

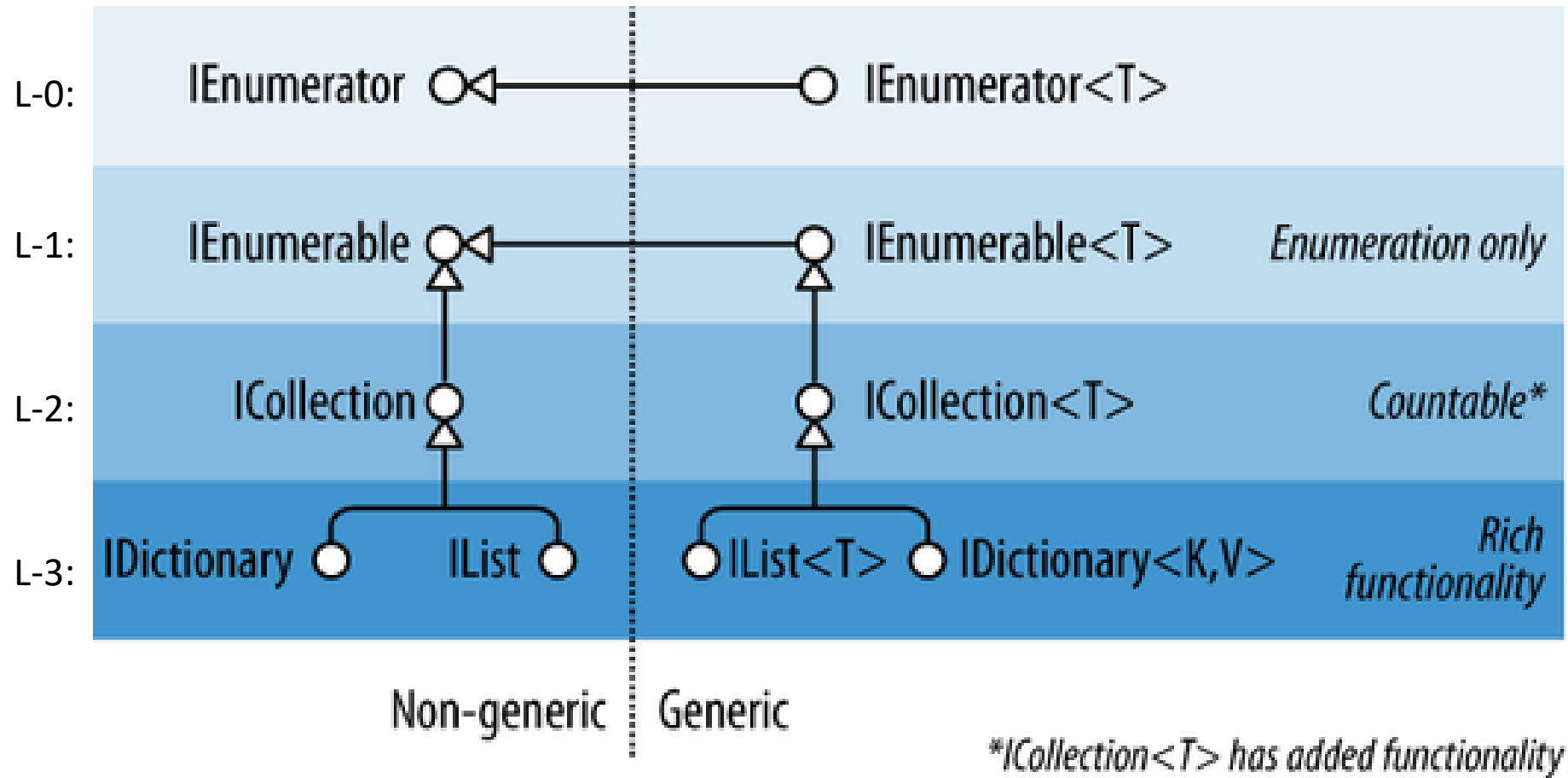
Collections og LINQ

Thomas Bøgholm

Indbyggede datastrukturer

- .NET frameworket, *som andre oopsprog*, indeholder en række standard datastrukturer (collections / containere) til forskellige formål:
 - Lister med variabel størrelse
 - Kædede lister
 - Sorterede lister, køer, stakke, dictionaries, mængder
- Typerne er opdelt i:
 - Interfaces der definerer standard operationer for datastrukturer
 - Prædefinerede klasser der implementerer interfaces

Collection Interfaces



Niveau 0: IEnumerator

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

```
public interface IEnumerator<out T>
    : IEnumerator,
    IDisposable
{
    T Current { get; }
}
```

```
class MyPersonEnumerator : IEnumerator<Person>
{
    private Person[] _people;
    private int position = -1;
    public MyPersonEnumerator(Person[] array)
    {
        _people = list;
    }
    ...
}
```

Niveau 0: IEnumerator

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<out T>
    : IEnumerator,
    IDisposable
{
    T Current { get; }
}
```

```
class MyPersonEnumerator : IEnumerator<Person>
{
    *snip* Constructor og dispose *snip*
    private Person[] _people;
    private int position = -1;
    object IEnumerator.Current => Current;
    public Person Current {
        get {
            try {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
    public bool MoveNext()
    {
        return (++position < _people.Length);
    }
    public void Reset()
    {
        position = -1;
    }
}
```

Brugen af enumerator

```
Person[] personArray = new Person[] {  
    new Person("Thomas"),  
    new Person("Hans"),  
    new Person("Bent"),  
    new Person("Lone"),  
    new Person("Kurt")  
};  
MyPersonEnumerator mpe = new  
MyPersonEnumerator(personArray);  
  
while (mpe.MoveNext())  
{  
    Console.WriteLine(mpe.Current.Name);  
}
```



Thomas
Hans
Bent
Lone
Kurt

Niveau 1: IEnumerable

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
public interface IEnumerable<T>
    : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Niveau 1: IEnumerable

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerable<T>
    : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

```
class PersonCatalog : IEnumerable<Person>
{
    private Person[] _content;
    public PersonCatalog(Person[] content)
    {
        this._content = content;
    }
    public IEnumerator<Person> GetEnumerator()
    {
        return new MyPersonEnumerator(_content);
    }

