Deel 17

Collectieklassen (vervolg)

In deze les behandelen we een totaal ander soort collectieklasse : **Dictionary** <TKey, TValue>.

De term 'dictionary' doet je wellicht aan een woordenboek denken : een object waarin je de betekenis kan opzoeken van een woord. Het opzoekwerk is makkelijk in deze richting : woord → betekenis.

Dat is ook de bedoeling van Dictionary <TKey, TValue> objecten. We gebruiken ze om een 'value' op te zoeken die bij een bepaalde 'key' hoort, dus opzoeken in deze richting : key → value.

De **TKey** en **TKey** geven aan wat de types zijn van de key en value elementen, net als de T bij List<T>.

Enkele voorbeelden om dit te verduidelijken :

- bijhouden wat de tekstvoorstelling van bepaalde gehele getallen is
    - bv. 13 → "dertien", 8 → "acht", 90 → "negentig", 1130 → "elvendertig", 1337 → "leet"
    - de keys waarop we kunnen zoeken zijn de getalllen, bv. 13, dus TKey is int
    - de value die bij een key hoort is de tekstvoorstelling, bv. "dertien", dus TValue is string
    - we bewaren deze overeenkomsten in een Dictionary<int, string>
- bijhouden hoe vaak woorden in een tekst voorkomen
    - bv. "het" → 34, "de" → 64, "zijn" → 34, "hoofd" → 2, "snor" → 87
    - de keys waarop we kunnen zoeken zijn de woorden, bv. "het", dus TKey is string
    - de value die bij een key hoort is het aantal, bv. 34, dus TValue is int
    - we houden de tellingen bij in een Dictionary<string, int>
- bijhouden wat de score is van de spelers in een wedstrijd
    - bv. speler Jan → 7, speler Griet → 5, speler Bart → 9, speler Stefanie → 9
        - veronderstel dat spelers worden voorgesteld door objecten van klasse Speler
    - de keys waarop we kunnen zoeken zijn de spelers, bv. speler Jan
        - dit zijn references naar Speler objecten, dus TKey is Speler
    - de value die bij een key hoort is een score, bv. 7, dus TValue is int
    - we houden de wedstrijd scores bij in een Dictionary<Speler, int>
- bijhouden wie welk telefoonnummer heeft
    - bv. "Jan" → "123456", "Piet" → "567890", "Cornelia" → "123456"
        - dit keer gebruiken we eens gewoon de naam en geen Persoon object
    - de keys waarop we kunnen zoeken zijn de namen, bv. "Jan", dus TKey is string
    - de value die bij een key hoort is een telefoonnummer, bv. "123456", dus TValue is string
    - we gebruiken een Dictionary<string, string> als simpel telefoonboek

- bijhouden wie de eigenaar is van welke hond
    - bv. hond Bobbie → persoon Kuifje, hond Pekkie → persoon Filiberke
        - veronderstel dat honden worden voorgesteld door objecten van klasse Hond
        - veronderstel dat person worden voorgesteld door objecten van klasse Persoon
    - de keys waarop we kunnen zoeken zijn de honden, bv. hond Bobbie
        - dit zijn references naar Hond objecten, dus TKey is Hond
    - de value die bij een key hoort is een persoon, bv. persoon Kuifje
        - dit zijn references naar Persoon objecten, dus TValue is Persoon
    - we houden de koppelingen bij in een Dictionary<Hond, Persoon>

Voor de naamgeving van variabelen en parameters gebruiken we vaak "**xNaarY**", bijvoorbeeld

Dictionary<int, string> **getalNaarTekst** = ...
Dictionary<string, int> **woordNaarAantal** = ...
Dictionary<Speler, int> **spelerNaarScore** = ...
Dictionary<string, string> **naamNaarTelefoon** = ...
Dictionary<Hong, Persoon> **hondNaarEigenaar** = ...

Een dictionary beheert dus **een verzameling key/value koppelingen** (aan elke key wordt een value gekoppeld) en kan **snel de value opzoeken** voor een bepaalde key. Soms noemt men zo'n structuur ook wel een 'map' of 'mapping' of 'associative array', maar in .NET heet het dus een dictionary.

Belangrijk : enkele beperkingen van de Dictionary<TKey, TValue> klasse :

- een key kan maar één keer voorkomen in een Dictionary
    - 'null' mag trouwens niet als key gebruikt worden
- elke key heeft precies één value
    - 'null' als value is toegelaten, indien TValue een reference type is
- meerdere keys kunnen eenzelfde gekoppelde value hebben
    - in de voorbeelden
        - hadden spelers Bart en Stefanie dezelfde score 9
        - kwamen de woorden "het" en "zijn" beiden 34 keer voor in de tekst
        - deelden "Jan" en "Cornelia" hetzelfde telefoonnummer "123456"
- opzoeken welke keys een specifieke gekoppelde value X hebben is complex en relatief traag
    - Je moet voor elke key de value opvragen en vergelijken met value X
        - indien ze gelijk zijn heb je een key gevonden waaraan X gekoppeld is
    - Je moet dit wel voor alle keys doen : er kunnen meerdere keys zijn die value X hebben
        - als je zoekt wie telefoonnummer "123456" heeft mag je niet stoppen bij "Jan" want "Cornelia" heeft dat telefoonnummer ook

