

1.0.0

## **H13: Datastrukturen**

# Foreach en var

## Foreach loops

Wanneer je geen indexering nodig hebt, maar toch snel over alle elementen in een array wenst te gaan, dan is het **foreach** statement een zeer nuttig is. Een foreach loop zal ieder element in de array een voor een in een tijdelijke variabele plaatsen (de **iteration variable**). Volgende code toont de werking waarbij we een array van `string`s hebben en alle elementen er in op het scherm willen tonen:

```
string[] boodschappen= {"ontbijtgranen", "koekjes", "fruit"};
foreach (string boodschap in boodschappen)
{
    Console.WriteLine($"Niet vergeten: {boodschap}");
}
```

De eerste keer dat we in de loop gaan zal het element `boodschappen[0]` aan `boodschap` toegewezen worden voor gebruik in de loop-body, vervolgens wordt `boodschappen[1]` toegewezen, enz.

Het voordeel is dat je dus geen teller/index nodig hebt en dat foreach zelf de lengte van de array zal bepalen.



### Opgelet bij het gebruik van foreach loops

- De foreach iteration variable is *read-only*: je kan dus geen waarden in de array aanpassen, enkel uitlezen.
- De foreach gebruik je enkel als je alle elementen van een array wenst te benaderen. In alle andere gevallen zal je een ander soort loop (for, while, etc.) moeten gebruiken.

## var keyword

C# heeft een **var** keyword. Je mag dit keyword gebruiken ter vervanging van het type (bv int) op voorwaarde dat de compiler kan achterhalen wat het type moet zijn.

```
var getal = 5; // var zal int zijn
var myArray = new double[20]; // var zal double[] zijn
var tekst = "Hi there handsome"; // var zal string zijn
```



**Opgelet:** het `var` keyword is in deze cursus nooit **nodig**. Het vergemakkelijkt het schrijfwerk, want het wordt door de compiler vertaald in een specifiek type. Er zijn scenario's waarin het wel nodig is, maar die zijn meer geavanceerd ("anonieme types").

Het betekent **niet** hetzelfde als de `var` van JavaScript. In JavaScript hoef je namelijk geen type vast te leggen voor variabelen en kan je dit doen:

```
var something = "hello";  
something = 3;
```

In C# levert dit een compilatiefout. Met de eerste regel zeg je dat de compiler uit de rechterzijde mag afleiden dat `var` hier vervangen kan worden door `string`. Je kan geen waarde `3` in een variabele van type `string` plaatsen, dus dit levert een compilatiefout.

## var en foreach

Wanneer je de Visual Studio [code snippet](#) voor foreach gebruikt `foreach [tab][tab]` dan zal deze code ook een `var` gebruiken voor de iteration variabele. De compiler kan aan de te gebruiken array zien wat het type van een individueel element in de array moet zijn. De foreach van zonet kan dus herschreven worden naar:

```
foreach (var boodschap in boodschappen)  
{  
    Console.WriteLine($"Niet vergeten: {boodschap}");  
}
```

# List



In onderstaande kennisclip wordt er een andere syntax gebruikt om objecten aan te maken dan wat wij gewoon zijn (namelijk constructors met parameters). Dat maakt geen verschil voor de werking van de datastructuur. Je zou net zo goed een constructor kunnen definiëren die hetzelfde doet.



Kennisclip voor deze inhoud

Een `List<>` collectie is de meest standaard collectie die je kan beschouwen als een flexibelere variant op een een doodnormale array.



De Generieke `List<>` klasse bevindt zich in de `System.Collections.Generic` namespace. Je dient deze namespace dus als `using` bovenaan toe te voegen wil je deze klasse kunnen gebruiken.

## List aanmaken

De klasse `List<T>` is een zogenaamde generieke klasse. Tussen de `< >` tekens plaatsen we het type dat de lijst zal moeten gaan bevatten. Vaak wordt dit genoteerd als `T` voor "type". Bijvoorbeeld:

- `List<int> alleGetallen= new List<int>();`
- `List<bool> binaryList = new List<bool>();`
- `List<Pokemon> pokedex = new List<Pokemon>();`
- `List<string[]> listOfStringarrays = new List<string[]>();`

Zoals je ziet hoeven we bij het aanmaken van een `List` geen begingrootte mee te geven, wat we wel bij arrays moeten doen. Dit is een van de voordelen van `List`: ze groeien mee. Als we toch een begingrootte meegeven (zoals in de kennisclip even getoond wordt) is dat enkel om de performantie van de code wat te verhogen in bepaalde scenario's. Wij gaan dit nooit doen.