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


Eksempel på brug af klasser der implementerer IEnumerable

```
PersonCatalog pc = new PersonCatalog(personArray);
```

```
IEnumerator<Person> enumerator = pc.GetEnumerator();  
while (enumerator.MoveNext())  
{  
    Console.WriteLine(enumerator.Current.Name);  
}
```



Thomas
Hans
Bent
Lone
Kurt

Eksempel på brug af klasser der implementerer IEnumerable

```
PersonCatalog pc = new PersonCatalog(personArray);
```

```
IEnumerator<Person> enumerator = pc.GetEnumerator();  
while (enumerator.MoveNext())  
{  
    Console.WriteLine(enumerator.Current.Name);  
}
```

```
foreach (var item in pc)  
{  
    Console.WriteLine(item.Name);  
}
```



Thomas
Hans
Bent
Lone
Kurt

Niveau 1 og 0: IEnumerable<T> og IEnumerator<T>

- Implementation af IEnumerable<T> betyder at en datastruktur kan gennemløbes (enumereres)
 - IEnumerable har én metode GetEnumerator() der returnerer en enumerator der bruges til at gennemløbe elementerne i data strukturen.
 - Er oftest nok!
- En enumerator implementerer IEnumerator<T>
 - MoveNext, Current, Reset (Dispose)

Niveau 2: ICollection<T>

- `public interface ICollection<T> : IEnumerable<T>, IEnumerable`

Properties:

	Name	Description
	Count	Gets the number of elements contained in the ICollection<T>.
	IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.

Metoder:

	Name	Description
	Add	Adds an item to the ICollection<T>.
	Clear	Removes all items from the ICollection<T>.
	Contains	Determines whether the ICollection<T> contains a specific value.
	CopyTo	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index.
	Remove	Removes the first occurrence of a specific object from the ICollection<T>.


ICollection<T> eksempel

- Add() giver os collection initializer faciliteter

```
static void Main(string[] args)
{
    ICollection<int> tal = new List<int>() { 1, 2, 3, 4 };

    tal.Add(5);

    Console.WriteLine(tal.Count);
    tal.Remove(2);
    Console.WriteLine(tal.Contains(2));
    Console.WriteLine(tal.IsReadOnly);
    int[] smallArray = { 6, 7, 8 };
    int[] bigArray = { 6, 7, 8, 9, 10 };
    tal.CopyTo(bigArray, 1);
    tal.CopyTo(smallArray, 0);
    tal.Clear();
}
```



```
// { 1, 2, 3, 4, 5 }
// 5
// { 1, 3, 4, 5 }
// false
// false
// ved kopi: array skal være stort nok
// bigArray = { 6, 1, 3, 4, 5 }
// ArgumentException.
// { }
```

Niveau 3a: IList<T>

- IList<T> er standard interface for collections der kan *indekseres* ved position.
- Elementer kan tilgås via **indexer**.

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this[int index] { get; set; } //indexer
    int IndexOf(T item); //Lineær søgning efter 'Item'. -1 hvis ej fundet.
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

```
static void Main(string[] args)
{
    IList<int> tal = new List<int>() { 1, 2, 3, 4, 5, 6 };

    tal[2] = tal[2] * 3;
    tal.IndexOf(4);
    tal.IndexOf(7);
    tal.Insert(0, 8);
    tal.RemoveAt(4);
}
```

```
// { 1, 2, 9, 4, 5, 6 }
// 3
// -1
// { 8, 1, 2, 9, 4, 5, 6 }
// { 8, 1, 2, 9, 5, 6 }
```

Niveau 3b: IDictionary<TKey, TValue>

- IDictionary<TKey,TValue> er standard interface for **key/value**-baserede collections. Elementer er af typen **KeyValuePair<TKey, TValue>**.
- Elementer kan tilgås via indexer på nøgle.
- **Nøgler skal være unikke (undersøges via objektet Equals() metode)**

```
public interface IDictionary <TKey, TValue> :  
    ICollection <KeyValuePair <TKey, TValue>>, IEnumerable  
{  
    bool ContainsKey (TKey key);  
    bool TryGetValue (TKey key, out TValue value);  
    void Add          (TKey key, TValue value);  
    bool Remove       (TKey key);  
  
    TValue this [TKey key]    { get; set; } // indexer - ved nøgle  
    ICollection <TKey> Keys   { get; }      // Kun nøgler  
    ICollection <TValue> Values { get; }     // Kun værdier  
}
```

```
KeyValuePair<int, string> kvp = new KeyValuePair<int, string>(17, "Hej");  
Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
```

```

static void Main(string[] args) {
    Person lis = new Person("210186-1111", "Lis");
    Person per = new Person("140385-2222", "Per");

    IDictionary<string, Person> personer = new Dictionary<string, Person>();
    //Tilføj elementer - via Add() el. indexer
    personer.Add(lis.Cpr, lis);
    personer.Add(new KeyValuePair<string, Person>(per.Cpr, per));
    personer[per.Cpr] = per; //Indsæt / overskriv
    personer.Add(lis.Cpr, lis); //ArgumentException - nøgle findes allerede.

    //Tilgå elementer
    Person perIgen = personer["140385-2222"];

    Person johnDoe;
    string unknownCpr = "000000-0000";
    johnDoe = personer[unknownCpr]; //giver KeyNotFoundException
    if (personer.ContainsKey(unknownCpr)) { /* */ }
    if (personer.TryGetValue(unknownCpr, out johnDoe))
        Console.WriteLine("Fundet!");
}

```

```

class Person {
    public string Cpr { get; set; }
    public string Name { get; set; }
    public Person(string cpr, string name) { /* */ }
}

```


Enumeration of dictionary-elementer

```
static void Main(string[] args)
{
    IDictionary<string, double> tal = new Dictionary<string, double>()
    {
        {"et", 1 },
        {"to", 2 },
        {"tre", 3 },
        {"fire", 4 },
        {"fem", 5 }
    };

    foreach (KeyValuePair<string, double> kvp in tal)
        Console.WriteLine("Key={0}, Value={1}", kvp.Key, kvp.Value);

    foreach (string key in tal.Keys)
        Console.WriteLine("Key={0}, Value={1}", key, tal[key]);

    foreach (double value in tal.Values)
        Console.WriteLine("Missing key, but value is {0}", value);
}
```

1

2

3

Indexer

- For at kunne understøtte indeksering (via nøgle eller index-værdi) skal vi tilbyde en index operator.
- (Typisk) Udseende:
 - `this[int index]`
 - `this[TKey key]`
- En indekser kan dog tage et vilkårligt antal parametre (ligesom alle andre metoder).
- Syntaks: Hybrid mellem property og metode
- En indexer er især nyttig til indeksering på collection klasser, men kan erklæres på en hvilken som helst klasse.

