Object-Oriented Programming with C#

Programming, Part III

By Per Laursen
24-08-2025

Introduction

The previous chapters have armed us with tools to create rather sophisticated applications with complex business logic. In this chapter, we introduce some additional programming concepts that are useful in certain scenarios. Before doing that, we will have a look at a classic topic in Computer Science known as **run-time complexity**. This will help us make informed decisions later on in the chapter.

We claimed earlier in this text that knowledge about the **List** and **Dictionary** class will go a long way with regards to managing a collection of identical elements, be they of a simple type or of a class type. That is indeed true, but additional collection classes exist in the .NET class library worth knowing about. Also, we should be a bit more precise about the advantages and drawbacks of the various collection classes. These advantages and drawbacks are often closely related to the **usage pattern** of the data, so we need to know more details about this.

We have previously discussed loop statements (**while**- and **for**-loops) as a construction for executing the same block of code multiple times. Another way of achieving this is to use **recursion**. Recursion is not a specific type of statement; it denotes the technique of letting a method call itself repeatedly, until some condition is no longer true. In that sense, recursion is definitely strongly related to traditional iteration, but certain problems can be solved very elegantly using recursion.

In the previous chapter on functions as parameters, we saw examples of using lambda expressions as parameters to various methods, for instance to the **FindAll** method for the **List** class. This was also an example of a very common problem in programming: Given a collection of data, find a subset of the data which fulfills certain conditions. We have solved such problems in various ways. One way is to explicitly iterate over the colllection – typically using a **foreach** loop – and evaluate each element in the collection against a selection condition. Another way was to use e.g. the **FindAll** method, which only require us to provide the selection condition itself. A third way available in C# is to use so-called **Language Integrated Queries**, or just **LINQ**.

Run-time complexity

When we need to execute a piece of code, we will often be interested in knowing the time it takes for the code to be executed. The specific time may depend on several factors, including

- The computer hardware
- Other programs running simultaneously on the computer
- The code itself, i.e. has the code been written in an effective way

Run-time complexity deals with the last factor. We will never be able to determine exactly how long execution of a piece of code will take in "absolute time" (measured in e.g. microseconds), but we can analyse how long execution will take relatively to the size of the data the code needs to process (we will denote such a piece of code as an algorithm).

Let's see an example. Suppose we have a **List** of **int** values, and want to determine if a specific value is in the list (the **List** class actually contains such a method, but we will analyse our own implementation here). This is a classic example of the so-called **Linear Search** algorithm:

```
List<int> numbers = new List<int>();
// Insert some values into the list…

int valueToLookFor = 37;
bool valueWasFound = false;
for (int index = 0; index < numbers.Count && !valueWasFound; index++)
{
      if (numbers[index] == valueToLookFor)
      {
            valueWasFound = true;
      }
}
```

The highlighted part is particularly interesting, since this part decides how many loop iterations we will perform. Not surprisingly, the number of iterations will increase, if we add more elements to the list (on average, we will probably need to search half the list). We can probably expect that if we double the number of elements in the list, the time it takes to complete the loop will also roughly double. If we increase the number ten times, the time it takes to complete the loop will also increase roughly ten times, and so on.

We can express this relation between list size and running time a bit more formally. If there are **n** elements in the list, the time **T** it takes to complete the loop is then:

$$T = c * n$$

The letter **c** denotes a constant; if the left- and right-hand side should truly be equal to each other, then c must be equal to **T**/**n** (**T** divided by **n**). We stated above that the absolute running time is not so interesting in itself, since it may depend on e.g. specific computer hardware. Therefore, we usually throw away the constant, and express the relation between running time and data size in this form:

$$T = O(n)$$

The **O-notation** (also known as the **Big-Oh notation**) is a standard notation in Computer Science, and should here be read as "the running time **T** is proportional to **n**". If **n** doubles, we expect **T** to double, and so on.

This is a fairly simple example. Suppose we want to calculate the product of all combinations of value pairs in the list:

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

This is a bit harder to analyse, but you can probably figure out that if we double the number of values in the list (i.e the value of **n**), the number of products printed will now quadruple, i.e. **T** will quadruple. An increase of **n** by a factor of 10 will increase the value of **T** by a factor of 100. In general, the relationship is now:

$$T = O(n^2)$$

This should be read as "the running time **T** is proportional to the square of **n**". Since this loop does something different than the first loop, we cannot really use this information to say whether or not one of these two algorithms is "better" than the other. However, if we did have two algorithms solving the same problem, we would deem the algorithm having run-time complexity O(**n**) as better than the algorithm having run-time complexity O(n^2). Even though the O(**n**)-algorithm might actually be slower

than the O($\mathbf{n^2}$)-algorithm for small values of **n**, the O(**n**)-algorithm will at some point beat the O($\mathbf{n^2}$)-algorithm.

