HO
GENT

Collections

Ludwig Stroobant

Table of Contents

1. Doelstellingen

- Kan de juiste generieke datastructuur uit het collections framework kiezen
- Kan de juiste generieke datastructuur uit het collections framework correct gebruiken
- Kan `IEnumerable<T>` gebruiken om een collectie te doorlopen
- Kan meerdere methodes uit het collections framework toepassen om collecties te verwerken

2. Inleiding

In dit hoofdstuk gaan we een blik werpen op de C# Collections en Data Structures. Bij software ontwikkeling komen collections, losweg vertaald 'groepen van objecten', 'verzamelingen', veelvuldig voor.

Stel je bijvoorbeeld software voor om een kaartspel te beheren. De stapel kaarten is een collectie van kaart-objecten, de verschillende spelers vormen een collectie van Speler-objecten, en de kaarten in die bij elke speler horen zijn ook collecties van Kaart-objecten.

Als je denkt aan een webshop kan je je ook verschillende collections voorstellen. De webshop biedt *producten* aan, beheert zijn *klanten*, *bestellingen*, ...

3. Namespace `System.Collections`

> Een **collection** is een datastructuur, i.e. een object, die een verzameling van objecten als hetzelfde type groepeert. De objecten in de verzameling noemen we de **elementen**.
>
> De **System.Collections namespace** biedt een waaier van interfaces en klassen aan die toelaten op een flexibele manier verzamelingen van objecten te beheren en te manipuleren.

Door gebruik te maken van wat het collections framework aanreikt hoef je niet alle details van de interne representatie van collections te kennen of zelf te implementeren, en hoef je ook niet zelf algoritmes te implementeren om bijvoorbeeld specifieke elementen in een verzameling te vinden, of je verzameling te sorteren.

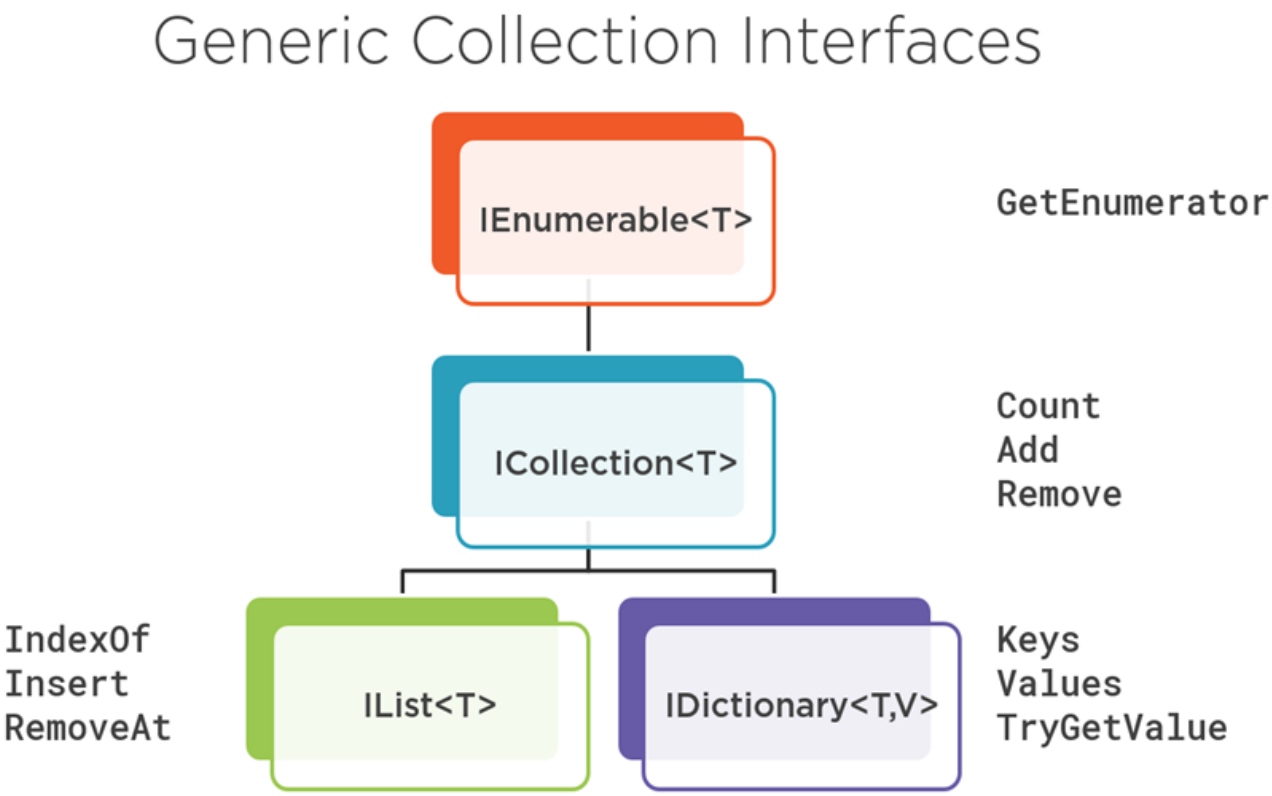
Het is een uitdaging voor de ontwikkelaar om vat te krijgen op dit framework. Verschillende collections hebben immers verschillende eigenschappen. Wil je een verzameling waarin je dubbels kan opslaan? Wil je een verzameling waarin de elementen gesorteerd zitten? Is het belangrijk dat je in je verzameling vliegensvlug elementen kunt opzoeken, of heeft het snel toevoegen en verwijderen van elementen een hoge prioriteit voor je?

Dergelijke eigenschappen kunnen mede bepalen met welke collection je aan de slag gaat. Met kennis van het collections framework zal je in staat zijn op een gefundeerde manier de meest geschikte collection te kiezen voor een taak en eveneens op een gepaste manier gebruik te maken van alle functionaliteit die het framework aanbiedt om performante software op een elegante en flexibele manier te ontwikkelen.

In dit hoofdstuk gaan we slechts een deel van de beschikbare collections toelichten. De zaken die je over collections leert zullen ook een goede basis vormen om collection frameworks in andere programmeertalen te leren begrijpen. Iemand die vertrouwd is met C# collections zal bijvoorbeeld snel vat krijgen op het JAVA Collections Framework, en omgekeerd…

- het collections framework is onderdeel deel van de `System.Collections` namespace
- het collections framework is ***non_generic*** en ***generic*** opgezet. Generic collections zijn type-safe tijdens compileren. **Wij maken zoveel mogelijk gebruik van de generic collections.**
  - de generic interfaces en klassen hebben een geparametriseerd type en bevatten dus herbruikbare code

4. Verkort overzicht collection interfaces



5. Interface IEnumerable<T>

Elke collection erft van de interface IEnumerable<T>. Deze interface voorziet een contract om te itereren over een generieke groep van elementen.