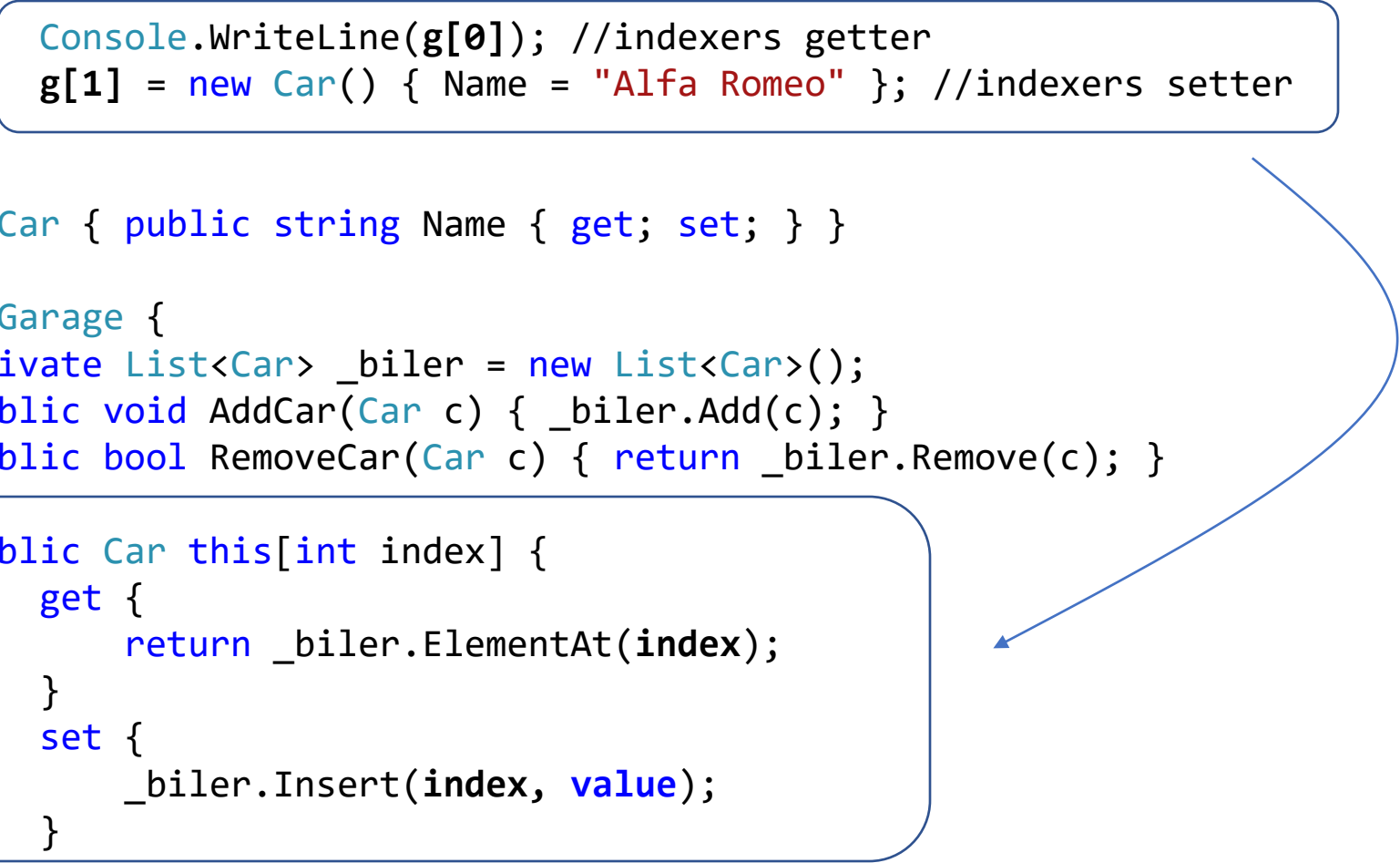
```
class Program {
    static void Main(string[] args) {
        Garage g = new Garage();
        g.AddCar(new Car() { Name = "Mazda 3" });
        g.AddCar(new Car() { Name = "Volvo" });

        Console.WriteLine(g[0]); //indexers getter
        g[1] = new Car() { Name = "Alfa Romeo" }; //indexers setter
    }
}

class Car { public string Name { get; set; } }

class Garage {
    private List<Car> _biler = new List<Car>();
    public void AddCar(Car c) { _biler.Add(c); }
    public bool RemoveCar(Car c) { return _biler.Remove(c); }

    public Car this[int index] {
        get {
            return _biler.ElementAt(index);
        }
        set {
            _biler.Insert(index, value);
        }
    }
}
```



Niveau 4: Konkrete klasser

- Der findes seks overordnede indbyggede datastrukturer:
 - Stack<T> (Last In First Out - LIFO)
 - Queue<T> (First In First Out - FIFO)
 - LinkedList<T>
 - HashSet<T>
 - List<T>
 - Implementation af IList<T>
 - Dictionary<TKey, TValue>
 - Implementation af IDictionary<TKey, TValue>

Stack<T>

- `public class Stack<T> :`
 `IEnumerable<T>, ICollection, IEnumerable`
- (Altså den ikke-generiske ICollection -> giver Count)
- Primære metoder:

Metode	Beskrivelse
Peek()	Returnerer objektet på toppen af stakken uden at fjerne det
Pop()	Fjerner, og returnerer, objektet på toppen af stakken
Push()	Indsætter objekt på toppen af stakken

