Deel 16

Collectieklassen

We zagen eerder dat we meerdere objecten (of waarden) kunnen bundelen in arrays. Arrays worden in zelfgeschreven klassen echter zelden gebruikt omdat

- arrays groeien (of krimpen) niet vanzelf naarmate je er meer (of minder) elementen in bewaart
    - je moet dit zelf inschatten en ze expliciet *resizen* op de juiste momenten
- arrays bieden weinig extra functionaliteit zoals bv. een element middenin toevoegen
    - de klasse Array bevat wel een aantal handige static methods, maar het blijft beperkt. Bijvoorbeeld, Array.Resize() en Array.IndexOf().

In de .NET omgeving zitten verschillende collectieklassen waarmee elementen kunnen worden bijgehouden, zoals bv. **List**<T>, **LinkedList**<T> en **HashSet**<T>. Deze zijn Harder, Better, Faster, Stronger en veel handiger om mee te werken. Ze verschillen onderling in de mate waarop ze efficient omgaan met het geheugen en CPU cycles, of hun geordend zijn en of ze dubbels toegelaten.

Er is dus geen 'beste' collectieklasse, men kiest het soort collectie op basis van wat men ermee wil doen.

Het **<T>** achtervoegsel duidt op het soort element dat we erin kunnen stoppen, bijvoorbeeld

- List**<int>** : een lijst van **int** waarden
- HashSet**<Persoon>** : een verzameling van verwijzingen naar **Persoon** objecten
- List<**string**> : een lijst van **string**s

In een List<int> kun je dus geen string, double of Persoon reference stoppen : enkel int waarden.

We bekijken drie eenvoudige voorbeelden :

- List<T>
- LinkedList<T>
- HashSet<T>

List<T>

Dit is een **lijst** van elementen in een bepaalde **volgorde**, je kan dus bv. het 5e element opvragen. Intern worden de objecten in een array bijgehouden maar daar merk je niks van (inkapseling). Het voordeel is dat het **opvragen van een willekeurig element quasi ogenblikkelijk** is. Het nadeel is dat als er een element middenin toegevoegd of verwijderd moet worden, er gemiddeld de helft van de element moet opgeschoven worden (zodat de volgorde in de lijst niet wijzigt). Bovendien, als het interne array te klein blijkt, komt er een *resize* waarbij alle verwijzingen gekopieerd worden naar een groter array (dat dan het nieuwe interne array wordt). Om te vermijden dat dit kopiëren te vaak gebeurt, zal het interne array altijd wat groter ingesteld worden dan het aantal elementen in de lijst, wat ruimte verspilt maar tijd spaart.

LinkedList<T>

Dit is een **lijst** van elementen in een bepaalde **volgorde**. Intern worden objecten bijgehouden in een ketting van schakels. Elk schakel-object bevat een verwijzing naar de volgende (en vorige) schakel in de ketting. Het voordeel daarvan is dat **middenin inlassen of verwijderen heel efficient** kan gebeuren door een paar verwijzingen in de schakels aan te passen. Het nadeel is echter dat het 1001e element opvragen relatief lang duurt omdat eerst de 1000 voorafgaande schakels moeten overlopen worden vooraleer we bij schakel 1001 komen. Wijzigingen vooraan of achteraan de lijst zijn efficiënt.

HashSet<T>

Dit is een **verzameling** van elementen **zonder vaste positie** en er kunnen geen dubbels in voorkomen. Vermits de elementen geen onderlinge volgorde hebben, zal het overlopen met foreach loop in "één of andere" volgorde gebeuren. Je hebt dus geen garanties over de volgorde waarin je ze tegenkomt. Opvragen van elementen, toevoegen en verwijderen gebeurt allemaal zeer efficient. De interne structuren zullen soms eens moeten groeien als we elementen toevoegen.

De keuze tussen deze 3 soorten collecties is doorgaans vrij eenvoudig als je hun voor- en nadelen kent :

- HashSet<T> is aangewezen als de elementen niet in een bepaalde volgorde moeten zitten en je geen dubbels in de verzameling nodig hebt of dubbels wil vermijden.

- LinkedList<T> gebruik je als je oog hebt voor performantieproblemen en de elementen steeds in volgorde wil overlopen (dus zelden zomaar een element middenin opvraagt of Contains() gebruikt) en als toevoegingen en verwijderingen vooral vooraan of achteraan de lijst gebeuren.