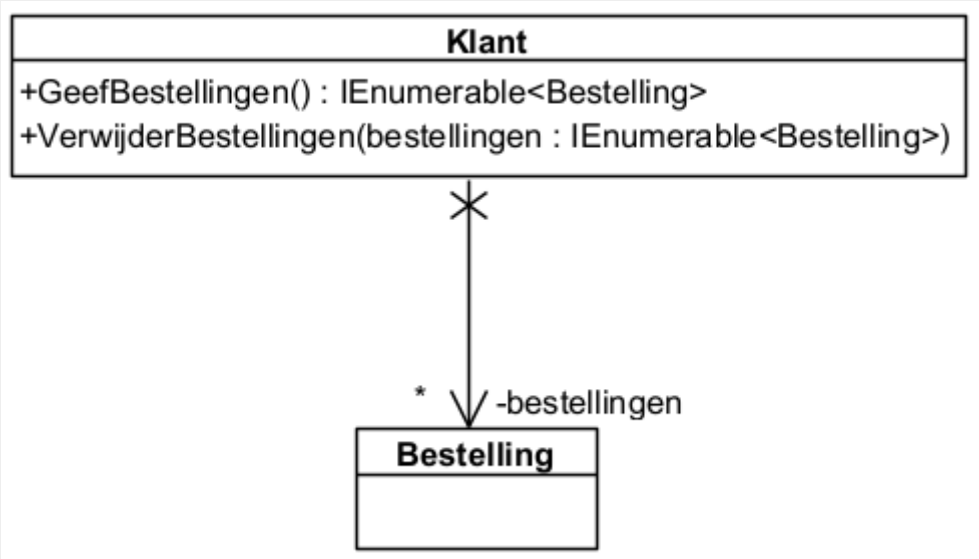
Elke collection kan op basis van deze interface overlopen worden met een `foreach` statement → *tijdens dergelijke iteratie mag de collectie* ***niet gewijzigd worden***!

> The `foreach` statement is a wrapper around the enumerator, which only allows reading from, not writing to, the collection.
>
> — Microsoft API

De implementatie van deze interface volgt het 'Iterator design pattern'.

> The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying represenation.
>
> — According to GoF (The Gang of Four)

Op deze manier kan een klasse op een vrij **algemene manier functionaliteiten aanbieden terwijl de implementatiedetails van de collection verborgen blijven** voor de client code (dit zijn andere klassen die van de aangeboden functionaliteit gebruik maken). De concrete implementatie van de collection zit ingekapseld en kan desgewenst gewijzigd worden zonder dat de client code hiervoor hoeft aangepast te worden.



- de klasse Klant retourneert alle bestellingen van een klant als een IEnumerable<Bestelling>. De concrete collection klasse die de bestellingen beheert, blijft verborgen voor de client code en kan desgewenst gewijzigd worden zonder dat dit invloed heeft op de cliënt code
- aan de klasse Klant kan gevraagd worden om een aantal bestellingen te verwijderen, de cliënt heeft de vrijheid om te kiezen hoe deze verzameling concreet zal aangeleverd worden
- merk op hoe de generische parameter <T> in IEnumerble<**T**> hier een invulling kreeg met het type **Bestelling**

Program to an interface, not an implementation!

5.1. Extension methods en LINQ

Een extension method is een manier om bestaande klassen of interfaces uit te breiden zonder ze daadwerkelijk te wijzigen of te erven van de desbetreffende klasse of interface.

Extension methods stellen ontwikkelaars in staat om nieuwe methoden toe te voegen aan bestaande typen zonder de broncode van die typen aan te passen. Dit is handig omdat het de mogelijkheid biedt om functionaliteit toe te voegen aan klassen die zich in externe assemblies bevinden.

Een extension method is gedefinieerd als een `static` methode in een `static` klasse. De methode moet een speciaal parameter genaamd `this` hebben, gevolgd door het type waaraan je functionaliteit wilt toevoegen.

Hier is een eenvoudig voorbeeld van een extension method:

```
public static class StringExtensions
{
    public static bool IsPalindrome(this string input)
    {
        string reversed = new string(input.Reverse().ToArray());
        return input.Equals(reversed, StringComparison.OrdinalIgnoreCase);
    }
}
```

In dit voorbeeld wordt de `IsPalindrome`-methode als een extension method toegevoegd aan de `string`-klasse. Hierdoor kun je de `IsPalindrome`-methode aanroepen op een string-object zoals dit:

```
string word = "racecar";
bool isPalindrome = word.IsPalindrome(); // Roep de extension method aan
```

LINQ (Language Integrated Query) is een onderdeel van C# dat gebruikmaakt van extension methods om query's uit te voeren op verzamelingen van gegevens, zoals arrays, lijsten en databases. LINQ biedt een gestructureerde en consistente manier om gegevens te doorzoeken, te filteren, te transformeren en te projecteren.

De LINQ-bibliotheek maakt uitgebreid gebruik van extension methods om query-operaties uit te voeren op verzamelingen. Bijvoorbeeld, je kunt de `Where`, `Select`, `OrderBy`, en vele andere LINQ-methoden aanroepen op een collectie van objecten. Deze methoden maken gebruik van extension methods om het gedrag van de collecties uit te breiden en bieden een elegante manier om gegevens te manipuleren.

Het belangrijkste type dat wordt uitgebreid door LINQ is IEnumerable<T>, dat een generieke interface is die wordt geïmplementeerd door de meeste collecties in C#. Hierdoor kunnen LINQ-query's worden toegepast op verschillende verzamelingen,

zoals arrays, lijsten, sets en meer, zonder dat de specifieke verzameling zelf moet worden gewijzigd.