Eksempel

```
static void Main(string[] args)
{
    Stack<int> stak = new Stack<int>();
    // Vi har ingen collection initializer da den ikke-generiske
    // ICollection ikke tilbyder en Add() metode.
    stak.Push(5);
    stak.Push(8);
    stak.Push(2);

    Console.WriteLine(stak.Peek()); //2 (jf. LIFO)
    Console.WriteLine("Stakken indeholder: ");
    foreach (int element in stak)    // { 5, 8, 2}
        Console.WriteLine(element);

    Console.WriteLine(stak.Pop()); //2 (stadigvæk)
    Console.WriteLine("Stakken indeholder: ");
    foreach (int element in stak)    // { 5, 8 }
        Console.WriteLine(element);
}
```

Queue<T>

- `public class Queue<T> :`
 `IEnumerable<T>, ICollection, IEnumerable`
- (Altså den ikke-generiske ICollection -> giver Count)
- Primære metoder:

Metode	Beskrivelse
Peek()	Returnerer objektet forrest i køen uden at fjerne det
Dequeue()	Fjerner, og returnerer, objektet forrest i køen
Enqueue()	Indsætter objekt i enden af køen

Eksempel

```
static void Main(string[] args)
{
    Queue<int> kø = new Queue<int>();
    //Vi har (stadig) ingen collection initializer da den ikke-generiske
    //ICollection (stadig) ikke tilbyder en Add() metode.

    kø.Enqueue(5);
    kø.Enqueue(8);
    kø.Enqueue(2);

    Console.WriteLine(kø.Peek()); //5 (jf. FIFO)
    Console.WriteLine("Køen indeholder: ");
    foreach (int element in kø) // { 5, 8, 2}
        Console.WriteLine(element);

    Console.WriteLine(kø.Dequeue()); //5 (stadigvæk)
    Console.WriteLine("Køen indeholder: ");
    foreach (int element in kø) // { 8, 2 }
        Console.WriteLine(element);
}
```


LinkedList<T>

- `public class LinkedList<T> : ICollection<T>`
- LinkedList<T> er en (dobbel)t-kædet liste
- Hver knude i en LinkedList<T> er af typen [LinkedListNode<T>](#).
- **First** og **Last** repræsenterer hhv. første og sidste element i den kædede liste
- Indsæt/fjern ift. head/tail eller bestemt knude.

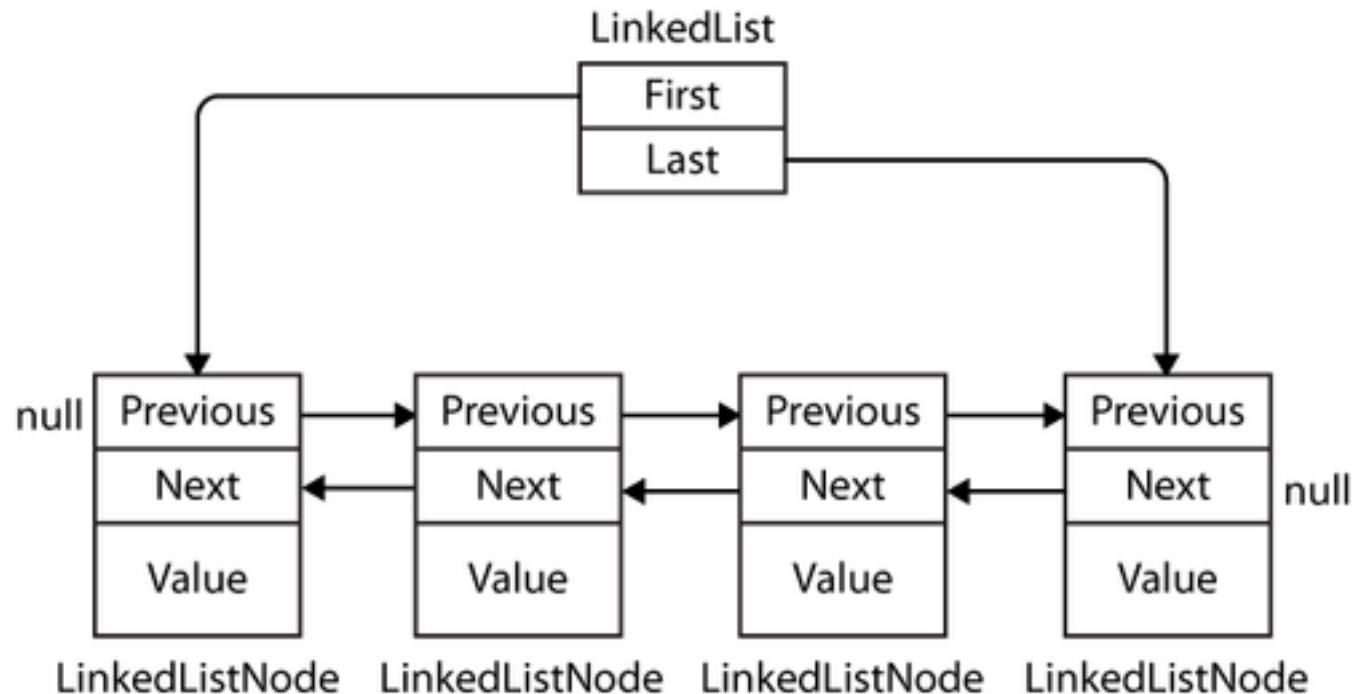


Fig 7-4 i "C#4.0 In a Nutshell"

Properties

[Count](#)

[First](#)

[Last](#)

Metoder

[AddAfter\(LinkedListNode<T> , LinkedListNode<T>\);](#) [AddAfter\(LinkedListNode<T>, T\)](#)

[AddBefore\(LinkedListNode<T>, LinkedListNode<T>\);](#) [AddBefore\(LinkedListNode<T>, T\)](#)

[AddFirst\(T\);](#) [AddFirst\(LinkedListNode<T>\)](#)

[AddLast\(T\);](#) [AddLast\(LinkedListNode<T>\)](#)

[Clear](#)

[Contains](#)

[Find](#)

[FindLast](#)

[Remove\(T\);](#) [Remove\(LinkedListNode<T>\)](#)

[RemoveFirst](#)

[RemoveLast](#)

