	Uitleg
1		Geldige declaratie en creatie van array met vaste grootte.
2		Correcte korte notatie van een string-array.
3		Linkerkant is `bool`, maar rechterkant is een `bool[]` (array).
4		Geldige array van doubles met correcte waarden.
5		`"één"` en `"twee"` zijn strings, maar het type is `int[]`.

Lijn	✅/❌	Uitleg
6	✅	`var` werkt hier, omdat de compiler kan afleiden dat het een `string[]` is.
7	✅	Lege array aanmaken is toegestaan.
8	❌	Verkeerde volgorde en syntaxis bij declaratie en initialisatie.

2.3. Waarden invullen

2.3.1. Met behulp van de index

Je kan nu elk vakje afzonderlijk invullen met behulp van de index:

```
scores[0] = 8;
scores[1] = 10;
scores[2] = 6;
scores[3] = 9;
```

Een array van 4 elementen heeft de indexen `0`, `1`, `2` en `3`. Als je probeert `scores[4]` te gebruiken, krijg je een foutmelding: `IndexOutOfRangeException`.

2.3.2. Directe initialisatie

Je kan ook alle waarden al meegeven op het moment dat je de array aanmaakt. Dat kan op 4 manieren:

```
int[] scores = new int[4] { 8, 10, 6, 9 };
```

Of iets korter:

```
int[] scores = new int[] { 8, 10, 6, 9 };
```

Of nog iets korter:

```
int[] scores = { 8, 10, 6, 9 };
```

Deze korte vorm mag alleen als je **tegelijkertijd** de variabele declareert en initialiseert. Je mag dus niet eerst `int[] scores;` schrijven en later pas `{ 8, 10, 6, 9 }` gebruiken zonder `new`.

```
int[] scores;
```

```
scores = { 8, 10, 6, 9 };  //Dit mag dus niet
```

Of met een collection expression:

```
int[] scores = [8, 10, 6, 9];
```

Deze syntax werd pas geïntroduceerd in C# 12 (2024). Je kan er lijsten en arrays korter mee initialiseren.

2.4. Lengte van een array

Je kan de lengte van een array opvragen met de `.Length`-eigenschap:

```
Console.WriteLine(scores.Length); // toont 4
```

2.5. Waarden lezen

2.5.1. Met de index

Met behulp van de index kan je eender welke waarde van een array lezen:

```
int highestScore = scores[1];  // 10
int lowestScore = scores[2];   // 6
```

2.5.2. Met een `for`-lus

Je kan met een `for`-lus elk element overlopen:

```
for (int i = 0; i < scores.Length; i++)
{
    Console.WriteLine(scores[i]);
}
```

2.5.3. Met een `foreach`-lus

Wil je gewoon elk element tonen, dan is `foreach` eenvoudiger:

```
foreach (int element in scores)
{
    Console.WriteLine(element);
}
```

2.6. Arrays zijn statisch

Een array heeft altijd een vaste lengte. Je kan geen elementen toevoegen of verwijderen. Als je dat wel

wil doen, gebruik je beter een `List<T>` (zie volgend hoofdstuk).

2.7. Oefening

Schrijf een consoleapplicatie die een array van 4 namen initialiseert en deze daarna één voor één afdrukt in het consolevenster.

Oplossing

```
string[] names = { "Alice", "Bob", "Charlie", "Diana" };
foreach (string element in names)
{
    Console.WriteLine(element);
}
```

2.8. Array als parameter

Je kan een array ook gebruiken als parameter van een methode, net zoals je dat met een ander datatype zou doen:

```
static void Main(string[] args)
{
    string[] lectors = { "Koen", "Wim", "Sander" };
    PrintNames(lectors);
}

private static void PrintNames(string[] names)
{
    foreach (string element in names)
    {
        Console.WriteLine(element);
    }
}
```

2.9. Array van objecten

Je kan ook een array van objecten maken.

Bijvoorbeeld: een array van studenten.

```
class Student
{
    public string FirstName;
    public string LastName;
}

Student[] students = new Student[3];

students[0] = new Student { FirstName = "Alice", LastName = "Janssen" };
```

```
students[1] = new Student { FirstName = "Bob", LastName = "Peeters" };
students[2] = new Student { FirstName = "Charlie", LastName = "Vermeulen" };

foreach (Student s in students)
{
    Console.WriteLine($"{s.FirstName} {s.LastName}");
}
```

2.10. Van tekst naar array met string.Split()

Gebruikers geven vaak gegevens in als één lange string (bijvoorbeeld: "Peter,Anna,Tom"). Met `string.Split()` kan je zo'n tekst in één stap opsplitsen naar een array van kleinere stukjes. Zo kan je de elementen daarna verwerken zoals elke andere array.