## Elementen toevoegen

Via de `Add()` methode kan je elementen toevoegen aan de lijst. Je dient als parameter aan de methode mee te geven wat je aan de lijst wenst toe te voegen. **Deze parameter moet uiteraard van het type zijn dat de `List` verwacht.**

In volgende voorbeeld maken we een List aan die objecten van het type string mag bevatten en vervolgens plaatsen we er twee elementen in.

```
List<String> myStringList = new List<String>();  
myStringList.Add("This is the first item in my list!");  
myStringList.Add("And another one!");
```

## Elementen indexeren

Het leuke van een List is dat je deze ook kan gebruiken als een gewone array, waarbij je met de indexer elementen kan aanspreken. Stel bijvoorbeeld dat we een lijst hebben met minstens 4 strings in. Volgende code toont hoe we de string op positie 3 kunnen uitlezen en hoe we die op positie 2 overschrijven:

```
Console.WriteLine(myStringList[3]);  
myStringList[2] = "andere zin";
```

Ook de klassieke werking met `for` blijft gelden. De enige aanpassing is dat `List<T>` niet met `Length` werkt maar met `Count`.

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

## Wat kan een List nog?

Interessante methoden en properties voorts zijn:

- `Clear()` :methode die de volledige lijst leegmaakt
- `Insert()` : methode om element op specifieke plaats in lijst toe te voegen, bijvoorbeeld:  

```
myStringList.Insert(1,"A fourth sentence");
```

voegt de string toe op de tweede plek en schuift de rest naar achter
- `Contains()` : geef als parameter een specifiek object mee (van het type `T` dat de `List<T>` bevat) om te weten te komen of dat specifieke object in de `List<>` terug te vinden is. Indien ja dan zal `true` worden teruggegeven.
- `IndexOf()` : geeft de index terug van het element item in de rij. Indien deze niet in de lijst aanwezig is dan wordt `-1` teruggegeven.
- `RemoveAt()` : verwijder een element op de index die je als parameter meegeeft.
- `Remove()` : verwijder het gegeven element



Contains, Remove en IndexOf zullen zich met jouw eigen klassen niet noodzakelijk gedragen zoals je verwacht. De verklaring hierachter komt later aan bod, wanneer we [Equals](#) en [GetHashCode](#) bekijken. Ze zullen wel werken zoals verwacht voor voorgedefinieerde types, inclusief DateTime.

## Foreach loops

Je kan met een eenvoudige `for` of while-loop over een lijst itereren, maar het gebruik van een foreach-loop is toch handiger.

Dit is dan ook de meestgebruikte operatie om eenvoudig en snel een bepaald stuk code toe te passen op ieder element van de lijst:

```
List<int> integerList=new List<int>();
integerList.Add(2);
integerList.Add(3);
integerList.Add(7);

foreach(int prime in integerList)
{
    Console.WriteLine(prime);
}
```

# Dictionary

✓ Kennisclip voor deze inhoud

Naast de generieke `List` collectie, zijn er nog enkele andere nuttige generieke 'collectie-klassen' die je geregeld in je projecten kan gebruiken.

In een **dictionary** wordt ieder element voorgesteld door een sleutel (**key**) en de waarde (**value**) van het element. Het idee is dat je de sleutel kan gebruiken om de waarde snel op te zoeken. De sleutel moet dan ook uniek zijn. **Dictionaries stellen geen reeks met een volgorde voor, maar geven je de mogelijkheid data met elkaar in verband te brengen.**

Enkele voorbeeldjes die het idee achter Dictionary kunnen verduidelijken:

- een papieren telefoonboek is als een Dictionary met gecombineerde namen en adressen als keys en telefoonnummers als values
- een echt woordenboek is als een Dictionary met woorden als keys en omschrijvingen als values
- een array is als een Dictionary met getallen als keys en waarden van het type van de array als values

Bij de declaratie van de `Dictionary<K, V>` dien je op te geven wat het datatype van de key zal zijn, alsook het type van de waarde (value). Met andere woorden, `K` en `V` komen niet letterlijk voor, maar je vervangt ze door types die je al kent.

Voor bovenstaande voorbeelden:

- een echt woordenboek stel je best voor met `string` als type van de key en `string` als type van de value, want een woord stel je voor als string en een omschrijving ook
- een array van `double` kan je nabootsen door `uint` te gebruiken als type van de key en `double` als type van de value
- het telefoonboek moeten we wat vereenvoudigen, maar als we op basis van naam meteen een telefoonnummer konden opzoeken (zonder adresgegevens,...), dan zou `string` (de naam) het type van de key zijn en `string` (telefoonnummer) het type van de value. Het telefoonnummer is geen getal omwille van zaken die je niet met een getaltype kan voorstellen zoals `+32` of `0473`.