In this way, we consider an O($\mathbf{n^2}$)-algorithm to be "slower" than an O(**n**)-algorithm. Can an algorithm be faster than O(**n**)? Indeed it can! Consider this problem: Retrieve the first element in a list. This sounds trivial, and we can indeed solve it with a single line of code:

```
Console.WriteLine(numbers[0]);
```

We should however be a bit cautious here; the statement **numbers[0]** will cause some code inside the **List** class to execute, so we don't know exactly what happens, or how long it takes. It does however turn out that for a **List** object, this operation is a so-called **constant-time operation**, i.e. the time needed to complete the operation does not depend on the size of data. This (hopefully) also seems intuitively correct; the time needed to find the first element in a list should not depend on the total number of elements in the list.

The run-time complexity of a constant-time operation is expressed as:

$$\mathbf{T} = \mathbf{O}(\mathbf{1})$$

This may look a bit weird the first time you see it, but it does makes perfect sense. This is indeed a run-time that does not depend on **n**. The **List** class actually has the property that any element can be retrieved in constant time.

What about the range from O(**1**) to O(**n**)? Are there algorithms that are not constant-time, but have a run-time complexity "lower" that O(**n**)? Indeed there are! Quite a lot, actually. Consider for instance the problem of checking if a given number is present in a list of sorted numbers. We will not write up code for this problem, but just try to think it through.

If the list is sorted, we could maybe look at the element in the middle of the list. We know we can look up this element in constant time. This can give us three outcomes:

1. The element is equal to the value we are looking for
2. The element is smaller than the value we are looking for
3. The element is greater than the value we are looking for

What is our next action in each case? Case 1 is easy: we are done. Case 2 is slightly more tricky, but if the middle element is smaller than the value we are looking for, we can draw the following conclusion: Either the element is in the "higher" half of the list (since the list is sorted), or it is not there at all. We can do a similar analysis for Case 3. We can thus repeat the logic above once more, but – and this is the crucial point – only for the "higher" half of the list! In this single step, which we know only takes constant time, we have effectively reduced the size of the problem to half the original size! We can keep doing this over and over, until we have a list of length 1, i.e. a single element. If this element is indeed the value we are looking for, we can answer the original question with **true**, otherwise answer it with **false**.

It is hopefully obvious that it is much faster to find a given value in a sorted list, as compared to an unsorted list. But how much faster? Suppose we started with a list of 64 elements. The first step will reduce the list size to 32, then 16, then 8, 4, 2 and finally 1. A total of 6 steps. How many more steps are needed, if the list contained 128 elements? Just one, i.e. a total of 7 steps. In general, we need **p** constant-time steps in order to find an element in a sorted list with $2^{\mathbf{p}}$ elements. This gives us a relation between the problem size and the running time:

$$\mathbf{n} = c * 2^{\mathbf{T}}$$

This is somewhat backwards, since we want the running time **T** as a function of **n**. If you remember your high-school mathematics, you can solve this by using **logarithms**, and end up with this run-time complexity:

$$\mathbf{T} = \mathbf{O}(\log(\mathbf{n}))$$

The term **log** means **logarithm**. If you don't remember (or know) what logarithms are, the important property to know here is that the logarithm function grows very slowly. A couple of examples of logarithm values are given below:

n	**log(n)**
$1.000 \approx 2^{10}$	≈ 10
$1.000.000 \approx 2^{20}$	≈ 20
$1.000.000.000 \approx 2^{30}$	≈ 30

An O(log(**n**)) algorithm is thus much, much faster than an O(**n**) algorithm. If you are familiar with an old-fashioned paper phonebook – where the person/number entries are ordered alphabetically by person name – the rather simple task of looking up a

number for a given person is an O(log(**n**)) algorithm, while the much harder task of looking up a person for a given number is an O(**n**) algorithm.

Run-time complexity can in this way be a very useful tool for selecting one implementation strategy over another. An important example of this is the problem of selecting the best (in terms of run-time) collection class for storing a collection of data, given a certain usage pattern for this data. We will get back to this topic soon, but let's first get to know some additional collection classes.

Data Structures revisited

We claimed earlier that knowledge about the **List** and **Dictionary** class will go a long way with regards to managing a collection of identical elements, be they of a simple type or of a class type. That is indeed true, but there are additional collection classes in the .NET class library worth knowing about. Also, we should be a bit more precise about the advantages and drawbacks of the various collection classes. These advantages and drawbacks are often closely related to the usage pattern of the data, so we need to know more details about this.