`Split()` is dus belangrijk omdat het een brug vormt tussen tekstinvoer en arrays.

```
string input = "Peter,Anna,Tom";
string[] names = input.Split(',');

// Resultaat:
// ["Peter", "Anna", "Tom"]
```

Je kan er ook voor kiezen om lege elementen te verwijderen uit het resultaat:

```
string input = "Peter,,Anna,,,Tom";
string[] names = input.Split(',', StringSplitOptions.RemoveEmptyEntries);

// Resultaat met RemoveEmptyEntries:
// ["Peter", "Anna", "Tom"]
```

2.11. Samenvatting

Kenmerk	Beschrijving
Vaste grootte	Aantal elementen ligt vast bij creatie
Indexering	Eerste element zit op index 0
Doorlopen	Via `for` of `foreach`
Type	Alle elementen moeten van hetzelfde type zijn
Lengte	Opvragen via `.Length` property


If you love me, meet me at 1st table
TABLE 00
TABLE 01
Array Indexing (crash course)
imgflip.com

3. De Array-klasse

3.1. De `Array`-klasse

Een array is intern gebaseerd op de `System.Array`-klasse. Die bevat een hele reeks handige methodes waarmee je een array kan manipuleren.

3.2. Sorteren

Met `Array.Sort()` kan je een array alfabetisch of numeriek sorteren **van klein naar groot**:

```
string[] names = { "Wouter", "Paul", "Andreas", "Silvia" };
Array.Sort(names);

foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Resultaat:

```
Andreas
Paul
Silvia
Wouter
```

3.2.1. Sorteren van een deel van de array

Je kan ook maar een deel sorteren:

```
string[] names = { "Peter", "Tom", "Anna", "Sven", "Bob" };
Array.Sort(names, 1, 3); // sorteert enkel index 1, 2 en 3
```

Resultaat:

```
Peter
Anna
Sven
Tom
Bob
```

3.3. Omkeren

Je kan de volgorde van de elementen omdraaien met `Array.Reverse()`:

```
int[] scores = { 4, 6, 10, 2 };
Array.Reverse(scores);

foreach (int score in scores)
```

```
{
    Console.WriteLine(score);
}
```

Resultaat:

```
2
10
6
4
```

3.4. Sorteren én omkeren

Wil je sorteren van groot naar klein? Combineer dan eerst `Sort` en dan `Reverse`:

```
int[] values = { 4, 6, 10, 2 };
Array.Sort(values);
Array.Reverse(values);

foreach (int v in values)
{
    Console.WriteLine(v);
}
```

Resultaat:

```
10
6
4
2
```

Het is belangrijk dat je de array **eerst sorteert** en daarna pas omkeert!

3.5. Kopiëren

Met `Array.Copy()` kan je de inhoud van een array kopiëren naar een andere:

```
string[] original = { "Alice", "Bob", "Charlie" };
string[] copy = new string[original.Length];
Array.Copy(original, copy, original.Length);

foreach (string name in copy)
{
    Console.WriteLine(name);
}
```

Resultaat:

```
Alice
Bob
Charlie
```

3.6. Positie van een element zoeken

Met `Array.IndexOf()` zoek je naar de **eerste** positie van een bepaald element:

```
string[] names = { "Tom", "Bob", "Tom", "Anna" };
int index = Array.IndexOf(names, "Tom");
Console.WriteLine(index);
```

Resultaat:

```
0
```

Met `Array.LastIndexOf()` vind je de laatste positie:

```
int lastIndex = Array.LastIndexOf(names, "Tom");
Console.WriteLine(lastIndex);
```

Resultaat:

```
2
```

3.6.1. Wat als het element niet bestaat?

Als het element dat je zoekt niet in de array zit, dan geeft `IndexOf()` of `LastIndexOf()` `-1` terug.