Hier is een voorbeeld van hoe LINQ extension methods worden gebruikt om gegevens te filteren en te projecteren:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var evenNumbers = numbers.Where(n => n % 2 == 0); // Filtert even getallen
var squaredNumbers = numbers.Select(n => n * n);   // Kwadrateert elk getal
```

In dit voorbeeld worden de `Where`- en `Select`-methods van LINQ gebruikt om de `numbers`-collectie te filteren en te transformeren. Deze methoden maken gebruik van extension methods om de collectie aan te vullen met de gewenste functionaliteit zonder de `List<T>`-klasse zelf aan te passen.

Zoek enkele LINQ methodes op in de C# API en bekijk de codevoorbeelden. Neem ze over in een voorbeeld programma en voer ze uit. Leer ze gebruiken, zo zal je later veel tijd besparen.

6. Interface ICollection<T>

De interface `ICollection<T>` vormt de basis, de root van het collection framework. Alle andere interfaces en concrete klassen die we in dit hoofdstuk bespreken stammen af van deze interface. Er zijn trouwens geen directe concrete afstammelingen van deze interface. Wel zijn er enkel andere interfaces, sub-interfaces, die deze interface uitbreiden, zoals de interfaces `IList<T>` en `IDictionary<TKey, TValue>`.

De interface ICollection<T> heeft een contract voor enkele algemene methodes, geldig voor elk type collectie.

De op deze manier aangeboden functionaliteit laat bijvoorbeeld toe om het aantal elementen die in een verzameling zitten op te vragen, na te gaan of de verzameling al dan niet leeg is, of ze een bepaald element al dan niet bevat, en laten eveneens toe om een element toe te voegen en te verwijderen...

De `Contains`-methode retourneert of een element al dan niet aanwezig is in een collection en maakt gebruik van de `Equals`-methode om op gelijkheid te testen. Dit is een belangrijk gegeven als we met het collection framework aan de slag gaan. We moeten zorgen dat deze, en dan uiteraard ook de `GetHashcode`-methode, een goede invulling heeft gekregen die de correcte notie van gelijkheid van elementen uitdrukt.

7. Interface IList<T>

De `IList<T>` interface staat voor een geïndexeerde collectie van objecten die kan

> worden benaderd met behulp van een integer-index. `IList<T>` is ontworpen om een flexibele en uitbreidbare lijst (of array-achtige) gegevensstructuur te vertegenwoordigen en biedt methoden voor het toevoegen, verwijderen, doorzoeken en wijzigen van elementen in de lijst.

Hier zijn enkele belangrijke eigenschappen en methoden die worden geërfd van de `IList<T>` interface:

1. **Count**: Dit is een eigenschap die aangeeft hoeveel elementen er in de lijst zijn.
2. **IsReadOnly**: Dit is een eigenschap die aangeeft of de lijst alleen-lezen is. Als deze eigenschap `true` is, kunnen elementen niet worden toegevoegd of verwijderd.
3. **Item[int index]**: Dit is de indexeringsmethode waarmee je een element in de lijst kunt ophalen of wijzigen op basis van een nulgebaseerde index.
4. **Add(T item)**: Hiermee voeg je een element aan het einde van de lijst toe.
5. **Remove(T item)**: Hiermee verwijder je het eerste exemplaar van een specifiek element uit de lijst.
6. **Insert(int index, T item)**: Hiermee voeg je een element op een specifieke index in de lijst in.
7. **RemoveAt(int index)**: Hiermee verwijder je een element op een specifieke index in de lijst.
8. **Contains(T item)**: Hiermee controleer je of een bepaald element in de lijst aanwezig is.
9. **IndexOf(T item)**: Hiermee vind je de index van het eerste exemplaar van een specifiek element in de lijst.

`IList<T>` is geïmplementeerd door verschillende klassen in C#, waaronder `List<T>`. Dit maakt het mogelijk om flexibel om te gaan met lijstgegevens in verschillende scenario's, of het nu gaat om eenvoudige lijsten van elementen of complexere gegevensstructuren.

Het gebruik van `IList<T>` kan handig zijn wanneer je gegevens wilt opslaan en manipuleren in een geïndexeerde lijst.

7.1. Klasse List<T>

> De `List<T>`-klasse is een veelgebruikte generieke klasse en vertegenwoordigt een dynamische lijst van objecten van een specifiek type `T`.

`List<T>` biedt een flexibele en efficiënte manier om gegevens op te slaan, toe te voegen, te verwijderen en te bewerken, en het is een van de meest gebruikte collectietypen vanwege zijn veelzijdigheid.

Hier zijn enkele belangrijke kenmerken en methoden van de `List<T>`-klasse:

1. **Dynamische grootte**: In tegenstelling tot arrays, kunnen `List<T>`-objecten dynamisch van grootte veranderen. Dit betekent dat je elementen kunt toevoegen of verwijderen zonder vooraf de grootte van de lijst te hoeven specificeren.
2. **Indexering**: Je kunt elementen in een `List<T>` benaderen door hun nulgebaseerde index te gebruiken. Bijvoorbeeld, `myList[0]` geeft toegang tot het eerste element.

3. **Add(T item)**: Hiermee voeg je een element aan het einde van de lijst toe.
4. **AddRange(IEnumerable<T> collection)**: Hiermee voeg je een hele verzameling elementen aan het einde van de lijst toe.
5. **Remove(T item)**: Hiermee verwijder je het eerste exemplaar van een specifiek element uit de lijst.
6. **RemoveAt(int index)**: Hiermee verwijder je een element op een specifieke index in de lijst.
7. **Insert(int index, T item)**: Hiermee voeg je een element op een specifieke index in de lijst in.
8. **Count**: Dit is een eigenschap die aangeeft hoeveel elementen er in de lijst zijn.
9. **Sort()**: Hiermee kun je de elementen in de lijst sorteren, mits het elementtype `T` 'Comparable' is (bijvoorbeeld, numerieke types of strings).
10. **Find() en FindAll()**: Hiermee kun je elementen in de lijst vinden op basis van een bepaalde voorwaarde.
11. **Clear()**: Hiermee verwijder je alle elementen uit de lijst.

`List<T>` is handig voor het opslaan van een variabel aantal gegevensitems en het biedt voordelen in vergelijking met arrays wanneer je gegevens wilt toevoegen of verwijderen, dus wanneer je niet op voorhand weet hoeveel elementen je nodig zult hebben.

Naast een parameterloze constructor kan je ook gebruik maken van een constructor waar je een `IEnumerable<T>` kan aan doorgeven. Deze constructor maakt een nieuwe `List<T>` en vult ze op met alle elementen uit de aangereikte `IEnumerable<T>`. Dergelijke constructor noemt men een **conversion constructor** en is beschikbaar in alle concrete klassen die we in dit hoofdstuk behandelen.

Het gebruik van de **conversion constructor** is handig want via deze constructor kan je dus **gelijk welk type collection uit het framework omvormen tot een ander type**.

```
internal class Part : IEquatable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "   Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
```

```
18          public override int GetHashCode()
19          {
20              return PartId;
21          }
22          public bool Equals(Part other)
23          {
24              if (other == null) return false;
25              return (this.PartId.Equals(other.PartId));
26          }
27          // Should also override == and != operators.
28      }
29      public class ListExample
30      {
31          public static void Main()
32          {
33              // Create a list of parts.
34              List<Part> parts = new List<Part>();
35
36              // Add parts to the list.
37              parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
38              parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
39              parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
40              parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
41              parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
42              parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });
43
44              // Write out the parts in the list. This will call the overridden
    ToString method
45              // in the Part class.
46              Console.WriteLine();
47              foreach (Part aPart in parts)
48              {
49                  Console.WriteLine(aPart);
50              }
51
52              // Check the list for part #1734. This calls the IEquatable.Equals
    method
53              // of the Part class, which checks the PartId for equality.
54              Console.WriteLine("\nContains(\"1734\"): {0}",
55              parts.Contains(new Part { PartId = 1734, PartName = "" }));
56
57              // Insert a new item at position 2.
58              Console.WriteLine("\nInsert(2, \"1834\")");
59              parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834
    });
60
61              //Console.WriteLine();
62              foreach (Part aPart in parts)
63              {
64                  Console.WriteLine(aPart);
65              }
```