Bestudeer aandachtig de documentatie van de Dictionary<TKey, TValue> klasse, zodat je weet welke mogelijkheden ze biedt. De interessantste zijn wellicht :

Property of Method	Uitleg
.Count	het aantal keys in de collectie (een int)
[*key*]	de value opvragen van een bestaande *key* (of een value koppelen aan *key*)
.Keys	een verzameling van alle keys opvragen
.Values	een verzameling van alle values opvragen
.ContainsKey(*key*)	Bevat het dictionary de gegeven *key*? Retourneert een bool waarde - als de parameter null is, komt er een ArgumentNullException fout
.ContainsValue(*value*)	Bevat het dictionary de gegeven *value*? Retourneert een bool waarde
.Remove(*key*)	Verwijder de value voor de gegeven *key* (en de key zelf ook). Retourneert een bool waarde die aangeeft of er daadwerkelijk iets verwijderd werd.
.Clear()	Verwijder alle values van alle keys alsook alle keys

Let op : als je met [*key*] de value opvraagt van een onbestaande *key*, krijg je een KeyNotFoundException!

De Dictionary<TKey, TValue> klasse zit eveneens in de System.Collections.Generic namespace, dus je zult wellicht de volgende regel bovenaan in je broncode bestand willen plaatsen :

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

Je kunt op de volgende manieren een nieuw Dictionary object aanmaken :

```
// een leeg dictionary
Dictionary<Speler, int> spelerNaarScore = new Dictionary<Speler, int> ( );

// een dictionary met literal values
Dictionary<int, string> getalNaarTekst = new Dictionary<int, string> {
        {13, "dertien"},
        {5, "vijf"}
};

// een dictionary gevuld met de koppelingen uit een ander dictionary
Dictionary<int, string> kopie = new Dictionary<int, string> ( getalNaarTekst );
```

Voorbeeld :

```
Dictionary<string, string> telefoonNaarNaam = new Dictionary<string, string> {
   {"764-8437", "Moe Szyslak" },
   {"555-0001", "Montgomery Burns"},
   {"555-6832", "Homer Simpson"}
};

telefoonNaarNaam["555-8707"] = "Homer Simpson";   // value koppelen aan nieuwe key (toevoegen)
Console.WriteLine( telefoonNaarNaam["555-8707"] );  // value opvragen van key, toont Homer Simpson

telefoonNaarNaam["555-8707"] = "Marge Simpson";   // value koppelen aan bestaande key (overschrijven)
Console.WriteLine( telefoonNaarNaam["555-8707"] ); // value opvragen van key, toont Marge Simpson

Console.WriteLine( telefoonNaarNaam.ContainsKey( "555-0001" ) );          // toont True
Console.WriteLine( telefoonNaarNaam.ContainsKey( "555-8904" ) );          // toont False

Console.WriteLine( telefoonNaarNaam.ContainsValue( "Marge Simpson" ) );   // toont True
Console.WriteLine( telefoonNaarNaam.ContainsValue( "Homer Simpson" ) );   // toont False

Console.WriteLine( String.Join( ", ", telefoonNaarNaam.Keys ) );
Console.WriteLine( String.Join(", ", telefoonNaarNaam.Values ) );

string naam = telefoonNaarNaam["12345"];  // dit geeft een KeyNotFoundException fout !!!
```

Merk op hoe [*key*] zowel voor lezen als schrijven gebruikt wordt, net als [*index*] bij een List<T> of array.

- schrijven : telefoonNaarNaam["555-8707"] **= "Homer Simpson";**
- lezen : **string naam =** telefoonNaarNaam["555-8707"];

Samengevat :

- elke key heeft exact één value
- er worden geen keys bijgehouden die geen value hebben
- null is een geldige value, maar geen geldige key
- meerdere keys kunnen dezelfde value hebben
- we kunnen de value opzoeken als we de key hebben, maar niet omgekeerd
- met [*key*] de value van een onbestaande key opvragen, resulteert in een KeyNotFoundException
    - je kunt dit vermijden door bv. eerst te checken met .ContainsKey(*key*)

Als je alle waarden in een dictionary wil overlopen, bv. om ze met hun corresponderende waarde op de console te zetten, kun je de .Keys property en een foreach loop gebruiken.

Bv.

```
Dictionary<string, string> telefoonNaarNaam = new Dictionary<string, string> {
   {"764-8437", "Moe Szyslak" },
   {"555-0001", "Montgomery Burns"},
   {"555-6832", "Homer Simpson"},
   {"555-8707", "Homer Simpsons"}
};

foreach(string telefoon in telefoonNaarNaam.Keys) {        // overloop alle keys
   string naam = telefoonNaarNaam[telefoon];               // vraag value van deze key
   Console.WriteLine($"{telefoon} hoort bij {naam}");      // key en value tonen
}
```

Op zich is dit niet erg efficiënt want we moeten voor elke key de corresponderende value opzoeken. Er zijn betere manieren (via KeyValuePair objecten) maar die laten we hier buiten beschouwing.