```
int index = Array.IndexOf(names, "Elise");
Console.WriteLine(index);
```

Resultaat:

```
-1
```

Gebruik altijd een controle zoals `if (index != -1)` voordat je de index gebruikt. Anders krijg je een fout als je probeert te werken met een onbestaand element.

3.7. Elementen zoeken

Met `Array.Find()` en `Array.FindAll()` kan je zoeken op basis van een voorwaarde. Dat werkt met een **predicate**: een stukje code dat bepaalt of een element wel of niet geschikt is.

3.7.1. Wat is een predicate?

Een predicate is een methode die als resultaat `true` of `false` geeft. Bijvoorbeeld:

```
bool StartsWithP(string word)
{
    return word.StartsWith("P");
}
```

Je kan deze methode dan gebruiken als voorwaarde in `Array.Find()`:

```
string[] consoles = { "PS2", "XBox", "Gamecube", "N64", "PS5" };

string result = Array.Find(consoles, StartsWithP);
Console.WriteLine(result);
```

Resultaat:

```
PS2
```

Net zoals de `IndexOf`-methode geeft ook de `Find`-methode enkel het eerste element terug wat voldoet aan de voorwaarde

3.7.2. Lambda-expressie (korte vorm)

In plaats van een aparte methode te maken, kan je ook een **lambda-expressie** gebruiken. Dat is een kortere manier om een voorwaarde te schrijven, meestal in één regel:

```
string result = Array.Find(consoles, element => element.StartsWith("P"));
Console.WriteLine(result);
```

Resultaat:

```
PS2
```

Hier betekent `element ⇒ element.StartsWith("P")` eigenlijk hetzelfde als:

```
bool check(string element)
{
    if (element.StartsWith("P"))
    {
        return true;
    }
    else
    {
        return false;
```

```
    }
}
```

3.7.3. Alle resultaten zoeken

Met `Array.FindAll()` krijg je niet slechts één resultaat, maar een nieuwe array met alle elementen die aan de voorwaarde voldoen:

```
string[] result = Array.FindAll(consoles, element => element.Length == 3);

foreach (string console in result)
{
    Console.WriteLine(console);
}
```

```
PS2
N64
PS5
```

De `=>`-notatie heet een **lambda-expressie**. Dit is nieuw en kan in het begin wat vreemd lijken, maar het is een erg krachtige en leesbare manier om voorwaarden te schrijven.

3.8. Bestaat het element?

Met `Array.Exists()` kan je testen of een element aan een bepaalde voorwaarde voldoet:

```
string[] consoles = { "PS2", "XBox", "Gamecube", "N64", "PS5" };

bool hasDreamcast = Array.Exists(consoles, element =>
element.Equals("Dreamcast")); // False
bool hasGamecube = Array.Exists(consoles, element =>
element.Equals("Gamecube"));   // True
```

3.9. Array wissen

Met `Array.Clear()` kan je een deel van de array op 0 (of `null`) zetten:

```
int[] numbers = { 5, 6, 7, 8, 9 };
Array.Clear(numbers, 1, 3);

foreach (int n in numbers)
{
    Console.WriteLine(n);
}
```

Resultaat:

```
5
0
0
0
9
```

`Array.Clear()` zet de opgegeven elementen op hun standaardwaarde: `0` voor getallen, `false` voor booleans en `null` voor objecten of strings. **De lengte van de array wordt hierdoor niet gewijzigd!**

```
string[] names = { "Alice", "Bob", "Charlie", "Diana", "Elise" };
Array.Clear(names, 1, 3);

foreach (string name in names)
{
    Console.WriteLine(name);
}
```

```
Alice
(null)
(null)
(null)
Elise
```

De lege regels in de output betekenen dat het element null is. Als je Console.WriteLine(null) doet, krijg je gewoon een lege regel.

3.10. `Array.Resize()` ⁉️

Je kan een array niet echt groter maken. Maar met `Array.Resize()` kan je een nieuwe array maken met andere grootte:

```
int[] values = { 1, 2, 3 };
Array.Resize(ref values, 5); // array is nu {1, 2, 3, 0, 0}
```

De `Resize`-methode maakt eigenlijk een nieuwe array aan, en kopieert de oude waarden.

3.11. Voorbeeld met objecten

```
class Product
{
```

```
    public string Name;
    public decimal Price;
}

Product[] products = new Product[]
{
    new Product { Name = "Laptop", Price = 999 },
    new Product { Name = "Mouse", Price = 25 },
    new Product { Name = "Keyboard", Price = 49 }
};

Array.Sort(products, (a, b) => a.Price.CompareTo(b.Price));

foreach (Product p in products)
{
    Console.WriteLine($"{p.Name} - {p.Price} EUR");
}
```

3.12. Samenvatting

Methode	Doel
`Array.Sort()`	Sorteert de array (alfabetisch of numeriek)
`Array.Reverse()`	Keert de volgorde van elementen om
`Array.Copy()`	Kopieert elementen naar een andere array
`Array.IndexOf()`	Geeft de eerste index van een bepaald element
`Array.Find()` / `FindAll()`	Zoekt met een voorwaarde
`Array.Exists()`	Controleert of minstens één element voldoet
`Array.Clear()`	Leegt (een deel van) de array
`Array.Resize()`	Past de grootte van de array aan (door een nieuwe te maken)

4. Generic List<T>

4.1. Waarom `List<T>` gebruiken?

Arrays hebben een vaste lengte. Wil je een element toevoegen, dan moet je een nieuwe array maken en alles kopiëren. Dat is omslachtig.

Met een `List<T>` gaat dat véél eenvoudiger:

```
List<int> scores = new List<int>();
scores.Add(10);
scores.Add(8);
scores.Add(9);
```

4.2. Generics

Een lijst in C# maak je aan met `List<T>`, waarbij je zelf kiest welk type (`T`) de lijst zal bevatten.

Die `<T>`-schrijfwijze is typisch voor **generics**. Dat is een moeilijk woord voor iets heel eenvoudigs: je wil een lijst die werkt voor **eender welk type**. In plaats van 100 aparte lijstsoorten te maken (`ListOfInt`, `ListOfString`, …), maakt C# één soort lijst met een invulvakje: `List<T>`.

Die `T` is geen verplichte letter, het is een aanduiding. Je kan zelf kiezen wat je invult. Bijvoorbeeld:

```
// Lijst van getallen
List<int> values = new List<int>();

// Lijst van strings
List<string> names = new List<string>();

// Lijst van objecten
List<Student> students = new List<Student>();
```

Je gebruikt dus altijd `List<T>` met tussen de `<…>` het type dat je wil opslaan.

Generics zorgen ervoor dat je een herbruikbare klasse hebt die voor elk type kan werken, zonder dat je telkens een nieuwe versie moet maken.

4.3. Een lijst aanmaken

Je moet steeds `using System.Collections.Generic;` toevoegen bovenaan je bestand.

Er zijn meerdere manieren om een lijst te initialiseren:

```
// Lege lijst
List<string> names = new List<string>();
```

```
// Meteen met inhoud
List<int> values = new List<int> { 5, 10, 15 };

// Vanuit een array
int[] ages = { 18, 20, 22 };
List<int> ageList = new List<int>(ages);
```

4.3.1. Oefening: Juist of fout?

Bepaal voor elke lijn of deze correct is in C#. Noteer voor jezelf of ze **juist** of **fout** is.

```
1. List<int> scores = new List<int>();
2. List<int> numbers = new List<string>();
3. List<string> names = new List<string> { "Alice", "Bob" };
4. List values = new List<int>();
5. List<Student> students = new List<Student>();
6. List<bool> flags = new List<bool> { true, false, "ja" };
7. List<DateTime> dates = new List<DateTime>();
8. List<float> numbers = new List<float> { 1.0f, 2.5f };
```

 Oplossing

Lijn	/	**Uitleg**
1		Correcte declaratie en initialisatie van een lijst van integers.
2		Linkerzijde is `List<int>`, rechterzijde maakt een `List<string>`.
3		Geldige lijst van strings met correcte inhoud.
4		Het type `List` is niet volledig: je moet altijd `<T>` specificeren.
5		Correct gebruik van een lijst van objecten (hier: `Student`).
6		`"ja"` is geen `bool`, dit geeft een compile-error.
7		`DateTime` is een geldig .NET-type, dus dit is correct.
8		De `f` achter de getallen is nodig voor `float`, dus dit is juist.

4.4. Element opvragen via index

Net zoals bij arrays kan je via een index een element opvragen:

```
List<string> cities = new List<string> { "Paris", "London", "Rome" };
```

```
string secondCity = cities[1];
Console.WriteLine(secondCity);
```

Resultaat:

```
London
```

De eerste index is 0. Als je een ongeldige index gebruikt, krijg je een `ArgumentOutOfRangeException`.

Je kan ook elementen overschrijven op een bepaalde positie:

```
cities[2] = "Berlin";

foreach (string city in cities)
{
    Console.WriteLine(city);
}
```

Resultaat:

```
Paris
London
Berlin
```

4.5. Elementen toevoegen

Met `Add()` voeg je één element toe. Met `AddRange()` voeg je meerdere elementen toe:

```
List<int> numbers = new List<int>();
numbers.Add(5);
numbers.Add(10);

int[] extra = { 15, 20 };
numbers.AddRange(extra);

foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Resultaat:

```
5
10
15
```

```
20
```

4.6. Aantal elementen opvragen

Met de `Count`-property weet je hoeveel elementen er in een lijst zitten.