We give an overview of a few of these collection classes below; if you need more detailed information, there are plenty of resources to be found online.

The LinkedList class

We claimed earlier that the **List** class is very efficient with regards to retrieving an element. Retrieving an element by index takes constant time, and can obviously not be done faster. Insertion and deletion are a bit different, however.

The **Add** method inserts a given element at the end of the list. This can also be done in constant time, since we do not need to move any other elements in the list. Insertion at the front of the list – by using the **InsertAt** method – will however cause all of the existing elements in the list to be moved up one position, to make room for the new element. If we make an insertion at a random position in the list, we will on average need to move half of the elements up one position. Insertion at a random position in a **List** containing **n** elements is therefore considered an O(**n**) operation. It is thus definitely – in theory, at least – possible to perform such insertions more efficiently.

The **LinkedList** offers better performance for certain types of insertion. Where a **List** can be thought of as one chunck of memory allocated to contain the elements in the list, the **LinkedList** can be thought of as several small chunks of memory, where each chunk contain one element, plus a reference to the previous and next element in the data structure. The **LinkedList** class provides properties **First** and **Last**, which returns references to the first and last element in the linked list, respectively.

Insertion into a linked list can be done in constant time at any position in the list, once you have a reference to the position where the element should be inserted. Suppose the new element is named E, and you wish to insert it just after an element named X. Before insertion, element Y follows after element X:

Insertion of E requires just two steps:

1. Set X to refer to E.
2. Set E to refer to Y.

These properties have several consequences with regards to the performance of a **LinkedList**:

- Looking up an element by index is **inefficient** (takes O(**n**)), since you will have to start at the first element of the linked list, and step through the chain of elements one by one.
- Inserting an element at the end of the linked list is **efficient** (takes O(1)), since the property **Last** returns the last element, and insertion as such only takes constant time.
- Inserting an element at the front of the linked list is **efficient** (takes O(1)), since the property **First** returns the first element, and insertion as such only takes constant time.
- Inserting an element in a random position is **inefficient** (takes O(**n**)), since you will have to look up the position first before inserting.

It is fairly easy to make a similar analysis for deletion, which is efficient when done at both ends of the linked list, otherwise inefficient.

All this leads us to the big question:

When should you use a **LinkedList** instead of a **List**?

This will require knowledge about the typical usage pattern for the collection. If you e.g. often need to look up an element by index, it will be a bad choice to use a **LinkedList**, since a **List** does this much more efficiently. However, if you:

- Often need to do insertions (or deletions) at the front of the collection.
- Often need to apply some operation to all elements, using e.g. a **foreach** loop.
- Rarely need to look up specific elements by index

it may be worth using a **LinkedList** instead. A problem related hereto is the fact that the **List** class and the **LinkedList** class do not offer the same set of properties and methods. If you have written code using a **List** for your collection, you will need to change that code if you decide to switch to a **LinkedList**. A way to encapsulate this problem could be to define an application-specific collection interface, containing exactly those collection-oriented properties and methods you need for your application. You can then create two implementations of this interface; one using a **List** class, and one using a **LinkedList** class. It will then be much simpler to conduct experiments with both classes.

The Queue and Stack class

The collection classes **Queue** and **Stack** are examples of collection classes that are not as such tuned for efficiency, but rather provide an easy-to-use interface for collections with some special properties. The terms “queue” and “stack” here denote some properties about the order in which elements enter and leave the collection.

A **queue** denotes the situation where elements leave the collection in the same order as they were entered. This resembles the real-life concept of a queue in e.g. a supermarket: If customer A enters a queue at a cash register before customer B, we also expect that customer A will be served – and thus leave the queue – before customer B. This ordering is usually denoted **FIFO** (First-In First-Out). If you need to maintain such an ordering of elements, you can use the **Queue<T>** class. The **Queue** class has three essential methods:

Enqueue(T element)	Inserts the element at the back of the queue
T Dequeue()	Returns and removes the element at the front of the queue
Peek()	Returns the element at the front of the queue

We leave it as a small exercise to think about whether a **List** or a **LinkedList** will provide the most efficient implementation of a **Queue**…

A **stack** denotes the situation where elements leave the collection in the opposite order as they were entered. You can imagine a stack of papers on a table; the bottom paper was entered first into the stack, but will be the last paper to leave the stack, since papers can only be removed from the top of the stack (at least we assume so). This ordering is usually denoted **LIFO** (Last-In First-Out). If you need to maintain such an ordering of elements, you can use the **Stack<T>** class. The **Stack** class has three essential methods:

Push(T element)	Inserts the element at the top of the stack
T Pop()	Returns and removes the element at the top of the stack
Peek()	Returns the element at the top of the stack

The reason for the diversity of method naming between **Queue** and **Stack** is mostly historical. The **Push** and **Pop** method names for the **Stack** class are common for all Object-Oriented languages, while e.g. Java uses the names **Add** and **Remove** for **Queue** methods.