```
66
67              Console.WriteLine("\nParts[3]: {0}", parts[3]);
68
69              Console.WriteLine("\nRemove(\"1534\")");
70
71              // This will remove part 1534 even though the PartName is different,
72              // because the Equals method only checks PartId for equality.
73              parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });
74
75              Console.WriteLine();
76              foreach (Part aPart in parts)
77              {
78                  Console.WriteLine(aPart);
79              }
80              Console.WriteLine("\nRemoveAt(3)");
81              // This will remove the part at index 3.
82              parts.RemoveAt(3);
83
84              Console.WriteLine();
85              foreach (Part aPart in parts)
86              {
87                  Console.WriteLine(aPart);
88              }
89
90              /*
91
92               ID: 1234   Name: crank arm
93               ID: 1334   Name: chain ring
94               ID: 1434   Name: regular seat
95               ID: 1444   Name: banana seat
96               ID: 1534   Name: cassette
97               ID: 1634   Name: shift lever
98
99               Contains("1734"): False
100
101              Insert(2, "1834")
102              ID: 1234   Name: crank arm
103              ID: 1334   Name: chain ring
104              ID: 1834   Name: brake lever
105              ID: 1434   Name: regular seat
106              ID: 1444   Name: banana seat
107              ID: 1534   Name: cassette
108              ID: 1634   Name: shift lever
109
110              Parts[3]: ID: 1434   Name: regular seat
111
112              Remove("1534")
113
114              ID: 1234   Name: crank arm
115              ID: 1334   Name: chain ring
116              ID: 1834   Name: brake lever
```

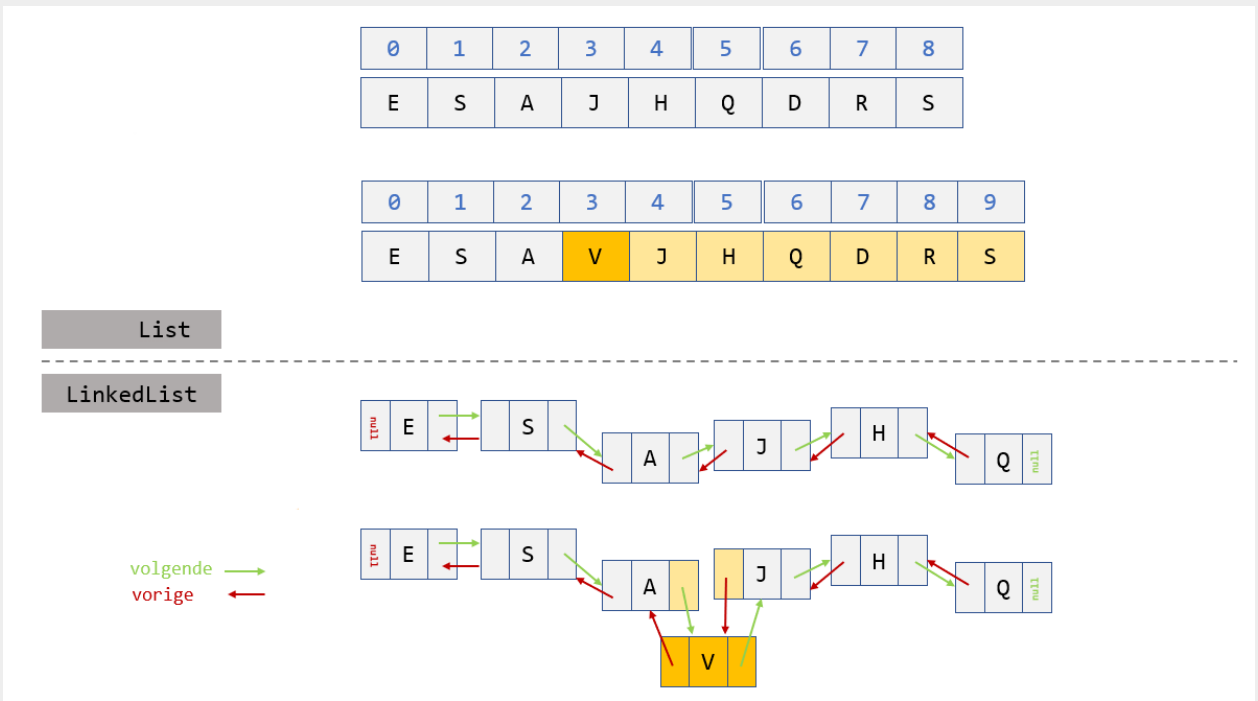
```
117             ID: 1434   Name: regular seat
118             ID: 1444   Name: banana seat
119             ID: 1634   Name: shift lever
120
121             RemoveAt(3)
122
123             ID: 1234   Name: crank arm
124             ID: 1334   Name: chain ring
125             ID: 1834   Name: brake lever
126             ID: 1444   Name: banana seat
127             ID: 1634   Name: shift lever
128
129
130          */
131         }
132     }
```

8. Klasse LinkedList<T>

Bij een `LinkedIList<T>` worden de elementen in de lijst gelinked via referenties.

Concreet heeft elk element in een **doubly-linked list** implementatie een referentie naar zijn voorganger, en naar zijn opvolger.

Let op: de `LinkedList<T>` klasse implementeert de interface `IList<T>` niet! Zo vermijdt men dat een gelinkte lijst kan geïndexeerd worden, aangezien dit zeer inefficiënt is.



- bij een insert in een array moeten meerder elementen verschoven worden.

- bij een insert in een doubly linked list moeten enkel een paar referenties aangepast worden.

In onderstaande tabel zijn enkele belangrijke verschillen tussen `List<T>` en `LinkedIList<T>` samengevat.

List<T>	**LinkedList<T>**
elementen in resizeable array	elementen in doubly linked list
constante toegangstijd voor elk element ⇒ random access	sequentiële toegang ⇒ starten vanaf eerste element
invoegen of verwijderen van element ⇒ veel verschuivingen	efficiënt toevoegen en/of verwijderen van element
bij voorkeur te gebruiken bij veel **opzoekingen**	bij voorkeur te gebruiken bij **veel invoegen/verwijderen** van elementen

```
1   public class LinkedListExample
2   {
3       public static void Main()
4       {
5           // Create the link list.
6           string[] words =
7               { "the", "fox", "jumps", "over", "the", "dog" };
8           LinkedList<string> sentence = new LinkedList<string>(words);
9
10          Display(sentence, "The linked list values:");
11
12          Console.WriteLine("sentence.Contains(\"jumps\") = {0}",
13              sentence.Contains("jumps"));
14
15          // Add the word 'today' to the beginning of the linked list.
16          sentence.AddFirst("today");
17          Display(sentence, "Test 1: Add 'today' to beginning of the list:");
18
19          // Move the first node to be the last node.
20          LinkedListNode<string> mark1 = sentence.First;
21          sentence.RemoveFirst();
22          sentence.AddLast(mark1);
23          Display(sentence, "Test 2: Move first node to be last node:");
24
25          // Change the last node to 'yesterday'.
26          sentence.RemoveLast();
27          sentence.AddLast("yesterday");
28          Display(sentence, "Test 3: Change the last node to 'yesterday':");
29
30          // Move the last node to be the first node.
31          mark1 = sentence.Last;
32          sentence.RemoveLast();
33          sentence.AddFirst(mark1);
```

```
34              Display(sentence, "Test 4: Move last node to be first node:");
35
36              // Indicate the last occurence of 'the'.
37              sentence.RemoveFirst();
38              LinkedListNode<string> current = sentence.FindLast("the");
39              IndicateNode(current, "Test 5: Indicate last occurence of 'the':");
40
41              // Add 'lazy' and 'old' after 'the' (the LinkedListNode named
    current).
42              sentence.AddAfter(current, "old");
43              sentence.AddAfter(current, "lazy");
44              IndicateNode(current, "Test 6: Add 'lazy' and 'old' after 'the':");
45
46              // Indicate 'fox' node.
47              current = sentence.Find("fox");
48              IndicateNode(current, "Test 7: Indicate the 'fox' node:");
49
50              // Add 'quick' and 'brown' before 'fox':
51              sentence.AddBefore(current, "quick");
52              sentence.AddBefore(current, "brown");
53              IndicateNode(current, "Test 8: Add 'quick' and 'brown' before
    'fox':");
54
55              // Keep a reference to the current node, 'fox',
56              // and to the previous node in the list. Indicate the 'dog' node.
57              mark1 = current;
58              LinkedListNode<string> mark2 = current.Previous;
59              current = sentence.Find("dog");
60              IndicateNode(current, "Test 9: Indicate the 'dog' node:");
61
62              // The AddBefore method throws an InvalidOperationException
63              // if you try to add a node that already belongs to a list.
64              Console.WriteLine("Test 10: Throw exception by adding node (fox)
    already in the list:");
65              try
66              {
67                  sentence.AddBefore(current, mark1);
68              }
69              catch (InvalidOperationException ex)
70              {
71                  Console.WriteLine("Exception message: {0}", ex.Message);
72              }
73              Console.WriteLine();
74
75              // Remove the node referred to by mark1, and then add it
76              // before the node referred to by current.
77              // Indicate the node referred to by current.
78              sentence.Remove(mark1);
79              sentence.AddBefore(current, mark1);
80              IndicateNode(current, "Test 11: Move a referenced node (fox) before
    the current node (dog):");
```