## Gebruik Dictionary

In het volgende voorbeeld maken we een `Dictionary` van klanten aan. Iedere klant heeft een unieke ID (de key, die we als `int` gebruiken) alsook een naam (die niet noodzakelijk uniek is en de waarde voorstelt):



```
Dictionary<int, string> customers = new Dictionary<int, string>();
customers.Add(123, "Tim Dams");
customers.Add(6463, "James Bond");

customers.Add(666, "The beast");
customers.Add(700, "James Bond");
```

Bij de declaratie van `customers` plaatsen we dus tussen de `<` `>` twee datatypes: het eerste duidt het datatype van de key aan, het tweede dat van de values.

Merk op dat je niet verplicht bent om een `int` als type van de key (of value) te gebruiken, dit mag eender wat zijn, zelfs een klasse.

```
Dictionary<int, Pokemon> pokedex;
Dictionary<Student, PuntenLijst> puntenTabel;
```

⚠ Bij dit laatste horen wel enkele nuances. Deze worden pas behandeld [in een later hoofdstuk](#). Voorlopig zullen we alleen voorgedefinieerde types opnemen in dictionaries.

We kunnen nu met behulp van bijvoorbeeld een `foreach`-loop alle elementen tonen. Hier kunnen we de key met de `.Key`-property uitlezen en het achterliggende object of waarde met `.Value`. `Value` en `Key` hebben daarbij ieder het type dat we hebben gedefinieerd toen we het `Dictionary`-object aanmaakten, in het volgende geval is de `Key` dus van het type `int` en `Value` van het type `string`:

```
foreach (var item in customers){
    Console.WriteLine($"{item.Key}\t:{item.Value}");
}
```

We kunnen echter ook een specifiek element opvragen aan de hand van de key. Stel dat we de waarde van de klant met key 123 willen tonen:

```
Console.WriteLine(customers[123]);
```

De key werkt dus net als de index bij gewone arrays, **alleen heeft de key nu geen relatie meer met de positie van het element in de collectie**.

Je kan de syntax met rechte haakjes ook gebruiken om een element toe te voegen. In tegenstelling tot `Add`, geeft deze syntax geen fout als de key al bestaat, maar vervangt hij het bestaande verband:

```
// dit gaat, terwijl het met Add verboden is
customers[123] = "klant A";
customers[123] = "klant A, opnieuw";
```

Als je wil weten of een bepaalde key voorkomt in een Dictionary, gebruik je de instantiemethode `ContainsKey`.

# Immutable datastructuren

De standaard datastructuren van C# zijn [reference types](#). Dit betekent dat iedereen die zo'n datastructuur te pakken krijgt (bijvoorbeeld omdat je hem als argument meegeeft aan een methode), de inhoud van deze datastructuur ook kan wijzigen. Dit kan met opzet of gewoonweg per vergissing gebeuren.

**i** Hoezo, "met opzet"? Denk eraan dat je typisch niet de enige programmeur bent die met bepaalde code in contact komt.

Bijvoorbeeld:

```
// je gebruikt deze methode in de veronderstelling dat ze je data alleen maar print
// maar, zonder dat je het daarom meteen merkt, wist ze ook data
public static void PrintData(List<string> data) {
    for (int i = 0; i < data.Count; i++) {
        Console.WriteLine(data[i]);
        data[i] = null;
    }
}
```

In het algemeen geldt: als iemand bepaalde mogelijkheden niet echt **nodig** heeft, geef ze dan niet. Dit is opnieuw **encapsulatie**.

Om te verhinderen dat een datastructuur wordt aangepast, kan je er een immutable versie van maken. Dit is een versie van die datastructuur waarvan de inhoud achteraf niet gewijzigd kan worden. Er bestaan immutable versies van de standaard datastructuren en ze heten gewoonweg `ImmutableList<T>` en `ImmutableDictionary<K,V>`.

Om deze versies te gebruiken, moet je de `System.Collections.Immutable` namespace gebruiken. Wanneer je hier een `using` directief voor hebt staan, kan je methodes zoals `ToImmutableList<T>` oproepen op een lijst om er een immutable versie van te produceren. Deze immutable versie kan je dan veilig delen met code waarvan je niet wenst dat ze de inhoud van je lijst aanpast.