The HashSet class

Whenever we have data with an obvious key/value relation, we usually prefer to use the **Dictionary** class, since it provides very efficient operations for insertion, deletion and lookup by key. We may however encounter situations where data has key-like properties, but do not refer to any other data. With "key-like" properties, we mean:

- Each element must be unique
- It must be efficient to check if an element is already present in the collection
- It must be efficient to insert an element into the collection

Furthermore, it may also be required that more advanced so-called **set-oriented operations** can be performed. A **set** is the mathematical term for a collection of elements, and certain operations are well-defined for sets, like:

Union	Given two sets A and B, the union of A and B is the set containing all elements that are a member of A or B (or both)
Intersection	Given two sets A and B, the intersection of A and B is the set containing all elements that are a member of A and B
Complement	Given two sets A and B, the complement of B is the set containing all elements that are a member of B and not a member of A
Subset	Given two sets A and B, A is a subset of B if all elements that are a member of A are also a member of B
Superset	Given two sets A and B, A is a superset of B if all elements that are a member of B are also a member of A

If you need to store data which has such key-like properties, and/or need to perform set-oriented operations on the data, the **HashSet** class offers support for this. In addition to **Add** and **Remove** methods, the class contains methods corresponding to the operations described above. See the class documentation for further details.

The "hash" part of the **HashSet** class name relates to the internal representation of the data. A so-called **hash table** is used to store data. This representation makes it possible to lookup data in "almost" constant time. By "almost" is meant that even though it is theoretically possible for the lookup to take O(**n**), it turns out that we in practice can look up elements in constant time, at the expense of using a bit more memory than by using e.g. a **List**. A **Dictionary** also uses a hash table for internal storage. If you are interested in more knowledge about hash tables, there are several sources online.

Recursion – iteration without loops

Recursion is not a specific type of statement; it denotes the technique of letting a method call itself repeatedly, until some condition is no longer true. In that sense, recursion is definitely strongly related to traditional iteration, but certain problems can be solved very elegantly using recursion. A first example of recursion is the method below:

```
public void PrintHello()
{
	Console.WriteLine("Hello");
	PrintHello();
}
```

This is not a particularly useful method, since it will keep calling itself over and over, and it is therefore effectively an **infinite loop**. We can improve the method by adding a way of breaking out of the infinite loop:

```
public void PrintHello(int numberOfCallsLeft)
{
	if (numberOfCallsLeft > 0)
	{
		Console.WriteLine("Hello");
		PrintHello(numberOfCallsLeft - 1);
	}
}
```

This will cause the the method to terminate at some point. However, this somewhat pointless functionality could have been written as e.g. a **for**-loop instead, so we have not really gained anything.

A more interesting example is the so-called **Factorial** function. The **Factorial** function takes an integer **n** as input (**n** must be a positive integer), and returns the product **n** x (**n** – 1) x (**n** – 2) x … x 2 x 1. If e.g. **n** = 5, the **Factorial** will be 5 x 4 x 3 x 2 x 1 = 120. The **Factorial** of **n** is usually written as **n!**

This definition of **n!** makes it fairly easy to calculate **n!** using a traditional loop statement. You can however also think of the definition of **n!** as:

- **Factorial**(1) = 1, and
- **Factorial**(**n**) = **n** x **Factorial**(**n** – 1)

This is a **recursive** definition of **n**!, which we can dissect into several smaller parts:

- **A trivial case**: a case for which we have a simple solution, that does not require any calculation.
- **A division strategy**: a way of splitting the problem into smaller parts, which can themselves be solved trivially or by recursion
- **A combination strategy**: a way of combining the solutions for the simpler problems into a solution for the original problem

For the **Factorial** function, these parts become:

- **A trivial case**: **Factorial**(1) = 1.
- **A division strategy**: Split **Factorial**(**n**) into **n** (trivial) and **Factorial**(**n** - 1), which can be solved by recursion
- **A combination strategy**: Multiply **n** and **Factorial**(**n** - 1).