Eksempel

```
private static void TestLinkedList()
{
    LinkedList<int> ll = new LinkedList<int>();

    ll.AddFirst(5); // { 5 }
    ll.AddLast(8); // { 5, 8 }
    LinkedListNode<int> node = ll.Find(8);
    ll.AddBefore(node, 9); // { 5, 9, 8 }
    ll.AddAfter(node, 2); // { 5, 9, 8, 2 }
    ll.AddAfter(node, new LinkedListNode<int>(6)); // { 5, 9, 8, 6, 2 }

    LinkedListNode<int> first = ll.First;
    Console.WriteLine("Third value: " + first.Next.Next.Value);
    Console.WriteLine("Fourth value: " + ll.Last.Previous.Value);

    ll.Remove(node); // { 5, 9, 6, 2 }
    ll.Remove(9); // { 5, 6, 2 }
    ll.RemoveFirst(); // { 6, 2 }
    ll.RemoveLast(); // { 6 }
}
```

ISet<T> og HashSet<T>

- Implementation af mængde-begrebet, dvs. duplikker forhindres
- **Identitet undersøges via tjek på elementers Equals() metode**
- `public interface ISet<T> : ICollection<T>`

a.ExceptWith (IEnumerable<T> b)	$a = a \setminus (a \cap b)$
a.IntersectWith (IEnumerable<T> b)	$a = a \cap b$
a.IsProperSubsetOf (IEnumerable<T> b)	$a \subset b$
a.IsProperSupersetOf (IEnumerable<T> b)	$a \supset b$
a.IsSubsetOf (IEnumerable<T> b)	$a \subseteq b$
a.IsSupersetOf (IEnumerable<T> b)	$a \supseteq b$
a.Overlaps (IEnumerable<T> b)	$a \cap b \neq \emptyset$
a.SetEquals (IEnumerable<T> b)	$a = b$
a.SymmetricExceptWith (IEnumerable<T> b)	$a = (a \cup b) \setminus (a \cap b)$
a.UnionWith (IEnumerable<T> other)	$a = a \cup b$

```

static void Main(string[] args)
{
    ISet<int> a = new HashSet<int>() { 1, 2, 3 };
    ISet<int> b = new HashSet<int>() { 1, 1, 2, 2, 3, 3 };
    ISet<int> c = new HashSet<int>() { 1, 2, 3, 4 };
    ISet<int> d = new HashSet<int>() { 3, 4, 5, 6 };

    ISet<int> a1 = new HashSet<int>(a);
    ISet<int> c1 = new HashSet<int>(c);
    a1.ExceptWith(c);    // a1 = { }
    c1.ExceptWith(a);    // c1 = { 4 }

    c1.IntersectWith(a); // c1 = { }

    Console.WriteLine(a.IsProperSubsetOf(c));    //true
    Console.WriteLine(c.IsProperSupersetOf(a));  //true

    Console.WriteLine(a.IsSubsetOf(d)); //false

    Console.WriteLine(a.Overlaps(a1)); //false

    Console.WriteLine(a.SetEquals(b)); //true

    c.SymmetricExceptWith(d); // c = { 1, 2, 5, 6 }
}

```

List<T> og Dictionary<TKey, TValue>

- List<T> indeholder ”bekvemmeligheds” udvidelser ift IList<T>, som:
 - AddRange(IEnumerable<T> coll),
 - InsertRange(IEnumerable<T> coll),
 - RemoveRange(int index, int count),
 - RemoveAll(Predicate<T> match)
- Indbyggede sorterings- og søge-metoder kræver at elementer kan **sammenlignes**:
 - Sort() – 3 overloads
 - BinarySearch() – kræver at elementerne er sorteret
- Dictionary<TKey, TValue> implementerer IDictionary<TKey, TValue>
- Dictionary-nøgler kræver et fornuftigt **identitets**-begreb (hvilket reference-identitet ikke er).
- Er a == b? giver ligeledes anledning til **operator-overloading**.
- Begge datastrukturer kræver en **indexer** (på hhv. position og nøgle)

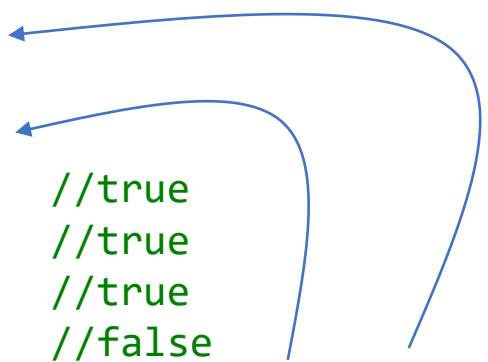
Et fornuftigt identitets-begreb

- I C# findes to slags lighed: **Reference-lighed** og **værdi-lighed**
 - *Værdi-lighed* (`==` eller `a.Equals(b)`): Hvis to objekter indeholder de samme (identificerende) værdier, så er *forventningen* at objekterne er ens.
 - *Reference-lighed* (`object.ReferenceEquals(a, b)`): To objekt-referencer refererer til det samme objekt (har samme hukommelsesadresse)
 - Som udgangspunkt implementerer referencetyper `Equals()` som `ReferenceEquals()`
- Det er ofte ønskværdigt at override `Equals` til at teste på værdi-lighed istedet for reference-lighed.

Default lighed for værdi- og referencetyper

```
//VÆRDI-TYPER
int a = 6;
int b = a;
Console.WriteLine(object.ReferenceEquals(a, b)); //false
Console.WriteLine(a.Equals(b));                //true
Console.WriteLine(a == b);                      //true

//REFERENCE-TYPER
Person p1 = new Person() { Cpr = "010101-0101" };
Person p2 = p1;
Person p3 = new Person() { Cpr = "010101-0101" };
Console.WriteLine(object.ReferenceEquals(p1, p2)); //true
Console.WriteLine(p1.Equals(p2));                //true
Console.WriteLine(p1 == p2);                     //true
Console.WriteLine(object.ReferenceEquals(p1, p3)); //false
Console.WriteLine(p1.Equals(p3));                //false (burde være true)
Console.WriteLine(p1 == p3);                     //false (burde være true)
```