Een tweede manier om een immutable datastructuur te maken, is met een **builder**. Dit is een object waar je via `Add` data aan toevoegt en dat je achteraf vraagt een immutable list te produceren. Je kan er een aanmaken met de statische methode `CreateBuilder` van de immutable datastructuur die je wil gebruiken. Bijvoorbeeld:

```

public static void PrintData(ImmutableList<string> data) {
    foreach(var datum in data) {
        Console.WriteLine(datum);
    }
}

public static void DemonstreerImmutableListBuilder() {
    var builder = ImmutableList.CreateBuilder<string>();
    bool doorgaan;
    do {
        Console.WriteLine("Geef een element om toe te voegen");
        builder.Add(Console.ReadLine());
        Console.WriteLine("Doorgaan met elementen toevoegen?");
        doorgaan = Console.ReadLine().ToLower() == "ja";
    } while (doorgaan);
    PrintData(builder.ToImmutableList<string>());
}

```

### het verschil met read-only properties

Beginnende programmeurs denken soms dat ze hetzelfde effect kunnen verkrijgen door een property voor een datastructuur "read only" te maken. Dit doen ze dan door alleen een getter te voorzien en geen setter of, als ze buiten deze cursus gaan zoeken, met het sleutelwoordje `readonly`.

**Dit maakt je datastructuur niet immutable!** Het zorgt er **wel** voor dat je het object op de heap waarin je data staat niet kan vervangen. Het zorgt er **niet** voor dat je de inhoud van dat object niet kan vervangen.

Bijvoorbeeld, als we personen voorzien van een array met lievelingsgerechten:

```

class Persoon {

    private List<string> lievelingsgerechten;
    public List<string> Lievelingsgerechten {
        get {
            return this.lievelingsgerechten;
        }
    }

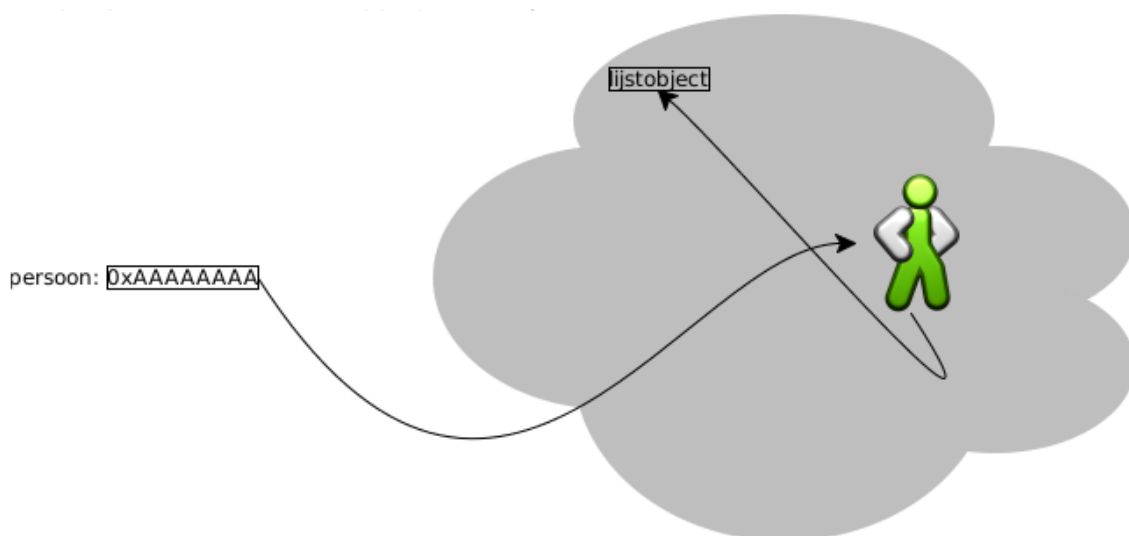
    public Persoon(string gerecht1, string gerecht2, string gerecht3) {
        this.lievelingsgerechten = new List<string>();
        this.lievelingsgerechten.Add(gerecht1);
        this.lievelingsgerechten.Add(gerecht2);
        this.lievelingsgerechten.Add(gerecht3);
    }
}

class Program {
    public static void Main() {
        var persoon = new Persoon("spaghetti", "koekjes", "ijs");
        // dit gaat WEL:

        persoon.Lievelingsgerechten[1] = "lasagne";
        persoon.Lievelingsgerechten.Add("frietten");
        foreach(var gerecht in persoon.Lievelingsgerechten) {
            Console.WriteLine(gerecht);
        }
        // dit gaat NIET:
        // persoon.Lievelingsgerechten = new List<string>();
    }
}

```

Onderstaande figuur toont een vereenvoudigde weergave van wat er aan de hand is:



Het is niet mogelijk de pijl van de persoon naar het lijstobject te vervangen. Het is wel mogelijk data in het lijstobject te veranderen.

# Verdere datastructuren

## HashSet en ImmutableHashSet

In sommige situaties wil je dat een element geen twee keer in een datastructuur terecht kan komen. Je wil bijvoorbeeld dat de lijst met cursussen die deel uitmaakt van een studieprogramma geen tweemaal dezelfde cursus kan bevatten.