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
Console.WriteLine(names.Count);
```

Resultaat:

```
3
```

Je kan `Count` ook gebruiken in een `for`-lus:

```
for (int i = 0; i < names.Count; i++)
{
    Console.WriteLine(names[i]);
}
```

Resultaat:

```
Alice
Bob
Charlie
```

Gebruik bij `List<T>` altijd `.Count` — niet `.Length` zoals bij arrays!

4.7. Elementen verwijderen

Je kan elementen verwijderen via hun waarde of index:

```
List<string> cities = new List<string> { "Paris", "London", "Rome", "London" };
cities.Remove("London");        // Verwijdert eerste voorkomen
cities.RemoveAt(1);             // Verwijdert op index 1

foreach (string city in cities)
{
    Console.WriteLine(city);
}
```

Resultaat:

```
Paris
London
```

Wanneer je een element verwijdert uit een lijst dan wijzigt de lengte van de lijst:

```
List<string> cities = new List<string> { "Paris", "London",
"Rome", "London" };
cities.Remove("London");        // Verwijdert eerste voorkomen
cities.RemoveAt(3);             // ⚠ Dit veroorzaakt een runtime
error!
```

4.7.1. De lijst leegmaken

Met `Clear()` kan je in één keer **alle** elementen uit een lijst verwijderen. De lijst zelf blijft bestaan, maar is leeg.

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
names.Clear();

Console.WriteLine($"Aantal elementen: {names.Count}");
```

Resultaat:

```
Aantal elementen: 0
```

Na `Clear()` is de lijst leeg, maar je kan ze nog verder gebruiken — je hoeft geen nieuwe lijst te maken.

4.8. Een element invoegen

Met `Insert()` voeg je een element toe op een specifieke positie:

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
names.Insert(1, "Diana");

foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Resultaat:

```
Alice
Diana
Bob
Charlie
```

4.9. Positie van een element zoeken

Met `IndexOf()` zoek je de eerste index van een element. Als het niet gevonden wordt, krijg je `-1`:

```
List<string> students = new List<string> { "Tom", "Elise", "Tom" };
int index = students.IndexOf("Tom");
Console.WriteLine(index);
```

Resultaat:

```
0
```

```
int missing = students.IndexOf("Anna");
Console.WriteLine(missing);
```

Resultaat:

```
-1
```

4.10. De lijst omzetten naar een array

```
List<string> fruits = new List<string> { "Apple", "Banana", "Pear" };
string[] fruitArray = fruits.ToArray();
```

4.11. Een stukje uit de lijst halen

```
List<int> numbers = new List<int> { 10, 20, 30, 40, 50 };
List<int> subset = numbers.GetRange(1, 3);

foreach (int n in subset)
{
    Console.WriteLine(n);
}
```

Resultaat:

```
20
30
40
```

4.12. Sorteren

```
List<string> names = new List<string> { "Bob", "Alice", "Charlie" };
names.Sort();

foreach (string name in names)
{
```

```
    Console.WriteLine(name);
}
```

Resultaat:

```
Alice
Bob
Charlie
```

Wil je sorteren van groot naar klein? Gebruik dan `Reverse()` na `Sort()`.

4.13. Voorbeeld met objecten

```
class Student
{
    public string FirstName;
    public int Score;
}

List<Student> students = new List<Student>
{
    new Student { FirstName = "Alice", Score = 15 },
    new Student { FirstName = "Bob", Score = 12 },
    new Student { FirstName = "Charlie", Score = 18 }
};

students.Sort((a, b) => a.Score.CompareTo(b.Score));

foreach (Student s in students)
{
    Console.WriteLine($"{s.FirstName}: {s.Score}/20");
}
```