Guidelines for implementation af værdi-lighed

1. Specialiser **Equals(object obj)** metoden.
 2. Når Equals() specialiseres, bør **GetHashCode()** også specialeres.
 - Det forventes at to ens objekter har samme hash-kode
 3. Når Equals() specialiseres, bør den generiske Equals(T obj) – fra IEquatable<T> interfacet - også implementeres (af performance hensyn).
- Krav: Equals() tjek må ikke kaste exception

GetHashCode(): Ens objekter skal have samme hash code

- En hashcode er et tilfældigt tal der bruges ifm Equals(), samt til at balancere nøglerne i et dictionary.
- Regl:
 - **Hvis `a.Equals(b)` så skal `a.GetHashCode() == b.GetHashCode()`**
- Hash koder skal så vidt muligt være unikke (forbedrer performance ved Dictionary lookup)
- Hash kode genereres typisk med XOR på instans-variable (da det giver et meget tilfældigt tal)
- **Krav:** GetHashCode() må aldrig kaste exceptions.
- Bemærk: `a.GetHashCode() == b.GetHashCode()` betyder ikke nødvendigvis at `a.Equals(b)` er sand.
- Hash-kode kan caches (gemmes i instansvariabel) for at forbedre performance. Hashcode bør være immutable!

XOR:	A = 3:	0	0	1	1
	B = 14:	1	1	1	0
	A ^ B = 13	1	1	0	1

Eksempel

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public double Weight { get; set; }

    public override int GetHashCode()
    {
        return
            (FirstName == null ? 0 : FirstName.GetHashCode()) ^
            (LastName == null ? 0 : LastName.GetHashCode()) ^
            Age.GetHashCode() ^
            Weight.GetHashCode();
    }
}
```

Tjek-procedure ved Equals

- Vi skal lave en række checks når vi overrider Equals:
 1. Check for null
 - null -> false
 2. Check for reference-lighed, hvis typen er en reference-type
 - Reference-lighed -> true
 3. Check for ækvivalente typer
 - Nej -> false
 4. Check for ækvivalente hash-koder
 - Genvej, da to ens objekter ikke kan have *forskellige* hash koder
 - KRÆVER selvfølgelig at GetHashCode() er implementeret
 - Ikke ækvivalente koder -> false
 5. Check base.Equals() - HVISS baseklassen overrider Equals()!!
 - False -> false
 6. **Check Equals() på identificerende felter**

Eksempel på overridding af Equals på Person

```
public class Person : IEquatable<Person>
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public double Weight { get; set; }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (this.GetType() != obj.GetType())
            return false;

        //Ikke null og Person type fastslået -> kalder Equals(Person)
        return Equals((Person)obj);
    }
}
```

Husk at implementere IEquatable

// 1. Check for null

// 2. Check type *

Diagram illustrating the implementation of the `Equals` method in the `Person` class, which implements the `IEquatable<Person>` interface.

The code shows the `Equals` method implementation, which includes checks for null and type, and then calls `Equals(Person)` for the actual comparison.

* NB: Hvis f.eks. Dat og Sw studerende skal kunne betragtes som ens, så brug `is` istedet.

```
public bool Equals(Person obj) { //Impl. af IEquatable<Person>
    //1. Check for null - hvis ref type
    if (obj == null)
        return false;

    //2. Check for referenceEquals - hvis ref type
    if (ReferenceEquals(this, obj))
        return true;

    //3. Check hashCode (forudsat override af GetHashCode())
    if (GetHashCode() != obj.GetHashCode())
        return false;

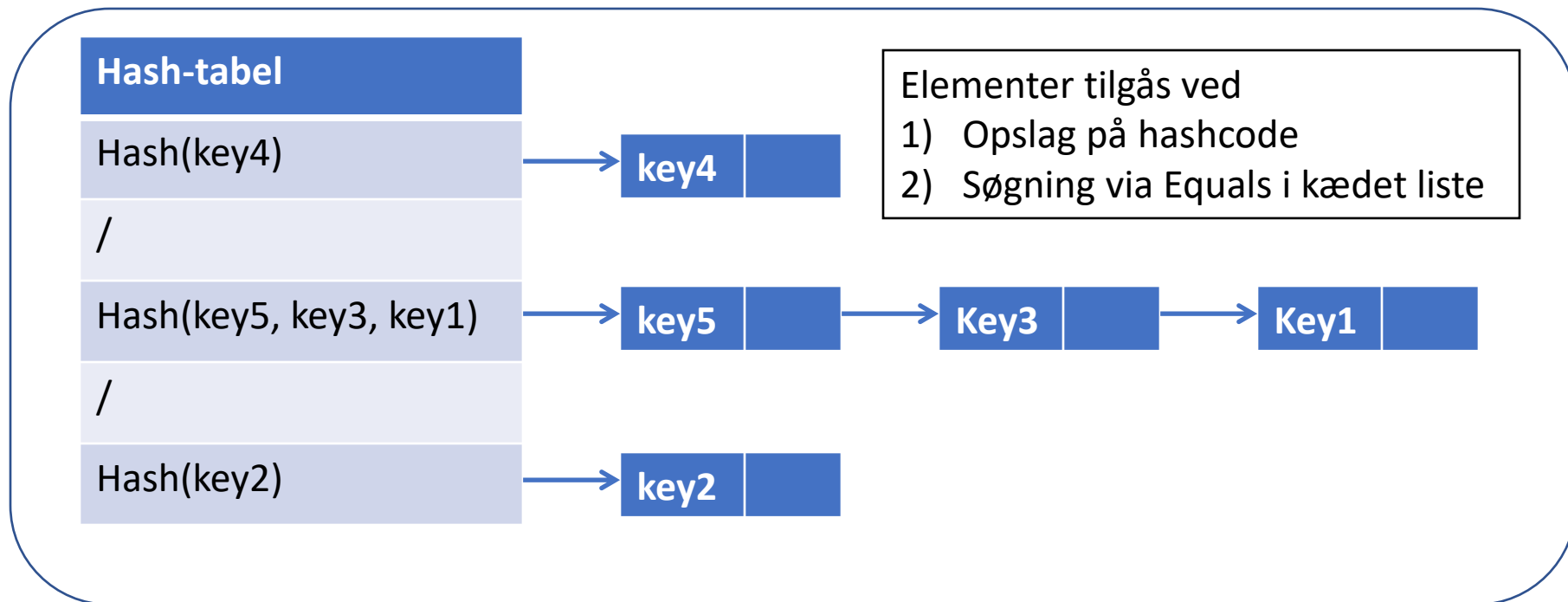
    //4. Check base-typens Equals (HVISS den overrider Equals())
    if (!base.Equals(obj))
        return false;