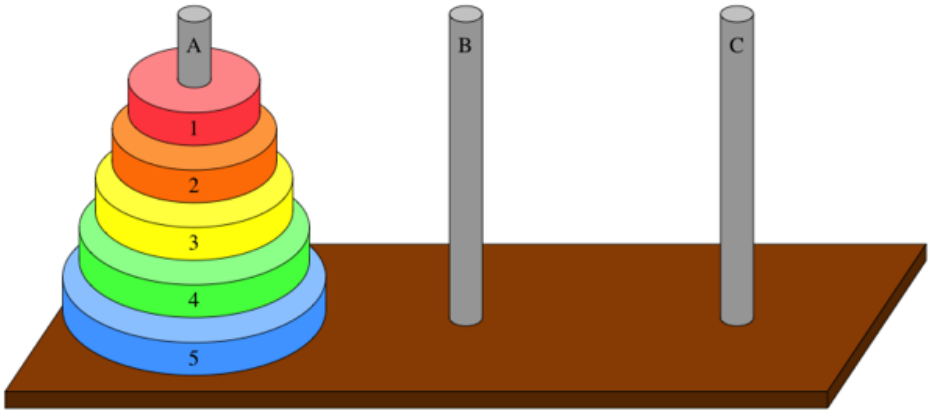
With these definitions, we can write actual recursive code for a **Factorial** method:

```
public int Factorial(int n)
{
      return (n <= 1) ? 1 : (n * Factorial(n - 1));
}
```

We have extended the "trivial case" a bit, to avoid an infinite loop if the method is called with a number smaller than 1. An alternative could be to throw an exception.

Even though the above looks fairly elegant, it is still not a significant improvement over the iterative version of **Factorial**. It serves more as an illustration of how to apply the steps defined above to a specific problem. A problem of a quite different nature – which turns out to be very elegantly solved by recursion – is the **Towers of Hanoi**[1] game (also see the illustration below). The problem to be solved in this game is as follows:

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi



1. Given three pegs A, B and C and a set of disks 1, 2, 3, …, **n**. The disks have increasing diameter, such that disk 1 is smallest, then disk 2, etc.
2. The starting point is as illustrated above, i.e. all disks are on peg A, with the largest disk at the bottom.
3. The goal is to end up with all disks on peg C, in the same order as they were initially on peg A. You can move disks by obeying these rules:
    a. Only one disk can be moved at a time
    b. A disk can only be placed on a larger disk
    c. All disks (except the disk being moved) must always be on a peg

One way to solve this puzzle is simply to apply the breakdown rules from above:

- **A trivial case**: **n** = 0, no disks to move.
- **A division strategy**: This will consist of three steps:
    1. Move (**n** – 1) disks from A to B
    2. Move disk **n** from A to C
    3. Move (**n** – 1) disks from B from C
- **A combination strategy**: The three steps in combination solves the original problem for **n** disks.

It might be a bit hard to spot the recursion here, but it actually occurs in steps 1 and 3. What we do in these steps is to solve a smaller Towers of Hanoi problem. In step 1, a problem with (**n** – 1) disks is solved, where A is the "source" peg and B is the "target" peg. Step 3 is similar, except that peg B is now "source" and peg C is "target". If we denote the last peg as "extra", we can see that the only difference between the original problem and the smaller problem is that the pegs A, B and C play different roles. In the original problem, peg A is "source", B is "extra" and C is "target", while the problem in step 1 has peg A as "source", C is "extra" and B is "target", and so forth. We can then write up this quite compact code for solving the puzzle:

```
public void TowersOfHanoi(string source, string extra, string target, int n)
{
	if (n > 0)
	{
		TowersOfHanoi(source, target, extra, n - 1);
		Console.WriteLine($"Move disk {n}: {source}->{target}");
		TowersOfHanoi(extra, source, target, n - 1);
	}
}
```

We can then call this method like this:

```
TowersOfHanoi("A", "B", "C", 3);
```

This will print out:

```
Move disk 1: A->C
Move disk 2: A->B
Move disk 1: C->B
Move disk 3: A->C
Move disk 1: B->A
Move disk 2: B->C
Move disk 1: A->C
```

It is indeed possible to write a non-recursive version of Towers of Hanoi, but it is substantially harder and not as intuitive as the above solution.