- List<T> gebruik je in alle andere gevallen.

Bekijk de properties en methods van deze klassen goed zodat je min of meer weet welke mogelijkheden ze bieden. Als je bv. niet weet dat er zoiets als IndexOf() bestaat ben je gedoemd om dit soort zoek-code zelf te schrijven en er zijn nuttigere manieren om je tijd te besteden. De interessantste zijn wellicht :

List<T>	LinkedList<T>	HashSet<T>
Count	Count	Count
[index]	ElementAt	
		Add
Add	AddFirst, AddLast	Contains
Insert	Contains	Remove
Contains	Remove	
IndexOf		ExceptWith
Remove		IntersectWith
RemoveAt		UnionWith
Sort		Overlaps

Belangrijk : alle collectieklassen zitten in de namespace **System.Collections.Generic** , dus je zult wellicht de volgende regel in je source code files willen plaatsen als je met collecties gaat werken :

```
using System.Collections.Generic;
```

Een voorbeeld met List<string> :

```
List<string> woorden = new List<string>();                  // een lege lijst maken
woorden.Add("hello");                                       // toevoegen achteraan de lijst
woorden.Add("world");                                       // toevoegen achteraan de lijst
Console.WriteLine( woorden.Count );                         // toont 2
woorden.Insert( 1, "small" );                               // lijst is nu : "hello", "small", "world"
Console.WriteLine( woorden[2] );                            // toont world
Console.WriteLine( woorden.IndexOf("small") ); // toont 1
woorden[0] = "simple";                                      // lijst is nu : "simple", "small", "world"
woorden.RemoveAt( 1 );                                      // verwijdert "small"
```

Als kleine terzijde : alle collecties aanvaarden 'null' als element indien het item type <T> een reference type is (bv. string of Persoon).

Bv.

```
List<string> woorden = new List<string>();
woorden.Add( null );                                        // null toevoegen aan List
HashSet<Persoon> personen = new HashSet<Persoon>();
personen.Add( null );                                       // null toevoegen aan HashSet
```

Doorgaans probeer je echter 'null' waarden in collecties te vermijden. Immers, als je code iets met de elementen wil doen (bv. één van hun methods oproepen) moet er extra code voorzien worden om de null waarden te vermijden zodat er geen 'NullReferenceException' fouten optreden.

Indien het item type <T> van een collectie geen reference type is, bv. List<**int**>, dan mag je er geen null waarden in stoppen.

Belangrijk : elk van deze klassen heeft zowel

- een default constructor (constructor zonder parameters)
    - deze maakt een lege collectie
- een constructor waaraan je een andere collectie (of array) kan meegeven als parameter
    - deze maakt een collectie aan met dezelfde elementen als de andere collectie (of array)

Er is ook een notatie die het wat makkelijker maakt om een collectie te vullen met literal values.

Enkele voorbeelden om collecties aan te maken :

```
// maak een nieuwe lege lijst
List<DateTime> momenten = new List<DateTime>();

// maak een nieuwe lege set
HashSet<Persoon> personen = new HashSet<Persoon>();

// maak een nieuwe lijst en vul ze met literals
List<string> woorden = new List<string> { "er", "was", "eens", "de", "was" };

// maak een nieuwe lijst met dezelfde elementen als een andere lijst (i.e. maak een kopie)
List<string> kopie = new List<string>( woorden );

// maak een nieuwe set en vul deze met de elementen van een lijst
// (handig om dubbels te elimineren!)
HashSet<string> uniekeWoorden = new HashSet<string>( woorden );
```

Een collectie kun je mooi op de console krijgen door de gekende String.Join method te gebruiken :

```
List<string> woorden = new List<string> { "er", "was", "eens", "de", "was" };
string listAsText = String.Join ( ", " , woorden);
Console.WriteLine( listAsText );
```

Dit toont :

```
er, was, eens, de, was
```

Let op, er zijn geen garanties zijn qua volgorde bij een HashSet!

Alle drie collectieklassen ondersteunen trouwens foreach loops.