In dit geval gebruik je geen `List<T>`, maar een `HashSet<T>`. Elementen toevoegen doe je met de methode `Add` en elementen verwijderen doe je met `Remove`. Ook hier beperken we ons voorlopig tot voorgedefinieerde soorten objecten. Wanneer we `System.Object` hebben bestudeerd, kunnen we ook `HashSet` s van onze eigen types maken.

De immutable variant is `ImmutableHashSet`.

## Queue

Een `Queue` is een collectie van elementen die in een welbepaalde volgorde behandeld moeten worden: van voor naar achter. Vergelijk met een wachtrij bij de bakker: de klant die eerst in de rij staat wordt eerst geholpen, dan steeds de volgende klant tot we aankomen bij de klant die het laatst in de rij is aangesloten. We noemen dit soort van collectie ook wel een *First In, First Out* oftewel *FIFO*-collectie: het item dat eerst in de rij is gezet, is ook het eerste dat behandeld wordt.

Een `Queue` is dus een speciaal soort lijst, waarbij het toevoegen en verwijderen van elementen op de lijst niet op gelijk welke plaats mag gebeuren. Een queue biedt daarom geen `Add()` of `RemoveAt()` methode aan. In plaats daarvan gebruik je:

- `Enqueue(T item)` om een item aan de rij toe te voegen
- `Dequeue()` om een item uit de rij te halen. Deze methode geeft als returnwaarde het weggehaalde item terug, zodat je er iets mee kan doen.
- `Peek()` geeft je het eerstvolgende item terug, maar verwijdert het nog niet uit de rij.

## Voorbeeld

```

public static void Bakker()
{
    Queue<string> klanten = new Queue<string>();

    klanten.Enqueue("Roos");
    klanten.Enqueue("Piet");
    klanten.Enqueue("Ellen");
    klanten.Enqueue("Frank");
    klanten.Enqueue("Oswald");

    Console.Write("De huidige wachtrij is: ");
    foreach (string klant in klanten)
    {
        Console.Write($"{klant} ");
    }
    Console.WriteLine('\n');

    Console.WriteLine($"We bedienen nu klant {klanten.Dequeue()}");
    klanten.Enqueue("Redouan"); //Nieuwe klant komt in de rij staan
    Console.WriteLine($"De volgende klant is {klanten.Peek()}");

    Console.Write("De huidige wachtrij is: ");
    foreach (string klant in klanten)
    {
        Console.Write($"{klant} ");
    }
    Console.WriteLine('\n');


    Console.WriteLine($"We bedienen nu klant {klanten.Dequeue()}");

    Console.Write("De huidige wachtrij is: ");
    foreach (string klant in klanten)
    {
        Console.Write($"{klant} ");
    }
    Console.WriteLine('\n');
}

```

- ❗ Op lijn 18 wordt de volgende klant uit de rij gehaald. Deze klant gebruiken we nog snel om zijn naam te tonen aan de gebruiker, maar na lijn 29 zal deze klant verdwijnen. Wil je deze klant in meer dan één statement gebruiken, zal je hem dus moeten opslaan in een lokale variabele:

```
//...
string volgendeKlant = klanten.Dequeue(); //sla de klant op in een lokale variabele
Console.WriteLine($"We bedienen nu klant {volgendeKlant}");
this.StuurFactuur(volgendeKlant); //klant "Roos" wordt nu ook in deze methodeoproep gebruikt
//...
```

 Op lijn 20 wordt er eerst 'gespiekt' wie de volgende klant is: Piet. Met `Peek()` wordt hij echter nog niet uit de rij gehaald, zoals je in onderstaande output kan zien.

De huidige wachtrij is: Roos Piet Ellen Frank Oswald

We bedienen nu klant Roos  
De volgende klant is Piet

De huidige wachtrij is: Piet Ellen Frank Oswald Redouan

We bedienen nu klant Piet

De huidige wachtrij is: Ellen Frank Oswald Redouan

## ImmutableQueue

De immutablevariant van Queue is ImmutableQueue.

## Stack

Het omgekeerde van een Queue is een Stack. Dit is een lijst van items waarbij je steeds het laatst toegevoegde item eerst wilt behandelen. Vergelijk dit met een stapel borden aan de afwas: het eerstvolgende bord dat je afwast, is het bovenste bord op de stapel, dus het laatst toegevoegde. Of met een rij wagens in een lange, smalle garage met maar één toegangspoort: de eerste wagen die kan buitenrijden, is degene die laatst is binnengereden.