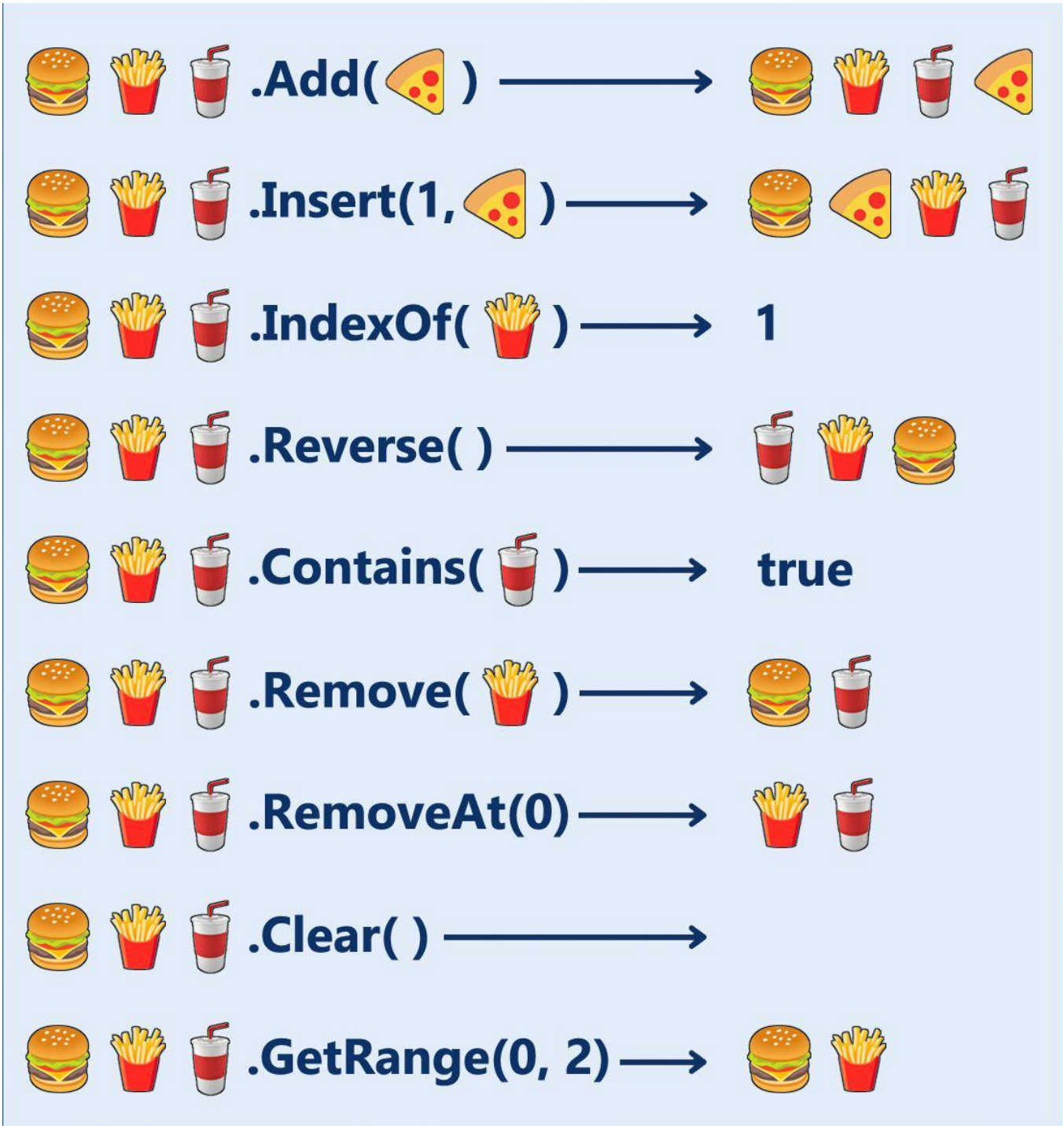
Resultaat:

```
Bob: 12/20
Alice: 15/20
Charlie: 18/20
```

4.14. Samenvatting

Methode of property	Doel
`Add()` / `AddRange()`	Element(en) toevoegen aan lijst
`Remove()` / `RemoveAt()`	Verwijderen via waarde of index
`Insert()`	Nieuw element invoegen

Methode of property	Doel
`Clear()`	Volledige lijst leegmaken
`Count`	Aantal elementen opvragen
`IndexOf()`	Zoekt de positie van een element
`[index]`	Opvragen of aanpassen van een element
`ToArray()`	Zet lijst om naar array
`GetRange()`	Haalt een deel uit de lijst
`Sort()` / `Reverse()`	Sorteert de lijst, eventueel in omgekeerde volgorde



4.15. Veelgemaakte fout

Stel dat je alle namen die de letter "0" bevatten moet schrappen uit een lijst. Vaak wordt hier een `foreach`-lus voor gebruikt in combinatie met een `if`-structuur:

```
List<string> names = new List<string>
{
    "Anna",
    "Bart",
    "Olga",
    "Dieter",
    "Eva",
    "Frank",
```

```
    "Gina",
    "Hugo",
    "Iris",
    "Jonas"
};

foreach(string name in names)
{
    if (name.Contains("o", StringComparison.OrdinalIgnoreCase))
    {
        names.Remove(name);
    }
}
```

Dit resulteert helaas in een foutmelding omdat je de `List` wijzigt (door de `Remove()`) terwijl je door alle elementen aan het lussen bent.