    //5. Sammenlign identificerende felter
    bool firstNameEqual = (FirstName == null && obj.FirstName == null) ||
        (FirstName.Equals(obj.FirstName));
    bool lastNameEqual = (LastName == null && obj.LastName == null) ||
        (LastName.Equals(obj.LastName));
    return
        firstNameEqual && lastNameEqual && Age.Equals(obj.Age) &&
        Weight.Equals(obj.Weight);
}
```

It depends

Equals og mutérbarhed

- **Equals()/GetHashCode()** bør kun basere sig på **readonly** felter!!
- Årsag: Hvis hashkode ændres, fungerer Dictionary og HashSet ikke efter hensigten.



Standard query operatorer (LINQ)

- `IEnumerable<T>` har kun én metode - `GetEnumerator()` – der tillader os at traversere collection klasser med `foreach`.
- Derudover findes mere end 40 extension metoder, kaldet **standard query operatorer**, der giver mulighed for at angive queries på en `IEnumerable<T>`.
- Der findes fem primære operatorer (inspireret af SQL):
 - Filtrering af elementer med **Where()**.
 - Sortering af elementer med **OrderBy()** og **ThenBy()**
 - Gruppering af elementer med **GroupBy()**
 - Projicering (transformation) af elementer med **Select()**
 - (DB-specifikt: Joins (af ikke-relaterede typer) med **Join()**)
- Query operatorerne kan kædes sammen.
- Implementeret som extensionmethods

Filtrering med Where()

- **Where()** metoden filtrerer elementer efter om de opfylder et givet **prædikat**.
- `public static IEnumerable<TSource> Where<TSource>(`
 `this IEnumerable<TSource> source,`
 `Func<TSource, bool> predicate)`

Extension metode

```
List<Person> persons = new List<Person>() { /* */ };  
  
IEnumerable<Person> drivers = persons.Where(p => p.Age >= 18);  
  
var teens = persons.Where(p => p.Age > 12 && p.Age < 20);  
var hansens = teens.Where(p => p.LastName == "Hansen");
```

Sortering med OrderBy() og ThenBy()

- Sortering: **OrderBy()** og **OrderByDescending()**
 - og derefter **ThenBy()** og **ThenByDescending()**
- **OrderBy()** og **ThenBy()** returnerer en **IOrderedEnumerable** der har en **ThenBy()** metode til yderligere sortering.
 - **OrderBy()** sletter eventuelle tidligere **OrderBy()** queries – brug derfor altid **ThenBy()** til yderligere sortering.
- Der forventes en **Func<TSource, TKey> keySelector**.
- **TKey** skal implementere **IComparable**
 - ellers returneres de i oprindelige rækkefølge.

Sortering med OrderBy() og ThenBy()

```
IEnumerable<Person> sortedNames = persons.OrderBy(p => p.LastName);
foreach (Person p in sortedNames)
    Console.WriteLine($"{p.LastName}, {p.FirstName}");

//{Hansen, Kaj}, {Hansen, Bo}, {Jensen, Niels}, {Jensen, Jens}

IEnumerable<Person> sortedNames2 = persons
    .OrderBy(p => p.LastName)
    .ThenBy(p => p.FirstName)
    .ThenByDescending(p => p.Age);
foreach (Person p in sortedNames2)
    Console.WriteLine($"{p.LastName}, {p.FirstName}");

//{Hansen, Bo}, {Hansen, Kaj}, {Jensen, Jens}, {Jensen, Niels}
```

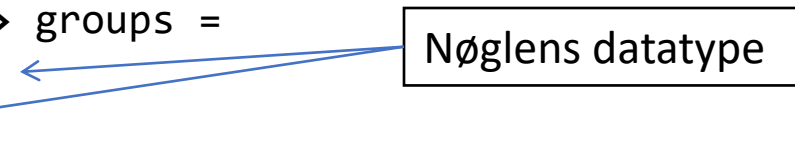
Gruppering af resultater med GroupBy()

- GroupBy() metoden giver mulighed for at gruppere objekter med ens karakteristika
- Flad sekvens -> grupper med ens **key**.

```
List<Person> persons = new List<Person>() { ... }

IEnumerable<IGrouping<string, Person>> groups =
    persons.GroupBy(p => p.LastName);

foreach (IGrouping<string, Person> lastnameGroup in groups) {
    Console.WriteLine($"Personer med efternavn {lastnameGroup.Key}:");
    foreach (Person p in lastnameGroup)
        Console.WriteLine("- {p.FirstName} {p.LastName}");
}
```



Nøglen datatype

Projicering med Select()

- **Select()** metoden kan bruges til at projicere (transformere) elementerne i en collection til en anden type
- **Select<TSource, TResult>(Func<TSource, TResult>)**

```
IEnumerable<string> driverFirstNames = persons
    .Where(p => p.Age >= 18)    //returnerer IEnumerable<Person>
    .Select(p => p.FirstName); //returnerer IEnumerable<string>

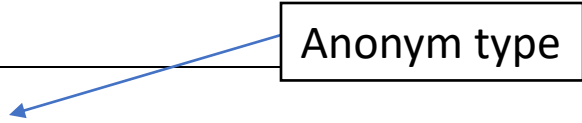
string lastName = "Jensen";
IEnumerable<int> nameAges = persons
    .Where(p => p.LastName == lastName) //IEnumerable<Person>
    .Select(p => p.Age);                //IEnumerable<int>
```

Select() med anonyme typer

Vi kan bruge Select() i kombination med en anonym type til kun at udtrække relevant data

```
var firstAndLast = persons
    .Select(p => new { p.FirstName, p.LastName });

foreach (var fl in firstAndLast)
    Console.WriteLine("First: {0}, Last: {1}",
        fl.FirstName, fl.LastName);
```



Anonym type

Anonyme typer

- **Anonyme typer** er data typer der erklæres af *compileren* – ikke gennem en eksplicit klasse definition
- Compileren genererer en CIL klasse indeholdende properties med de givne navne, data typer og værdier

```
static void Main(string[] args) {  
    var person = new  
    {  
        Name = "John",  
        NumLegs = 2,  
        Born = new DateTime(1965, 12, 24),  
        Weight = 79.9,  
        Money = 23.45M  
    };  
    Console.WriteLine(person.Name); //”John”  
    Console.WriteLine(person.Born.ToShortDateString()); //12/24/1965  
}
```

Implicit typeerklæring

- Lokale variable defineret med **var** tildeles type implicit
- Compiler afgør type på variabel fra assignment.