Dit noemen we een *LIFO*-collectie, oftewel *Last In, First Out*. Waar Queue `Enqueue(T item)` en `Dequeue()` gebruikte om items toe te voegen en uit de rij te halen, gebruikt Stack

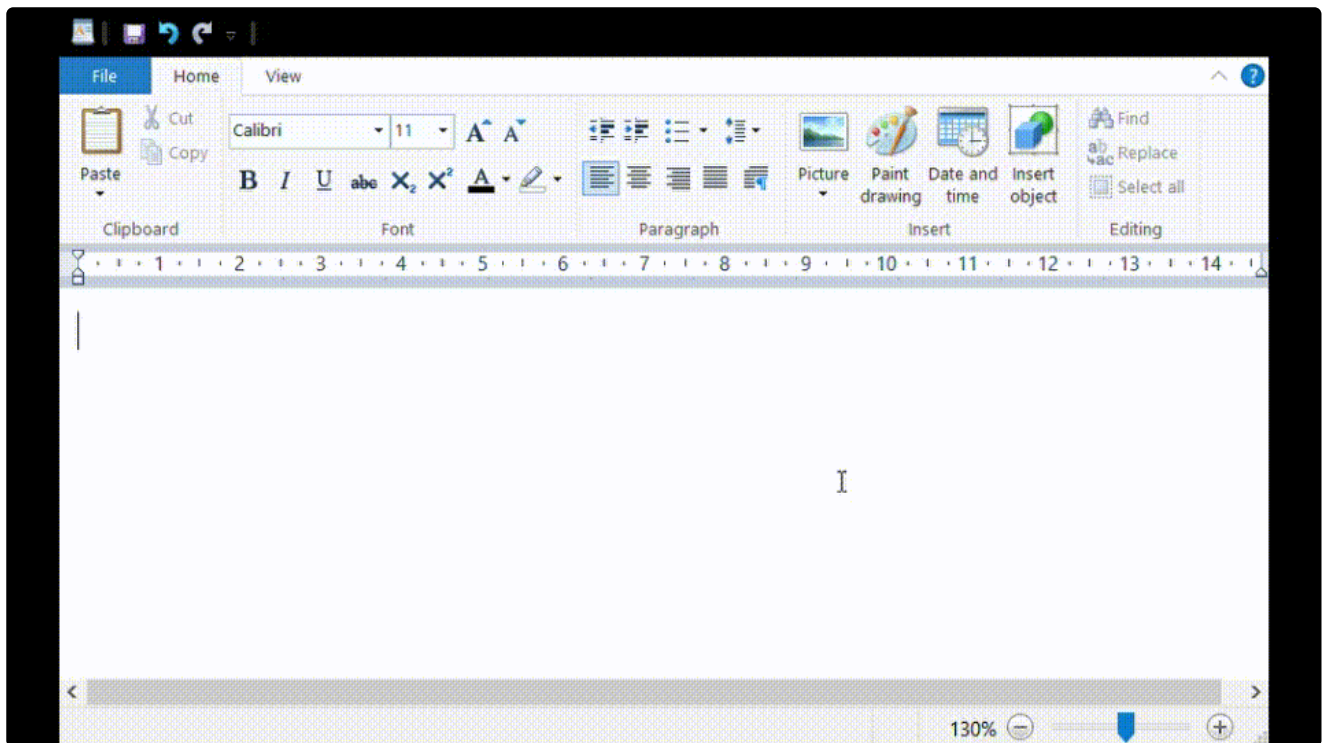
- `Push(T item)` om een item op de stapel te leggen.
- `Pop()` om een item van de stapel te nemen.
- `Peek()` om het bovenste item op de stapel te bekijken, zonder het er af te nemen.

## Voorbeeld



Dit voorbeeld demonstreert de werking van de 'Maak ongedaan' functionaliteit die je hebt in de meeste programma's op je computer. Als je op 'Maak ongedaan' (Engels: undo, commando: Ctrl+Z) klikt, wordt enkel dat wat je als laatste gedaan hebt, teruggedraaid.

Volgend filmpje demonstreert de acties die de gebruiker uitvoert in een tekstbewerkingsprogramma:



Voorbeeld van het gebruik van een tekstverwerker met undo-functionaliteit

De gebruiker neemt volgende stappen, vertrekkende vanaf een wit blad:

1. Voeg paragraaf toe
2. Zet tekst in vet
3. Haal stuk tekst weg
4. *Maak laatste actie ongedaan*
5. Maak tekst groter.
6. *Maak laatste actie ongedaan*
7. Maak tekst kleiner.
8. Voeg tekst toe.