Kan jij een mogelijke oplossing bedenken?

Example 1. Oplossing

Je kan de `foreach`-lus vervangen door een `for`-lus. Omdat de waarde van de `Count`-eigenschap elke keer wijzigt wanneer je een element uit de lijst verwijdert zal deze code geen fout veroorzaken:

```
for (int i = 0; i < names.Count; i++)
{
    names.RemoveAt(i);
}
```

In de vervolgcursus **C# Advanced** maak je kennis met LINQ. Deze bibliotheek bevat de `RemoveAll` -methode waarmee je met een lambda-expressie hetzelfde resultaat kan bereiken:

```
names.RemoveAll(n => n.Contains("0", StringComparison.OrdinalIgnoreCase));
```

5. Dictionary<TKey, TValue>

5.1. Wat is een Dictionary?

Een `Dictionary` lijkt een beetje op een telefoonboek:

- De **key** is zoals een naam (bijv. "Tom").
- De **value** is de bijhorende info (bijv. "0499/12.34.56").

Je kan het telefoonnummer (value) dus opzoeken op basis van de naam (key)! Anders dan een array of list, gebruik je hier dus geen index, maar een unieke sleutel.

5.2. Een Dictionary aanmaken

Een dictionary bestaat uit **sleutel-waarde-paren**. In C# gebruik je het type `Dictionary<TKey, TValue>`, waarbij:

- `TKey` het type van de **sleutel** is
- `TValue` het type van de **waarde** is

Bijvoorbeeld:

```
Dictionary<string, int> scores = new Dictionary<string, int>();
```

In dit voorbeeld:

- de **key** is een `string` (bijv. een naam)
- de **value** is een `int` (bijv. het aantal punten)

Andere voorbeelden:

```
// Een dictionary met studentnummer als int en naam als string
Dictionary<int, string> students = new Dictionary<int, string>();

// Een dictionary met string als sleutel en bool als waarde
Dictionary<string, bool> status = new Dictionary<string, bool>();
```

5.2.1. Initialisatie

Je kan een dictionary ook meteen bij het aanmaken invullen met sleutel-waarde-paren. Dat doe je met een blok tussen accolades `{ … }`, net zoals bij een lijst:

```
Dictionary<string, bool> toggles = new Dictionary<string, bool>
{
    { "DarkMode", true },
    { "Notifications", false },
    { "AutoSave", true }
```

```
};
```

Je kan dit ook gebruiken bij een dictionary met getallen:

```
Dictionary<string, int> scores = new Dictionary<string, int>
{
    { "Alice", 14 },
    { "Bob", 16 },
    { "Charlie", 18 }
};
```

Bij deze vorm moet je elk item schrijven als `{ sleutel, waarde }` tussen accolades.

5.2.2. Oefening: Juist of fout?

Bepaal voor elke lijn of deze correct is in C#. Noteer voor jezelf of ze **juist** of **fout** is.

```
1. Dictionary<string, int> scores = new Dictionary<string, int>();
2. Dictionary<int, string> map = new Dictionary<string, int>();
3. Dictionary<string, bool> flags = new Dictionary<string, bool> { { "Visible",
true } };
4. Dictionary<string> items = new Dictionary<string>();
5. var dict = new Dictionary<string, string>();
6. Dictionary<string, int> d = { { "A", 1 }, { "B", 2 } };
7. Dictionary<string, double> values = new Dictionary<string, double>();
8. Dictionary<string, string> products = { { 1, "Laptop" }, { 2, "Desktop" } };
```

Oplossing

Lijn	/	Uitleg
1		Geldige dictionary met string als key en int als value.
2		Linkerzijde en rechterzijde gebruiken verschillende types.
3		Correcte initialisatie met standaardwaarde.
4		Dictionary vereist altijd twee types: `<TKey, TValue>`.
5		`var` werkt hier omdat het type volledig afleidbaar is. (we raden het gebruik van var echter niet aan)
6		Object-initializer mist `new Dictionary<…>(…)` constructie.
7		Geldige declaratie van een lege dictionary.

Lijn	?/?	Uitleg
8	?	De key `"1"` is een `int`, terwijl de dictionary `string` verwacht als key.

5.3. Elementen toevoegen met `Add`

Met de `Add(key, value)`-methode voeg je één item toe aan een dictionary. De key moet **uniek** zijn — als dezelfde sleutel al bestaat, krijg je een foutmelding.

```
Dictionary<string, int> scores = new Dictionary<string, int>();

scores.Add("Tom", 12);
scores.Add("Elise", 16);
scores.Add("Ozgun", 17);
```

Als je `Add()` probeert te gebruiken met een sleutel die al bestaat, krijg je een `ArgumentException`.