```
81
82              // Remove the node referred to by current.
83              sentence.Remove(current);
84              IndicateNode(current, "Test 12: Remove current node (dog) and attempt
    to indicate it:");
85
86              // Add the node after the node referred to by mark2.
87              sentence.AddAfter(mark2, current);
88              IndicateNode(current, "Test 13: Add node removed in test 11 after a
    referenced node (brown):");
89
90              // The Remove method finds and removes the
91              // first node that that has the specified value.
92              sentence.Remove("old");
93              Display(sentence, "Test 14: Remove node that has the value 'old':");
94
95              // When the linked list is cast to ICollection(Of String),
96              // the Add method adds a node to the end of the list.
97              sentence.RemoveLast();
98              ICollection<string> icoll = sentence;
99              icoll.Add("rhinoceros");
100             Display(sentence, "Test 15: Remove last node, cast to ICollection, and
    add 'rhinoceros':");
101
102             Console.WriteLine("Test 16: Copy the list to an array:");
103             // Create an array with the same number of
104             // elements as the linked list.
105             string[] sArray = new string[sentence.Count];
106             sentence.CopyTo(sArray, 0);
107
108             foreach (string s in sArray)
109             {
110                 Console.WriteLine(s);
111             }
112
113             // Release all the nodes.
114             sentence.Clear();
115
116             Console.WriteLine();
117             Console.WriteLine("Test 17: Clear linked list. Contains 'jumps' =
    {0}",
118                 sentence.Contains("jumps"));
119
120             Console.ReadLine();
121         }
122
123         private static void Display(LinkedList<string> words, string test)
124         {
125             Console.WriteLine(test);
126             foreach (string word in words)
127             {
```

```
128                 Console.Write(word + " ");
129             }
130             Console.WriteLine();
131             Console.WriteLine();
132         }
133 
134         private static void IndicateNode(LinkedListNode<string> node, string test)
135         {
136             Console.WriteLine(test);
137             if (node.List == null)
138             {
139                 Console.WriteLine("Node '{0}' is not in the list.\n",
140                     node.Value);
141                 return;
142             }
143 
144             StringBuilder result = new StringBuilder("(" + node.Value + ")");
145             LinkedListNode<string> nodeP = node.Previous;
146 
147             while (nodeP != null)
148             {
149                 result.Insert(0, nodeP.Value + " ");
150                 nodeP = nodeP.Previous;
151             }
152 
153             node = node.Next;
154             while (node != null)
155             {
156                 result.Append(" " + node.Value);
157                 node = node.Next;
158             }
159 
160             Console.WriteLine(result);
161             Console.WriteLine();
162         }
163 
164     }
165 
166     //This code example produces the following output:
167     //
168     //The linked list values:
169     //the fox jumps over the dog
170 
171     //Test 1: Add 'today' to beginning of the list:
172     //today the fox jumps over the dog
173 
174     //Test 2: Move first node to be last node:
175     //the fox jumps over the dog today
176 
177     //Test 3: Change the last node to 'yesterday':
178     //the fox jumps over the dog yesterday
```

```
179
180     //Test 4: Move last node to be first node:
181     //yesterday the fox jumps over the dog
182
183     //Test 5: Indicate last occurence of 'the':
184     //the fox jumps over (the) dog
185
186     //Test 6: Add 'lazy' and 'old' after 'the':
187     //the fox jumps over (the) lazy old dog
188
189     //Test 7: Indicate the 'fox' node:
190     //the (fox) jumps over the lazy old dog
191
192     //Test 8: Add 'quick' and 'brown' before 'fox':
193     //the quick brown (fox) jumps over the lazy old dog
194
195     //Test 9: Indicate the 'dog' node:
196     //the quick brown fox jumps over the lazy old (dog)
197
198     //Test 10: Throw exception by adding node (fox) already in the list:
199     //Exception message: The LinkedList node belongs a LinkedList.
200
201     //Test 11: Move a referenced node (fox) before the current node (dog):
202     //the quick brown jumps over the lazy old fox (dog)
203
204     //Test 12: Remove current node (dog) and attempt to indicate it:
205     //Node 'dog' is not in the list.
206
207     //Test 13: Add node removed in test 11 after a referenced node (brown):
208     //the quick brown (dog) jumps over the lazy old fox
209
210     //Test 14: Remove node that has the value 'old':
211     //the quick brown dog jumps over the lazy fox
212
213     //Test 15: Remove last node, cast to ICollection, and add 'rhinoceros':
214     //the quick brown dog jumps over the lazy rhinoceros
215
216     //Test 16: Copy the list to an array:
217     //the
218     //quick
219     //brown
220     //dog
221     //jumps
222     //over
223     //the
224     //lazy
225     //rhinoceros
226
227     //Test 17: Clear linked list. Contains 'jumps' = False
228     //
```

9. Interface ISet<T>

De `ISet<T>`-interface vertegenwoordigt een verzameling unieke elementen zonder dubbele waarden van het generieke type `T`. Met andere woorden, een set is een verzameling waarin elk element uniek is, en het maakt geen onderscheid tussen de volgorde van elementen.

We benadrukken nogmaals dat ook de gelijkheid binnen een `ISet<T>` gebaseerd is op de **Equals** -methode. Mogelijks kunnen elementen in een set wel geordend zijn, maar dit is niet altijd het geval zoals bij een `IList<T>`. Dit is de reden waarom indexering zoals bij een `IList<T>` niet mogelijk is.

Een set bevat nooit twee elementen e1 en e2 waarvoor e1.Equals(e2) true is.

`ISet<T>` biedt een reeks methoden om elementen aan de set toe te voegen, te verwijderen, te controleren op de aanwezigheid van elementen en andere set-operaties uit te voeren. Het is nuttig wanneer je unieke elementen wilt opslaan en efficiënt wilt werken met deze elementen zonder duplicaten.

Hier zijn enkele belangrijke methoden en eigenschappen die worden geërfd van de `ISet<T>` -interface:

1. **Add(T item)**: Voegt een element toe aan de set. Als het element al in de set aanwezig is, wordt het niet opnieuw toegevoegd.
2. **Remove(T item)**: Verwijdert een element uit de set als het aanwezig is.
3. **Contains(T item)**: Controleert of een bepaald element in de set aanwezig is.
4. **Count**: Dit is een eigenschap die aangeeft hoeveel elementen er in de set zijn.
5. **Clear()**: Verwijdert alle elementen uit de set.
6. **UnionWith(IEnumerable<T> other)**: Voert een unie-operatie uit met een andere verzameling en voegt alle unieke elementen samen in de huidige set.
7. **IntersectWith(IEnumerable<T> other)**: Voert een intersectie-operatie uit met een andere verzameling en behoudt alleen de elementen die in beide verzamelingen voorkomen.

9.1. Klasse HashSet<T>

`HashSet<T>` is een concrete implementatie van `Set<T>` die een hashtabel als onderliggende structuur heeft en waarbij elementen niet geordend zijn. Dieper ingaan op deze structuur valt buiten de scope van deze cursus maar het is handig om weten dat in dergelijke structuur de **hashcode** van een object wordt gebruikt als een index in een tabel.

Werken met een `HashSet<T>` betekent concreet dat elementen enorm snel te vinden zijn en dat ook het toevoegen en verwijderen van elementen heel performant kan gebeuren.

Wat betreft het enumereren over de elementen van de verzameling zullen andere collections dikwijls een betere performantie geven. Daar de elementen niet geordend zijn kan je trouwens ook

niet voorspellen in welke volgorde de elementen geretourneerd worden wanneer je over een HashSet itereert.