Merk op dat de **.Values** property dubbels kan bevatten, maar de **.Keys** property natuurlijk niet. Er is trouwens geen vastgelegde volgorde voor de keys of values in een dictionary, maar de volgorde in .Keys en .Values is wel op elkaar afgestemd.

Bv.

```
Dictionary<string, string> telefoonNaarNaam = new Dictionary<string, string> {
   {"764-8437", "Moe Szyslak" },
   {"555-0001", "Montgomery Burns"},
   {"555-6832", "Homer Simpson"},
   {"555-8707", "Homer Simpson"}
};

Console.WriteLine( String.Join(", ", telefoonNaarNaam.Keys) );
Console.WriteLine( String.Join(", ", telefoonNaarNaam.Values) );
```

Dit toont

```
764-8437, 555-0001, 555-6832, 555-8707
Moe Szyslak, Montgomery Burns, Homer Simpson, Homer Simpsons
```

Meerdere waarden per key bijhouden?

Indien je meerdere waarden aan een key wil koppelen, moet je een truc toepassen. Als je voor TValue een collectieklasse kiest, bv. HashSet<string> zal er zoals altijd aan elke key één value gekoppeld worden. Op zich niks bijzonders zou je denken.

Maar in dit geval is die value een HashSet<string> object waarin we meerdere strings kunnen bewaren!

Bijvoorbeeld :

key is een string		value is een HashSet<string>
"Jan"	→	{ "123456", "987654", "673984"}
"Piet"	→	{ "567890" }
"Cornelia"	→	{ "123456", "563245" }

In onze code ziet het er dan zo uit :

```
Dictionary< string, HashSet<string> > naamNaarTelefoons = …
```

Deze creatieve oplossing vergt wel extra code bij vrijwel elke manipulatie van het Dictionary object.

Bijvoorbeeld, als we een telefoonnummer willen toevoegen van Harry en we weten niet op voorhand of er al nummers gekend zijn voor Harry :

```
Dictionary< string, HashSet<string> > naamNaarTelefoonNummers = …
...
// voeg "325476" toe als telefoonnummer voor Harry
string naam = "Harry";
HashSet<string> telefoons;
if (naamNaarTelefoons.ContainsKey( naam ) {          // is er al een HashSet value voor Harry?
        telefoons = naamNaarTelefoons[ naam ];       // ja, vraag de HashSet value voor Harry
} else {
        telefoons = new HashSet<string>();           // neen, maak een nieuwe HashSet
        naamNaarTelefoons[ naam ] = telefoons;       // en voeg deze toe als value voor Harry
}
telefoons.Add("325476");                    // voeg het nummer toe aan de HashSet voor Harry
```

Voor verwijderingen en opzoekingen zullen we ook extra code moeten schrijven om met de HashSets rekening te houden.

Parallelle lists

We hebben eerder de techniek van **parallelle arrays** besproken : stukjes data die met elkaar te maken hebben, worden op dezelfde positie in verschillende arrays geplaatst.

Bijvoorbeeld :

> We verdelen de data van een persoon over 3 parallelle arrays : namen, postcodes en gemeenten. De gegevens van Jan staan dan verspreid over namen[6], postcodes[6] en gemeenten[6]. Cornelia's info staat in namen[3], postcodes[3] en gemeenten[3], enz..

Bij **parallele lists** passen we dezelfde techniek toe op bv. List<T> objecten i.p.v. arrays.

Welnu, met twee parallelle lists kunnen we min of meer hetzelfde doen als met een dictionary!

Veronderstel dat we deze twee variabelen en objecten hebben :

```
Persoon kuifje = new Persoon ("Kuifje", ...) ;
Hond bobbie = new Hond ("Bobbie", ...) ;
```

Alles wat we kunnen doen met

```
Dictionary<Hond, Persoon> hondNaarEigenaar = new Dictionary<Hond, Persoon> () ;
```

kunnen we evenzeer realiseren met twee parallelle lists

```
List<Hond> honden = new List<Hond> ();              // een lijst met keys
List<Persoon> eigenaars = new List<Persoon> ();     // een lijst met values
```

Bijvoorbeeld

> Toevoegen dat Kuifje de eigenaar is van Bobbie
>
> > In plaats van
> >
> > ```
> > hondNaarEigenaar[ bobbie ] = kuifje;
> > ```
> >
> > gebruiken we
> >
> > ```
> > honden.Add( bobbie );
> > eigenaar.Add( kuifje );
> > ```

Opzoeken wie de eigenaar is van Bobbie

In plaats van

```
Persoon p = hondNaarEigenaar[ bobbie ];
```

gebruiken we

```
int idx = honden.IndexOf( bobbie );
Persoon p = eigenaars[ idx ];
```

Je kunt je dus de vraag stellen, waarom bestaat Dictionary<TKey, TValue> eigenlijk? De hoofdreden is dat **de value die bij een key hoort wordt quasi ogenblikkelijk gevonden**.

Bij parallelle lists is het zoeken relatief traag, want de IndexOf() method moet gemiddeld de helft van alle keys checken vooraleer de juiste positie gevonden wordt.