Even though several problems can be elegantly solved by recursion , it is by no means guaranteed that recursion **efficiently** solves the problem! A famous number sequence known as the **Fibonacci** sequence is defined as:

- **Fibonacci**(**n**) = 1 for **n** = 1, 2 else
- **Fibonacci**(**n**) = **Fibonacci**(**n** – 1) + **Fibonacci**(**n** – 2)

This translates very easily to a recursive method:

```
	public int Fibonacci(int n)
	{
		return (n < 3) ? 1 : (Fibonacci(n - 1) + Fibonacci(n - 2));
	}
```

This looks very similar to the **Factorial** method, but with one extremely important difference: Each call of **Factorial** generates a single recursive call, while each call of **Fibonacci** generates two recursive calls! That may seem insignifcant, but each of

those calls will in turn generate two recursive calls, an so on. In terms of run-time complexity, this has dramatic consequences. While the run-time complexity of **Factorial** is O(**n**), the run-time complexity of **Fibonacci** is O($2^{\mathbf{n}}$)! Remember the logarithm function from earlier? The function $\mathbf{2^n}$ is the inverse function of the logarithm function, meaning that it grows very, very fast:

n	$\mathbf{2^n}$
10	≈ 1.000
20	≈ 1.000.000
30	≈ 1.000.000.000

You should thus see recursion as an additional *tool-in-the-toolbox*, that certainly offers very elegant and compact solutions to certain problems, but can also be somewhat deceptive with regards to efficiency. Consider using recursion if

- The non-recursive solution to the problem is substantially more complex.
- The recursive solution has an efficiency comparable to the non-recursive solution.

LINQ (Language Integrated Query) - fundamentals

The main idea behind **Language Integrated Queries** (or just **LINQ**) is to provide a way of selecting data, which focuses on specifying the data subset to retrieve, without specifying how to retrieve it. This idea is not new; the **Structured Query Language**[2] (SQL) used for retrieving data from relational databases also relies on this idea. In fact, the original syntax used in LINQ was quite heavily inspired by SQL.

Another main idea behind LINQ is to make it as independent as possible from specific data structures. That is, it should not matter if data is stored in e.g. an old-fashioned array, a **List**, a **Dictionary**, or some other data structure. The only requirement LINQ sets on the data structure is that it implements the **IEnumerable<T>** interface. This is a very small interface:

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

The **IEnumerator<T>** interface is also quite small:

```
public interface IEnumerator<out T>
{
      bool MoveNext();
      void Reset();
      T Current { get; }
}
```

You can perceive the **IEnumerator<T>** interface as the absolutely minimal requirement needed for being able to iterate over a collection of data of type **T**. The interface enables you to perform these action with a collection:

- Go to the start of the collection (**Reset**)
- Get the element currently pointed to by the enumerator (**Current**)
- Move forward (if possible) to the next element in the collection (**MoveNext**)

If a collection can implement these two methods and this single property, it becomes possible to iterate over the collection using a **foreach** loop:

[2] https://en.wikipedia.org/wiki/SQL

```
IEnumerable<int> collection = new List<int>(); // e.g.
foreach (var item in collection)
{
      // do something with the item
}
```

Under the covers, the **foreach** loop first calls **Reset**. It then calls **MoveNext**; if **MoveNext** returns **true**, the now pointed-to element is returned by **Current**. **MoveNext** is then called again, until it at some point returns **false**. This indicates that the end of the collection has been reached, and the loop terminates.

The point is that all collection classes in the .NET library implement **IEnumerator<T>,** so we can apply LINQ queries (yes, it should strictly speaking be written as "LIN queries", but the phrase "LINQ queries" is widely accepted…) to any collection, without worrying about its specific type.

Sample Data

Since LINQ is used for selection of data, we need a bit of data to work with. We have defined two simple classes **Movie** and **Studio**, and created collections containing the data given below:

Movie

Title	*Year*	*DurationInMins*	*StudioName*
Se7en	1995	127	New Line Cinema
Alien	1979	117	20th Century Fox
Forrest Gump	1994	142	Paramount Pictures
True Grit	2010	110	Paramount Pictures
Dark City	1998	111	New Line Cinema

Studio

Name	*HQCity*	*NoOfEmployees*
New Line Cinema	Boston	4000
20th Century Fox	New York	2500
Paramount Pictures	New York	8000

The classes **Movie** and **Studio** just contain instance fields and properties corresponding to the columns in each of the tables above. In addition hereto, we also create two collections for storing **Movie** and **Studio** objects:

```
List<Movie> movies = new List<Movie>();
List<Studio> studios = new List<Studio>();
```

Selection

The most fundamental ability of LINQ is to perform "selections" on a given collection of data.

Single-property selection

The first kind of query we address, is a query for selecting a single property from a collection. If we want to select the **Title** property for all objects in the **movies** collection, this is the LINQ query for the job:

```
IEnumerable<string> titles =  from m in movies
                              select m.Title;
```

There are several things to take note of:

- The formatting is intentionally a bit strange; you can write a LINQ query on a single line if you prefer, but the common way to format a LINQ query is to split it into sections according to operators (see below), each section on a new line. Note that there is not a semi-colon at the end of the first line!
- We use a couple of so-called **LINQ operators** (highlighted), which perform certain operations on data. We will dissect these operators in a moment.
- Even though the original data is stored in a **List**, the return type of a LINQ query has the type **IEnumerable**. You can thus iterate over the result, but you cannot e.g. insert an item into it.