```
var Name = "John";    //string
var NumLegs = 2;      //int
var Born = new DateTime(1965, 12, 24);
var Weight = 79.9;    //double
var Money = 23.45M;   //decimal
```

- Implicitte variable er type-sikre.
- Begræns implicit typeerklæring til anonyme typer
- Med mindre typen er krystalklar fra kontekst:
 - `var names = new List<string>` //ja, ok
 - `var aValue = GetValue();` //hvad type er det?

Collectionoperatorer

- De 5 grundlæggende operatorer er:
 - Select, Where, OrderBy, GroupBy (og Join)
- Men der findes mange flere:
 - Partitionerings-operatorer:
 - Take, TakeWhile, Skip, SkipWhile
 - Element-operatorer:
 - First, FirstOrDefault, Last, LastOrDefault, Single, ElementAt, DefaultIfEmpty
 - Mængde-operatorer:
 - Distinct, Union, Intersect, Except, Concat, Reverse
 - Aggregerings-operatorer:
 - Count, Sum, Min, Max, Avg

Partitioning operator

	List<int> numbers = {3, 5, 6, 8, 11, 12}
Take(int count)	numbers.Take(2) => {3, 5}
TakeWhile(Func<TSource, bool>)	numbers.TakeWhile(n => n < 5) => {3}
Skip(int count)	numbers.Skip(4) => {11, 12}
SkipWhile(Func<TSource, bool>)	numbers.SkipWhile(n => n < 10) => {11, 12}

Elementoperatorer

	<code>List<int> numbers = {3, 5, 6, 8, 0}</code>
<code>First(Func<TSource,bool>)</code>	<code>numbers.First(n => n > 10) -> EXCEPTION HVIS Ø</code>
<code>FirstOrDefault()</code>	<code>numbers.First(n => n > 10) -> 0</code>
<code>Last()</code>	Sidste element, eller exception
<code>LastOrDefault()</code>	
<code>Single()</code>	Returnerer det ENESTE element der opfylder prædikat – eller exception
<code>SingleOrDefault()</code>	
<code>ElementAt(offset)</code>	Elem på specificerede indeks-plads el. <code>OutOfRangeException</code>
<code>ElementAtOrDefault()</code>	Eller default hvis <code>OutOfRangeException</code>
<code>DefaultIfEmpty()</code>	Default værdi hvis listen er tom eller null - Ellers listen selv (ignoreres ved ikke-tom) <code>IEnumerable<int> v = numbers.DefaultIfEmpty();</code>

Aggregeringsfunktioner

	<code>var persons = new List<Person>() {...}</code>
<code>Count()</code> <code>Count(Func<TSource, bool>)</code>	Antal elementer (der matcher prædikat) <code>int preSchoolCount = persons.Count(p => p.Age < 5)</code>
<code>Average()</code> <code>Average(Func<TSource, numeric>)</code> - numeric = int, long, decimal, double, float	<code>double avgAge = persons.Average(p => p.Age)</code>
<code>Sum()</code> - Som average (numeriske værdier)	<code>double totalAge = persons.Sum(p => p.Age)</code>
<code>Max()</code> <code>Max(Func<TSource, TResult>)</code>	<code>int maxAge = persons.Max(p => p.Age)</code>
<code>Min()</code>	Som max

Mængdeoperatorer og andet nyttigt

	<code>List<int> a = {1, 3, 3, 4}, b = {2, 1, 5}</code>
<code>Distinct()</code>	<code>var aDistinct = {1, 3, 4}</code>
<code>Union()</code>	<code>var ab = a.Union(b) => {1, 3, 4, 2, 5}</code>
<code>Intersect()</code>	<code>var ab = a.Intersect(b) => {1}</code>
<code>Except()</code>	<code>var e = a.Except(b) => {3, 4}</code>
<code>Concat()</code>	<code>var c = a.Concat(b) => {1, 3, 3, 4, 2, 1, 5}</code>
<code>SequenceEqual()</code>	<code>bool b = a.SequenceEqual(b) => false</code> <code>c = {1, 3, 3, 4, 4}</code> <code>bool b1 = a.SequenceEqual(c) => false</code> <code>d = {1, 3, 4, 3}</code> <code>bool b2 = a.SequenceEqual(d) => false</code> - Værdier og rækkefølge skal matche
<code>Reverse()</code>	<code>a.Reverse() => {4, 3, 3, 1}</code>
<code>OfType<T>()</code> Vehicle; Car: Vehicle; Bus: Vehicle	<code>List<Vehicle> vs = new List<Vehicle>() {...}</code> <code>IEnumerable<Car> cs = vs.OfType<Car>()</code>

Konverteringsoperatorer

- `ToArray()`
- `ToList()`
- `ToDictionary(Func<TSource, Tkey>)`

```
int[] ages = persons.Select(p => p.Age).ToArray();

List<string> firstnames = persons.Select(p => p.FirstName).ToList();

Dictionary<string, Person> personDic =
    persons.ToDictionary(p => p.FirstName);

foreach (string first in personDic.Keys) {
    Console.WriteLine(personDic[first]);
}
```

Key skal være unik – ellers `ArgumentException`

Opsummering

- I .Net findes et hierarki af datastrukturer
 - Som i Java og andre sprog
- Standardbibliotek
- Giver mulighed for iteration
- LINQ er et simpelt kraftfuldt bibliotek til ofte brugte operationer på iterérbare datastrukturer