```
        public static void Main(string[] args)
        {
            HashSet<int> evenNumbers = new HashSet<int>();
            HashSet<int> oddNumbers = new HashSet<int>();

            for (int i = 0; i < 5; i++)
            {
                // Populate numbers with just even numbers.
                evenNumbers.Add(i * 2);

                // Populate oddNumbers with just odd numbers.
                oddNumbers.Add((i * 2) + 1);
            }

            Console.Write("evenNumbers contains {0} elements: ",
evenNumbers.Count);
            DisplaySet(evenNumbers);

            Console.Write("oddNumbers contains {0} elements: ", oddNumbers.Count);
            DisplaySet(oddNumbers);

            // Create a new HashSet populated with even numbers.
            HashSet<int> numbers = new HashSet<int>(evenNumbers);
            Console.WriteLine("numbers UnionWith oddNumbers...");
            numbers.UnionWith(oddNumbers);

            Console.Write("numbers contains {0} elements: ", numbers.Count);
            DisplaySet(numbers);

            void DisplaySet(HashSet<int> collection)
            {
                Console.Write("{");
                foreach (int i in collection)
                {
                    Console.Write(" {0}", i);
                }
                Console.WriteLine(" }");
            }

            /* This example produces output similar to the following:
            * evenNumbers contains 5 elements: { 0 2 4 6 8 }
            * oddNumbers contains 5 elements: { 1 3 5 7 9 }
            * numbers UnionWith oddNumbers...
            * numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
            */
        }
```

9.2. Klasse SortedSet<T>

Zoals uit de naam af te leiden valt is een `SortedSet<T>` een set met een **sortering van de elementen**. De sortering van de elementen kan op natuurlijke (`IComparable<T>>`) of op absolute wijze (`IComparer<T>`) zijn. Het ordenen van objecten werd uitvoerig behandeld in het hoofdstuk "Polymorfisme en Interfaces".

Door de ordening van de elementen wordt het itereren over een `SortedSet<T>` voorspelbaar: elementen worden gesorteerd geretourneerd door de enumerator.

- natuurlijk ordening maakt gebruik van de **CompareTo**-methode
  - *de elementen in de set moeten* `IComparable<T>` *implementeren*
  - _deze ordening wordt gebruikt indien je geen comparator specifieert tijdens constructie van de `SortedSet<T>`
- totale ordening maakt gebruik van een **IComparer<T>**
  - *er moet een instantie van een comparer klasse die* `IComparer<T>` *implementeert voorzien worden*
  - *de comparer kan aan de constructor van concrete* `SortedISet<T>` *implementaties meegegeven worden*

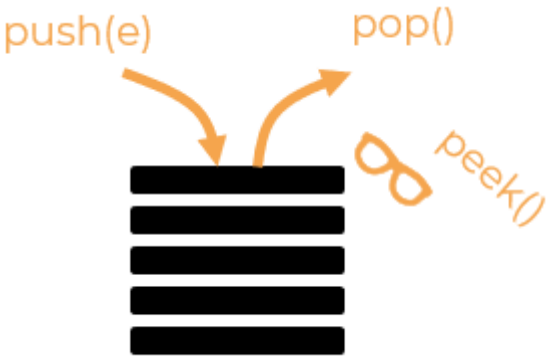
Het feit dat de elementen gesorteerd in deze verzameling zitten geeft dan ook aanleiding tot meer specifieke functionaliteit die van de volgorde van de elementen gebruik maakt bv. Max, Min, Reverse ...

```
1     public class SortedSetExample
2     {
3         private static readonly string[] _Names = new string[] {"yellow", "green",
  "black", "tan", "grey", "white", "orange", "red", "green"};
4
5         public SortedSetExample()
6         {
7             SortedSet<string> tree = new SortedSet<string>(_Names);
8             Console.WriteLine("sorted set: ");
9             PrintSet(tree);
10
11            // alle elementen die < zijn dan element "orange"
12            Console.Write("\nView between \"a\" and \"m\":  ");
13            PrintSet(tree.GetViewBetween("a","m"));
14
15            // het eerste en het laatste element
16            Console.Write("first: {0}\n", tree.Min);
17            Console.Write("last : {0}\n", tree.Max);
18        }
19
20
21        private void PrintSet(SortedSet<string> set)
```

```
22          {
23              foreach (string s in set)
24              {
25                  Console.Write("{0} ", s);
26              }
27              Console.WriteLine();
28          }
29
30          public static void Main(string[] args)
31          {
32              new SortedSetExample();
33          }
34
35      }
```

10. Klasse Stack<T>

De `Stack<T>`-klasse vertegenwoordigt een LIFO (Last-In, First-Out) gegevensstructuur, waarbij elementen worden toegevoegd en verwijderd volgens het principe van "last in, first out.". Met andere woorden, het laatste element dat aan de stapel wordt toegevoegd, is het eerste element dat wordt verwijderd. Een stack is vergelijkbaar met een stapel borden waarop je nieuwe borden bovenop plaatst en de bovenste plaat eerst verwijdert.



Hier zijn enkele belangrijke eigenschappen en methoden van de `Stack<T>`-klasse:

1. **Push(T item)**: Hiermee voeg je een element toe aan de bovenkant van de stapel.
2. **Pop()**: Hiermee verwijder je en retourneer je het bovenste element van de stapel.
3. **Peek()**: Hiermee bekijk je het bovenste element van de stapel zonder het te verwijderen.
4. **Count**: Dit is een eigenschap die aangeeft hoeveel elementen er momenteel in de stapel zitten.
5. **Clear()**: Hiermee verwijder je alle elementen uit de stapel.

`Stack<T>` is handig in situaties waarin je gegevens moet bijhouden in de volgorde waarin ze zijn toegevoegd, en waarbij je eerst de meest recente gegevens wilt verwerken voordat je teruggaat naar eerdere gegevens. Het wordt vaak gebruikt voor het bijhouden van function calls in een LIFO-oproepstack of voor het implementeren van algoritmen.

Merk op dat bij het gebruik van een stack het belangrijk is om ervoor te zorgen dat je elementen in de juiste volgorde plaatst en verwijdert, aangezien het LIFO-principe wordt gehandhaafd.