Wil je een bestaande waarde vervangen? Dan gebruik je niet `Add()` maar gewoon toewijzing via de index:

```
scores["Tom"] = 18;
```

5.4. Een waarde opvragen via key

```
int result = scores["Tom"];
Console.WriteLine(result);
```

Resultaat:

```
12
```

Als de sleutel niet bestaat, krijg je een foutmelding (`KeyNotFoundException`).

5.5. Bestaat de sleutel?

Gebruik `ContainsKey()` om te controleren of een key bestaat:

```
if (scores.ContainsKey("Tom"))
{
    Console.WriteLine("Tom is ingeschreven!");
}
```

Resultaat:

```
Tom is ingeschreven!
```

5.6. Waarde veilig opvragen met `TryGetValue()`

```
int points;
bool found = scores.TryGetValue("Anna", out points);

if (found)
    Console.WriteLine($"Anna: {points}");
else
    Console.WriteLine("Anna werd niet gevonden.");
```

Resultaat:

```
Anna werd niet gevonden.
```

Dit werkt zoals `TryParse` en is een veilig alternatief voor `dict["key"]`.

5.7. Element overschrijven

```
scores["Tom"] = 18;
Console.WriteLine(scores["Tom"]);
```

Resultaat:

```
18
```

5.8. Element verwijderen

```
scores.Remove("Elise");

foreach (var kv in scores)
{
    Console.WriteLine($"{kv.Key}: {kv.Value}");
}
```

Resultaat:

```
Tom: 18
Ozgun: 17
```

5.9. Wat is een KeyValuePair?

Wanneer je een dictionary doorloopt met `foreach`, krijg je telkens een **sleutel-waarde-paar**. Dat is een

object van het type `KeyValuePair<TKey, TValue>`.

Een `KeyValuePair` bevat dus twee stukjes informatie:

- `.Key` → de sleutel
- `.Value` → de bijhorende waarde

Bijvoorbeeld:

```
Dictionary<string, int> scores = new Dictionary<string, int>
{
    { "Alice", 15 },
    { "Bob", 18 }
};

foreach (KeyValuePair<string, int> entry in scores)
{
    Console.WriteLine($"{entry.Key} heeft {entry.Value} punten.");
}
```

Resultaat:

```
Alice heeft 15 punten.
Bob heeft 18 punten.
```

In de praktijk schrijven we dit vaak korter als:

```
foreach (var kv in scores)
{
    Console.WriteLine($"{kv.Key}: {kv.Value}");
}
```

Maar `kv` **is** dan nog steeds een `KeyValuePair<string, int>` — je gebruikt gewoon `var` om het eenvoudiger te houden.

Je kan `KeyValuePair` zien als een mini-object dat 2 eigenschappen bevat: één voor de key en één voor de value.

5.10. Alle keys of values ophalen

```
foreach (string name in scores.Keys)
{
    Console.WriteLine(name);
}
```

Resultaat:

```
Tom
Ozgun
```

```
foreach (int score in scores.Values)
{
    Console.WriteLine(score);
}
```

Resultaat:

```
18
17
```

5.11. Over de dictionary lopen

```
foreach (var entry in scores)
{
    Console.WriteLine($"{entry.Key}: {entry.Value}/20");
}
```

Resultaat:

```
Tom: 18/20
Ozgun: 17/20
```

5.12. Dictionary met objecten

```
class Student
{
    public string FirstName;
    public int Score;
}

Dictionary<string, Student> studentMap = new Dictionary<string, Student>();

studentMap.Add("s001", new Student { FirstName = "Alice", Score = 16 });
studentMap.Add("s002", new Student { FirstName = "Bob", Score = 14 });

foreach (var kv in studentMap)
{
    Console.WriteLine($"{kv.Key}: {kv.Value.FirstName} - {kv.Value.Score}/20");
}
```

Resultaat:

```
s001: Alice - 16/20
```

```
s002: Bob - 14/20
```

5.13. Samenvatting

Methode of property	Doel
`Add(key, value)`	Nieuw item toevoegen
`[key]`	Een waarde opvragen of overschrijven
`ContainsKey(key)`	Controleren of een sleutel bestaat
`TryGetValue(key, out value)`	Veilig een waarde opvragen
`Remove(key)`	Een item verwijderen
`Keys` / `Values`	Alleen de sleutels of waarden doorlopen
`foreach (var kv in dict)`	Volledige dictionary overlopen