De code om deze acties bij te houden in een actiehistoriek zou kunnen zijn:

```

public static void UndoDemo()
{
    Stack<string> acties = new Stack<string>();

    acties.Push("Voeg paragraaf toe.");
    acties.Push("Zet tekst in vet.");
    acties.Push("Haal stuk tekst weg.");

    Console.WriteLine("De actiehistoriek is: ");
    foreach (string actie in acties)
    {
        Console.WriteLine($"{t{actie} ");
    }
    Console.WriteLine('\n');

    Console.WriteLine($"Maak ongedaan: {acties.Pop()}\n");
    acties.Push("Maak tekst groter.");
    Console.WriteLine($"Maak ongedaan: {acties.Pop()}\n");

    acties.Push("Maak tekst kleiner.");

    Console.WriteLine($"De meest recente actie was: {acties.Peek()}\n");

    acties.Push("Voeg tekst toe.");

    Console.WriteLine("De actiehistoriek is: ");
    foreach (string actie in acties)
    {
        Console.WriteLine($"{t{actie} ");
    }
    Console.WriteLine('\n');
}

```

Dit geeft volgende output:

De actiehistoriek is:

Haal stuk tekst weg.

Zet tekst in vet.

Voeg paragraaf toe.

Maak ongedaan: Haal stuk tekst weg.

Maak ongedaan: Maak tekst groter.

De meest recente actie was: Maak tekst kleiner.

De actiehistoriek is:

Voeg tekst toe.

Maak tekst kleiner.

Zet tekst in vet.

Voeg paragraaf toe.

## ImmutableStack

De immutablevariant van Stack is ImmutableStack.

# Labo

## H13-boodschappenlijstje

### Functionele analyse

Schrijf een programma om een boodschappenlijstje samen te stellen en af te werken aan de hand van `List<string>`. Maak deze versie deel van de klasse `Datastructuren`.

### Technische analyse

- Vraag eerst om items toe te voegen, totdat er een lege regel wordt ingegeven. Toon telkens om het hoeveelste item het gaat, zoals in de voorbeeldinteractie.
- Sorteer vervolgens de lijst.
- Vraag dan, zo lang er nog items op de lijst staan en zo lang de gebruiker nog wenst te winkelen, welk item gekocht is. Wanneer er een item wordt ingegeven dat op het lijstje staat, verwijder je dat van het lijstje.
- Als er op het einde nog niet-gekochte items over zijn, laat je zien welke items de gebruiker is vergeten te kopen.

**Tip:** voor lijsten is `Sort` een instantiemethode

### Voorbeeldinteractie

```
Welkom bij de demo Objectgeoriënteerd Programmeren!
Topic van de uit te voeren oefening?
1. DateTime
2. Eigen objecten
3. Datastructuren
3
Uit te voeren oefening?
1. H13-boodschappen
2. H13-kerstinkopen
3. H13-telefoonboek met naam en nummer
4. H13-telefoonboek met gemeente, naam en nummer
5. H13-telefoonboek met naam en nummer en builder
1
We gaan de boodschappenlijst samenstellen.
Wat is item 1 op je lijst? Geef een lege regel in om te stoppen.
koekjes
Wat is item 2 op je lijst? Geef een lege regel in om te stoppen.
melk
Wat is item 3 op je lijst? Geef een lege regel in om te stoppen.
pistolets
Wat is item 4 op je lijst? Geef een lege regel in om te stoppen.

Dit is je gesorteerde lijst:
1: koekjes
2: melk
3: pistolets
Op naar de winkel!
Welk item heb je gekocht? Geef de naam exact zoals hij op het lijstje staat.
1
Dit item bevindt zich niet op de lijst!
Nog winkelen? (Ja of Nee)
ja
Welk item heb je gekocht? Geef de naam exact zoals hij op het lijstje staat.
melk
Nog winkelen? (Ja of Nee)
nee
Naar huis met de boodschappen!
Volgende items van je lijst ben je vergeten te kopen:
koekjes
pistolets
```

## H13-telefoonboek (naam en nummer)

### Functionele analyse

We wensen een simpel telefoonboek bij te houden, waarin je namen en nummers plaatst.

### Technische analyse

- maak eerst een blanco Dictionary van string naar string aan
- vraag in een lus telkens of de gebruiker nog wil doorgaan en, zo ja, vraag om een naam en een nummer
- hou de koppeling van de naam en dat nummer bij
  - dit mag geen fout leveren als de naam al in het woordenboek staat - overschrijf in dat geval de waarde
    - je kan controleren met de instantiemethode `ContainsKey`
- toon tenslotte de inhoud van heel je telefoonboek
- noem je methode `TelefoonboekNaamNummer`