A translation of the above LINQ query to human language would read *"from the collection named **movies**, select the property **Title** from each object, and form a collection (of type **IEnumerable<string>**) containing these values"*. We are thus stating a **data source**, and a **set of properties** (in this case just one) we wish to select from each element in the data source. We can then write the query in a more general form:

```
from item in collection
select item.PropertyName;
```

You can hopefully see that **item** is simply a "placeholder" variable, that will be set equal to each of the items in the collection, one by one. This is exactly as we have seen it many times for a **foreach**-loop:

```
foreach (var item in collection)
{
      // ...do something with item
}
```

Returning to the specific query stated above, we can then use the result returned in **titles** in a **foreach**-loop:

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

This will indeed print out the titles – and only the titles – of the movies in our collection. The operation of selecting some of the properties from objects in a collection is also known as **projecting** the objects to a set of properties.

Multi-property selection

The obvious next step is to consider how to select – or project, if you prefer – several properties. Suppose we wish to select both the **Title** and the **Year** property from the **Movie** objects. This complicates the query – and the returned result – a bit:

```
var titlesAndYears =    from m in movies
                        select new {m.Title, m.Year};
```

It is probably not so surprising that we must add **m.Year** after **m.Title**, now that we need to return the **Year** property as well. But why the **new {…}** construction? The problem is that the return type of the query is now something like **IEnumerable<(pair of string and int)>**, which we cannot express in a simple way. By using the **new** operator, we are creating a new object of an **anonymous type**. By anonymous is meant that we have created a new type consisting of an **int** and a **string**, but since this new type only serves as being a return type for this query, we create it *on-the-fly*, and do not bother giving it a name. On top of that, we let the compiler figure out what the return type actually is, by stating that the type of **titlesAndYears** is **var**, i.e. “let the compiler figure it out”…

Even though the specific type of the returned query result is a bit obscure, it is pretty straightforward to use it in a **foreach**-loop:

```
foreach (var item in titlesAndYears)
{
	Console.WriteLine(item.Title + " made in " + item.Year);
}
```

In this fashion, we can create queries for selecting any set of properties we wish to be part of the result.

It may seem surprising that we can refer to named properties in the **foreach** loop above, where we iterate over the query result. Since the query result is a collection of objects of an anonymous type, how do we then know that such an object has e.g. a **Title** property? When an object of an anonymous type is constructed by simple selection as in the example, the property name simply becomes the name of the property the data was selected from, i.e. **Title** and **Year** in the example. In case of a more complex selection, you can specify a custom property name explicitly:

```
var titlesAndYears =    from m in movies
                        select new
                        {
                              Summary = $"{m.Title} made by {m.StudioName}",
                              m.Year
                        };
```

You can then refer to the **Summary** property when iterating over the query result:

```
foreach (var item in titlesAndYears)
{
	Console.WriteLine(item.Summary + " " + item.Year);
}
```

This example also illustrates that "selection" should be understood in a broad sense. You can select simple data like the value of a property, but also "select" more complex data, involving logic or arithmetic expressions. In such cases, LINQ is used as a data **transformation** language.

Selection with collections containing collections

The queries in the above examples return a collection of objects, where each object contains a couple of properties with simple types, like **int** or **string**. It is straightforward to process such a collection, as shown in the **foreach** loops. However, you will often face scenarios where the objects contain non-simple types, like e.g. a collection. We could imagine that the **Movie** class definition also contains a list of **Actor** objects, which can be accessed through an **Actors** property of type **List**<**Actor**>.

A LINQ query to retrieve this data could be:

```
var titlesAndActors =   from m in movies
                        select new {m.Title, m.Actors};
```

This query is perfectly valid, but running it through the standard **foreach**-loop will not produce a very useful result. The below loop

```
foreach (var item in titlesAndActors)
{
      Console.WriteLine(item.Title + " -> " + item.Actors);
}
```

will print something like

```
The Godfather -> System.Collections.Generic.List`1[LINQ01.Actor]
```

This is not in itself surprising, since this is what we in general see, if we try to print a **List** object simply by handing it to **Console.WriteLine**. The fact that this object is now part of a query result does not change this. If we want a more useful output, we must print each element in the **Actors** collection explicitly, like:

```
foreach (var item in titlesAndActors)
{
      Console.WriteLine(item.Title);
      foreach (var actor in item.Actors)
      {
            Console.WriteLine(actor.Name);
      }
}
```

Filtering

If we relate the above queries to the data tables with the sample data, we can perceive selection as picking out vertical “slices” of the data. How do we then pick out horizontal slices of data, i.e. only include data which fulfills certain criteria? This is done by **filtering**.