Bv.

```
foreach (Persoon p in personen) {                   // foreach met een List
        Console.WriteLine( p.Voornaam );
}

foreach (string woord in uniekeWoorden) {           // foreach met een HashSet
        Console.WriteLine( woord );
}
```

Er zijn geen garanties over de volgorde waarin de elementen van een HashSet aan bod komen!

Let op, als er 'null' waarden in de collectie kunnen voorkomen, moet je daar goed rekening mee houden.

Bv.

```
foreach (Persoon p in personen) {
        if ( p != null ) {                                  // extra null check nodig!
                Console.WriteLine( p.Voornaam ); // om NullReferenceException te vermijden
        }
}
```

Belangrijk : je mag de inhoud van de collectie niet wijzigen (elementen toevoegen of verwijderen) terwijl je ze overloopt in een foreach loop!

Bv.

```
List<string> woorden = new List<string> { "groen", "ui", "bak", "asfalt", "vlakken" };
foreach (string woord in woorden) {
        // verwijder alle te korte woorden
        if (woord.Length <= 3) {
                woorden.Remove(woord);
        }
}
```

Deze code geeft de volgende error wanneer geprobeerd wordt een kort woord te verwijderen:

Twee manieren die wel werken om tijdens het overlopen een element te verwijderen :

1. de lijst overlopen met een gewone for loop, maar dan van achteren naar voren
    - zodat je het volgende element niet overslaat telkens je een element verwijdert met **RemoveAt**(). Hieronder staat een voorbeeld dat dit demonstreert.

2. tijdens de foreach loop worden de "te verwijderen elementen" in andere verzameling onthouden en pas na de foreach worden ze verwijderd met Remove() of ExceptWith().

De eerste manier is handiger voor List en LinkedList

- HashSet heeft immers geen RemoveAt()

De tweede manier is handiger voor HashSet

- In List en LinkedList verwijdert Remove() enkel het eerste voorkomen van een element als er dubbels zijn (je zou dus een loop moeten schrijven totdat Remove() de waarde false oplevert). De method Remove() zoekt ook steeds vanaf het begin van de lijst, wat niet zo efficient is.

Ivm **mogelijkheid 1**, merk op dat dit niet werkt:

```
List<string> woorden = new List<string> { "groen", "ui", "bak", "asfalt", "vlakken" };
for ( int i = 0 ; i < woorden.Count ; i++ ) {
   string woord = woorden[i];
   if (woord.Length <= 3) {
      woorden.Remove(woord);
   } else {
      Console.WriteLine(woord);
   }
}
```

Op het moment dat het woordje "ui" op positie 1 verwijderd wordt, schuift woordje "bak" op naar positie 1. Daarna wordt de teller 'i' verhoogd naar 2, waardoor we "asfalt" checken. We hebben dus het woordje "bak" overgeslaan!

Door de lijst achterstevoren te doorlopen hebben we dit probleem niet :

```
List<string> woorden = new List<string> { "groen", "ui", "bak", "asfalt", "vlakken" };
for ( int i = woorden.Count-1 ; i >= 0 ; i - -) {
   string woord = woorden[i];
   if (woord.Length <= 3) {
      woorden.Remove(woord);
   } else {
      Console.WriteLine(woord);
   }
}
```

Mogelijkheid 2 zou er als volgt kunnen uitzien :

```
List<string> woorden = new List<string> { "groen", "ui", "bak", "asfalt", "vlakken" };
List<string> ongewensteWoorden = new List<String>();

// zoek de ongewenste woorden
foreach (string woord in woorden) {
   if (woord.Length <= 3) {
      ongewensteWoorden.Add(woord);
   }
}

// verwijder nu alle ongewenste woorden
foreach(string ongewenstWoord in ongewensteWoorden) {
   woorden.Remove(ongewenstWoord);
}

foreach(string woord in woorden) {
   Console.WriteLine(woord);
}
```

Merk op dat in beide foreach loops, er telkens twee lijsten zijn : de lijst die we overlopen en de lijst waar we aan sleutelen (toevoegen of verwijderen).

Dit werkt ook correct als er dubbels in de lijst voorkomen. De dubbels komen gewoon meermaals in 'ongewensteWoorden' terechtkomen en zullen dus ook meermaals verwijderd worden.

Voor een HashSet kan het verwijderen natuurlijk veel eenvoudiger d.m.v. ExceptWith().