### Voorbeeldinteractie

```

Uit te voeren oefening?
1. H13-boodschappen
2. H13-kerstinkopen
3. H13-telefoonboek met naam en nummer
4. H13-telefoonboek met gemeente, naam en nummer
5. H13-telefoonboek met naam en nummer en builder
3
Wil je (nog) een naam en nummer inlezen?
ja
Naam?
Gebruiker 1
Nummer?
123456789
Wil je (nog) een naam en nummer inlezen?
ja
Naam?
Gebruiker 2
Nummer?
987654321
Wil je (nog) een naam en nummer inlezen?
ja
Naam?
Gebruiker 3
Nummer?
000111222
Wil je (nog) een naam en nummer inlezen?
ja
Naam?
Gebruiker 3
Nummer?
111222333
Wil je (nog) een naam en nummer inlezen?
nee
Gebruiker 1: 123456789
Gebruiker 2: 987654321
Gebruiker 3: 111222333
  
```

## **H13-telefoonboek (gemeente, naam en nummer)**

### **Functionele analyse**

Zie boven, maar we willen nu telefoonnummers ook groeperen per gemeente

### **Technische analyse**

- per gemeente heb je een Dictionary dat werkt zoals in de vorige oefening
  - om aan het Dictionary van een gemeente te komen, gebruik je een "groter" Dictionary met de naam van de gemeente als opzoekingsleutel
- achteraf print je de gegevens per gemeente, zoals dat ook in een fysiek telefoonboek ongeveer het geval is
- noem je methode `TelefoonboekGemeenteNaamNummer`

### **Voorbeeldinteractie**

```
Uit te voeren oefening?
1. H13-boodschappen
2. H13-kerstinkopen
3. H13-telefoonboek met naam en nummer
4. H13-telefoonboek met gemeente, naam en nummer
5. H13-telefoonboek met naam en nummer en builder
4
Wil je (nog) een gemeente, naam en nummer inlezen?
ja
Gemeente?
Antwerpen
Naam?
Gebruiker 1
Nummer?
123456789
Wil je (nog) een gemeente, naam en nummer inlezen?
ja
Gemeente?
Antwerpen
Naam?
Gebruiker 2
Nummer?
987654321
Wil je (nog) een gemeente, naam en nummer inlezen?
ja
Gemeente?
Antwerpen
Naam?
Gebruiker 1
Nummer?
111222333
Wil je (nog) een gemeente, naam en nummer inlezen?
ja
Gemeente?
Mechelen
Naam?
Gebruiker 3
Nummer?
999888777
Wil je (nog) een gemeente, naam en nummer inlezen?
nee
Gemeente: Antwerpen
Gebruiker 1: 111222333
Gebruiker 2: 987654321
Gemeente: Mechelen
Gebruiker 3: 999888777
```

## H13-telefoonboek-met-builder

### Functionele analyse



We willen graag dat ons Dictionary (zonder gemeente) veilig doorgegeven kan worden aan methodes enz. Daarom zullen we er een ImmutableDictionary van maken.

## Technische analyse

- start met aanmaak van een builder voor een ImmutableDictionary
- vraag de gegevens zoals in de eerdere oefening
- plaats deze stap voor stap in de builder (ook hier kan je ContainsKey gebruiken)
- zet, voor je alle gegevens print, om naar een ImmutableDictionary en pas daar een foreach lus op toe

## Voorbeeldinteractie

In de interactie zie je geen verschil [met de eerdere oefening](#).

## SchoolAdmin project

Als je alles eerder mee hebt kunnen volgen, werk dan vanaf je recentste commit.

**StudieProgramma.ToonOverzicht()** , **Cursus.ToonOverzicht()** en **Student.ToonOverzicht()** met **foreach**

Pas je ToonOverzicht-methodes aan zodat er geen gebruik wordt gemaakt van een klassieke `for` , maar wel van een `foreach` .

## Alle studenten in het systeem bijhouden

Voorzie de klasse Student van een statische read-only property `AlleStudenten` . Deze is van het type `List<Student>` en bevat altijd elke student die in het systeem aanwezig is. Dit gebeurt door bij de constructie van elk `Student` -object de lijst uit te breiden.

## AlleStudenten beveiligen

Maak van `AlleStudenten` een `ImmutableList<T>` in plaats van een gewone `List<T>` . Merk op dat je dit niet hoeft te doen voor het achterliggend attribuut.

## Lijsten

Vervang alle properties van `StudieProgramma` , `Cursus` en `Student` van een arraytype naar een `List` type. AlleCursussen maak je immutable.

Vervang hierbij ook `for` -lussen door `foreach` -lussen waar je kan. Je hoeft geen rekening te houden met capaciteiten die eerder zijn vastgelegd voor arrays. Je mag er ook van uitgaan dat er geen `null` waarden in lijsten worden geplaatst als dat niet zinvol is. Dit kan je code wat korter maken.