```
1     public class StackExample
2     {
3         public static void Main()
4         {
5             Stack<string> numbers = new Stack<string>();
6             numbers.Push("one");
7             numbers.Push("two");
8             numbers.Push("three");
9             numbers.Push("four");
10            numbers.Push("five");
11
12            // A stack can be enumerated without disturbing its contents.
13            foreach (string number in numbers)
14            {
15                Console.WriteLine(number);
16            }
17
18            Console.WriteLine("\nPopping '{0}'", numbers.Pop());
19            Console.WriteLine("Peek at next item to destack: {0}",
20                numbers.Peek());
21            Console.WriteLine("Popping '{0}'", numbers.Pop());
22
23            // Create a copy of the stack, using the ToArray method and the
24            // constructor that accepts an IEnumerable<T>.
25            Stack<string> stack2 = new Stack<string>(numbers.ToArray());
26
27            Console.WriteLine("\nContents of the first copy:");
28            foreach (string number in stack2)
29            {
30                Console.WriteLine(number);
31            }
32
33            // Create an array twice the size of the stack and copy the
34            // elements of the stack, starting at the middle of the
35            // array.
36            string[] array2 = new string[numbers.Count * 2];
37            numbers.CopyTo(array2, numbers.Count);
38
39            // Create a second stack, using the constructor that accepts an
40            // IEnumerable(Of T).
41            Stack<string> stack3 = new Stack<string>(array2);
42
43            Console.WriteLine("\nContents of the second copy, with duplicates and
  nulls:");
44            foreach (string number in stack3)
45            {
```

```
                Console.WriteLine(number);
            }

            Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
                stack2.Contains("four"));

            Console.WriteLine("\nstack2.Clear()");
            stack2.Clear();
            Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
        }
    }

    /* This code example produces the following output:

    five
    four
    three
    two
    one

    Popping 'five'
    Peek at next item to destack: four
    Popping 'four'

    Contents of the first copy:
    one
    two
    three

    Contents of the second copy, with duplicates and nulls:
    one
    two
    three




    stack2.Contains("four") = False

    stack2.Clear()

    stack2.Count = 0
     */
```

11. Klasse Queue<T>

De Queue<T>-klasse vertegenwoordigt een FIFO (First-In, First-Out) gegevensstructuur, waarin elementen worden toegevoegd aan het einde van de wachtrij en worden verwijderd van het begin van de wachtrij. Met andere woorden, het eerste element dat aan de wachtrij wordt toegevoegd, is

het eerste element dat wordt verwijderd, vergelijkbaar met een wachtrij in het dagelijks leven.

Hier zijn enkele belangrijke eigenschappen en methoden van de `Queue<T>`-klasse:

1. **Enqueue(T item)**: Hiermee voeg je een element toe aan het einde van de wachtrij.
2. **Dequeue()**: Hiermee verwijder je en retourneer je het eerste element van de wachtrij.
3. **Peek()**: Hiermee bekijk je het eerste element van de wachtrij zonder het te verwijderen.
4. **Count**: Dit is een eigenschap die aangeeft hoeveel elementen er momenteel in de wachtrij zitten.
5. **Clear()**: Hiermee verwijder je alle elementen uit de wachtrij.

`Queue<T>` is handig in situaties waarin je gegevens moet verwerken in de volgorde waarin ze zijn toegevoegd, en waarbij je eerst de oudste gegevens wilt verwerken voordat je doorgaat naar nieuwere gegevens. Het wordt vaak gebruikt voor het beheren van taken die moeten worden uitgevoerd in de volgorde waarin ze zijn ontvangen, zoals het verwerken van berichten in een berichtensysteem of het beheren van taken in een wachtrij.

Bij het gebruik van een wachtrij is het belangrijk om de volgorde van toevoeging en verwijdering te respecteren, aangezien het FIFO-principe wordt gehandhaafd. Het eerste element dat wordt toegevoegd, is het eerste dat wordt verwijderd.

```
class QueueExample
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");

```

```
12              // A queue can be enumerated without disturbing its contents.
13              foreach (string number in numbers)
14              {
15                  Console.WriteLine(number);
16              }
17
18              Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
19              Console.WriteLine("Peek at next item to dequeue: {0}",
20                  numbers.Peek());
21              Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());
22
23              // Create a copy of the queue, using the ToArray method and the
24              // constructor that accepts an IEnumerable<T>.
25              Queue<string> queueCopy = new Queue<string>(numbers.ToArray());
26
27              Console.WriteLine("\nContents of the first copy:");
28              foreach (string number in queueCopy)
29              {
30                  Console.WriteLine(number);
31              }
32
33              // Create an array twice the size of the queue and copy the
34              // elements of the queue, starting at the middle of the
35              // array.
36              string[] array2 = new string[numbers.Count * 2];
37              numbers.CopyTo(array2, numbers.Count);
38
39              // Create a second queue, using the constructor that accepts an
40              // IEnumerable(Of T).
41              Queue<string> queueCopy2 = new Queue<string>(array2);
42
43              Console.WriteLine("\nContents of the second copy, with duplicates and
    nulls:");
44              foreach (string number in queueCopy2)
45              {
46                  Console.WriteLine(number);
47              }
48
49              Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
50                  queueCopy.Contains("four"));
51
52              Console.WriteLine("\nqueueCopy.Clear()");
53              queueCopy.Clear();
54              Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
55          }
56      }
57
58      /* This code example produces the following output:
59
60      one
61      two
```

```
62      three
63      four
64      five
65
66      Dequeuing 'one'
67      Peek at next item to dequeue: two
68      Dequeuing 'two'
69
70      Contents of the copy:
71      three
72      four
73      five
74
75      Contents of the second copy, with duplicates and nulls:
76
77
78
79      three
80      four
81      five
82
83      queueCopy.Contains("four") = True
84
85      queueCopy.Clear()
86
87      queueCopy.Count = 0
88       */
```

12. Interface IDictionary<TKey, TValue>

De `IDictionary<TKey, TValue>`-interface vertegenwoordigt een generieke collectie van sleutel-waardeparen, waarbij elke sleutel uniek is en wordt gekoppeld aan een specifieke waarde van het type TValue. Dit maakt `IDictionary<TKey, TValue>` geschikt voor het opslaan en beheren van associatieve gegevens, waarbij je gegevens kunt ophalen en wijzigen op basis van een sleutel.

Het uniek zijn van sleutels wordt bepaald aan de hand van de `Equals`-methode.

Hier zijn enkele belangrijke methoden en eigenschappen van de `IDictionary<TKey, TValue>` -interface:

1. **Add(TKey key, TValue value)**: Voegt een nieuw sleutel-waardepaar toe aan de dictionary. Als de sleutel al bestaat, resulteert dit in een uitzondering.
2. **Remove(TKey key)**: Verwijdert het sleutel-waardepaar met de opgegeven sleutel uit de dictionary.
3. **TryGetValue(TKey key, out TValue value)**: Probeert de waarde op te halen die is gekoppeld aan de opgegeven sleutel. Als de sleutel niet in de dictionary aanwezig is, wordt value op default(TValue) ingesteld en retourneert de methode false.

4. **this[TKey key]**: Dit is de indexeringsmethode waarmee je de waarde kunt ophalen of wijzigen die is gekoppeld aan een specifieke sleutel.
5. **Keys**: Dit is een eigenschap die een verzameling van alle sleutels in de dictionary retourneert.
6. **Values**: Dit is een eigenschap die een verzameling van alle waarden in de dictionary retourneert.
7. **ContainsKey(TKey key)**: Controleert of de opgegeven sleutel in de dictionary aanwezig is.
8. **ContainsValue(TValue value)**: Controleert of de opgegeven waarde in de dictionary aanwezig is.
9. **Count**: Dit is een eigenschap die aangeeft hoeveel sleutel-waardeparen er in de dictionary zijn.

Om met een `foreach` statement door de elementen te lopen maak je gebruik van het `KeyValuePair<TKey, TValue>` type.

```
1     foreach (KeyValuePair<int, string> kvp in myDictionary)
2     {
3         Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
4     }
```

12.1. Klasse Dictionary<TKey, TValue>

`IDictionary<TKey, TValue>` is vaak geïmplementeerd door de `Dictionary<TKey, TValue>`-klasse, die een efficiënte en veelgebruikte implementatie is voor een dictionary.

Hier is een voorbeeld:

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

scores["Alice"] = 95;   // Voeg een sleutel-waardepaar toe
scores["Bob"] = 87;
scores["Charlie"] = 78;

int bobScore = scores["Bob"];  // Haal de waarde op voor de sleutel "Bob"

if (scores.ContainsKey("Alice")) // Controleer of een sleutel aanwezig is
{
    scores["Alice"] = 100;   // Werk de waarde voor de sleutel "Alice" bij
}

foreach (var key in scores.Keys)
{
    Console.WriteLine($"{key}: {scores[key]}");
}
```

Dit voorbeeld toont het gebruik van `IDictionary<TKey, TValue>` om een dictionary van scores te beheren met namen als sleutels en numerieke scores als waarden. Je kunt sleutels toevoegen,

waarden ophalen en bijwerken, en ook controleren of een bepaalde sleutel in de dictionary aanwezig is.

```
    public class DictionaryExample
    {
        public static void Main()
        {
            // Create a new dictionary of strings, with string keys,
            // and access it through the IDictionary generic interface.
            IDictionary<string, string> openWith =
                new Dictionary<string, string>();

            // Add some elements to the dictionary. There are no
            // duplicate keys, but some of the values are duplicates.
            openWith.Add("txt", "notepad.exe");
            openWith.Add("bmp", "paint.exe");
            openWith.Add("dib", "paint.exe");
            openWith.Add("rtf", "wordpad.exe");

            // The Add method throws an exception if the new key is
            // already in the dictionary.
            try
            {
                openWith.Add("txt", "winword.exe");
            }
            catch (ArgumentException)
            {
                Console.WriteLine("An element with Key = \"txt\" already
exists.");
            }

            // The Item property is another name for the indexer, so you
            // can omit its name when accessing elements.
            Console.WriteLine("For key = \"rtf\", value = {0}.",
                openWith["rtf"]);

            // The indexer can be used to change the value associated
            // with a key.
            openWith["rtf"] = "winword.exe";
            Console.WriteLine("For key = \"rtf\", value = {0}.",
                openWith["rtf"]);

            // If a key does not exist, setting the indexer for that key
            // adds a new key/value pair.
            openWith["doc"] = "winword.exe";

            // The indexer throws an exception if the requested key is
            // not in the dictionary.
            try
            {
```

```
            Console.WriteLine("For key = \"tif\", value = {0}.",
                openWith["tif"]);
        }
        catch (KeyNotFoundException)
        {
            Console.WriteLine("Key = \"tif\" is not found.");
        }

        // When a program often has to try keys that turn out not to
        // be in the dictionary, TryGetValue can be a more efficient
        // way to retrieve values.
        string value = "";
        if (openWith.TryGetValue("tif", out value))
        {
            Console.WriteLine("For key = \"tif\", value = {0}.", value);
        }
        else
        {
            Console.WriteLine("Key = \"tif\" is not found.");
        }

        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
                openWith["ht"]);
        }

        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach (KeyValuePair<string, string> kvp in openWith)
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }

        // To get the values alone, use the Values property.
        ICollection<string> icoll = openWith.Values;

        // The elements of the ValueCollection are strongly typed
        // with the type that was specified for dictionary values.
        Console.WriteLine();
        foreach (string s in icoll)
        {
            Console.WriteLine("Value = {0}", s);
        }

        // To get the keys alone, use the Keys property.
```

```
            icoll = openWith.Keys;

            // The elements of the ValueCollection are strongly typed
            // with the type that was specified for dictionary values.
            Console.WriteLine();
            foreach (string s in icoll)
            {
                Console.WriteLine("Key = {0}", s);
            }

            // Use the Remove method to remove a key/value pair.
            Console.WriteLine("\nRemove(\"doc\")");
            openWith.Remove("doc");

            if (!openWith.ContainsKey("doc"))
            {
                Console.WriteLine("Key \"doc\" is not found.");
            }
        }
    }
```

13. Interface IReadOnlyCollection<T>

De IReadOnlyCollection<T>-interface is een subinterface van IEnumerable<T> en wordt gebruikt om toegang te bieden tot een verzameling van items van het generieke type T zonder de mogelijkheid om elementen toe te voegen, te verwijderen of te wijzigen. Met andere woorden, een implementatie van IReadOnlyCollection<T> biedt alleen-lezen toegang tot een verzameling en voorkomt elke wijziging van de gegevens.

IReadOnlyCollection<T> is nuttig wanneer je een verzameling van gegevens wilt blootstellen aan andere delen van je code, maar je wilt voorkomen dat deze code de verzameling kan wijzigen. Dit kan handig zijn om gegevens alleen-lezen te delen, wat de veiligheid en consistentie van je code kan verbeteren.

Hier is een eenvoudig voorbeeld van hoe je IReadOnlyCollection<T> kunt gebruiken:

```
public class MyReadOnlyCollection<T> : IReadOnlyCollection<T>
{
    private List<T> data;

    public MyReadOnlyCollection(IEnumerable<T> source)
    {
        data = new List<T>(source);
    }

    public int Count => data.Count;

    public IEnumerator<T> GetEnumerator()
```

```
    {
        return data.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

// Gebruik van de MyReadOnlyCollection
var sourceData = new List<int> { 1, 2, 3, 4, 5 };
var readOnlyCollection = new MyReadOnlyCollection<int>(sourceData);

// Proberen elementen toe te voegen, verwijderen of wijzigen zal een compileerfout
genereren
// readOnlyCollection.Add(6); // Compileerfout
// readOnlyCollection.Remove(3); // Compileerfout

// Itereren over de gegevens is toegestaan
foreach (var item in readOnlyCollection)
{
    Console.WriteLine(item);
}
```

In dit voorbeeld wordt `MyReadOnlyCollection<T>` geïmplementeerd om `IReadOnlyCollection<T>` te gebruiken om een verzameling gegevens alleen-lezen te maken en toegang te bieden tot de gegevens zonder ze te wijzigen. Het voorkomt dat elementen worden toegevoegd, verwijderd of gewijzigd, maar staat iteratie over de gegevens toe.