The LINQ operator for filtering is named **where**. We can extend the selection from above to only include movies from earlier than 1996:

```
var titlesAndYears =    from m in movies
                        where m.Year < 1996
                        select new {m.Title, m.Year};
```

Filtering is thus a logical condition, and only those objects for which the condition is **true** will be included in the result. You can create more complex conditions by using the well-known logical operators:

```
var titlesAndYears =    from m in movies
                        where (m.Year < 1996 && m.Year > 1980)
                        select new {m.Title, m.Year};
```

Note that the order of the operators matter; the **where** operator must be placed before the **select** operator.

Ordering

We can now pick out both horizontal and vertical slices of data, by combining the **where** and **select** operators. These operations preserve the ordering of the elements in the collection. If we wish to order the result according to the value of a specific property, we use the **orderby** operator:

```
var titlesAndYears =    from m in movies
                        where (m.Year < 1996 && m.Year > 1980)
                        orderby m.Year
                        select new {m.Title, m.Year};
```

It is even possible to specify additional properties, like this:

```
var titlesAndYears =    from m in movies
                        where (m.Year < 1996 && m.Year > 1980)
                        orderby m.Year, m.Title
                        select new {m.Title, m.Year};
```

This should be read as "order the result by the value of **Year**; for elements having the same value for **Year**, order by the value of **Title**".

Aggregation functions

It can often be useful to be able to perform various numeric operations on the query result. A set of functions – known as **aggregation functions** – are available for such operations. They can be applied to the variable holding the result of the query, or directly to the query statement. Using the simple initial LINQ query as an example, we can e.g apply the **Count** function (**NB**: note that **Count** is a function here, so we do need to add the **()** after **Count**!):

```
IEnumerable<string> titles =  from m in movies
                              select m.Title;
// This is fine
Console.WriteLine(titles.Count());

// This is also fine
Console.WriteLine((from m in movies select m.Title).Count());
```

Other useful aggregation functions of a numerical nature are **Min**, **Max**, **Sum** and **Average**, which are applied in the same style:

```
Console.WriteLine((from m in movies select m.Year).Average());
```

Some combinations of functions and data types do not really make sense. The below line will not compile:

```
Console.WriteLine((from m in movies select m.Title).Average()); // Error
```

However, this line will – a bit surprising, perhaps – work just fine:

```
Console.WriteLine((from m in movies select m.Title).Max()); // OK
```

Joining

The examples above have all been concerned with selection from a single collection. However, you can construct (more or less) sensible questions which "transcend" a single collection, for instance: *"Return the title of movies produced by studios with headquarters in New York"*. This requires combination of data from both collections, and use of the **join** operator:

```
var joinTitleStudio =   from m in movies
                        join s in studios
                        on m.StudioName equals s.Name
                        where s.HQCity == "New York"
                        select m.Title;
```

The first highlighted line defines the query to work on the "joined" collection, i.e. the collection created by joining **movies** and **studios**. What does it mean to "join" two collections? A "join" is obtained by creating all combinations of an object from the first collection, and an object from the second collection. The "raw" result of joining **movies** (containing five objects, each with four properties) and **studios** (containing three objects, each with three properties) is thus a collection with 3x5 = 15 objects, each with 3 + 4 = 7 properties!

The second highlighted line specifies that out of the 15 objects, we are only interested in the objects for which the **Name** property from **Studio** equals the **StudioName** property from **Movie**. If they are not equal, we cannot really gain useful any information from that object, since the underlying **Movie** and **Studio** objects are not related in the way we are interested in. This constraint reduces the number of objects from 15 to just five, which are exactly those objects we are interested in. To those objects, we apply the **where** clause, and finally select the movie title.

Joining of collections can be extended to involve more than two collections, but then also becomes more complex. If you need to create very complex LINQ queries using **join**, it may be an indication that the data model in general could need an overhaul.

Deferred evaluation

Since a LINQ query only defines what data to retrieve – witout any details about how to retrieve the data – it is not obvious when the query is actually **executed**. A LINQ query is not executed when it is defined; the execution is deferred until the result of the query is needed, typically when you iterate over the query result. If you are not aware of this, you may see unexpected results! The code below illustrates this:

```
// Create the collection
List<Movie> movies = new List<Movie>();

// Enter two objects
movies.Add(new Movie("Se7en", 1995, 127, "New Line Cinema"));
movies.Add(new Movie("Alien", 1979, 117, "20th Century Fox"));

// Define the query
IEnumerable<string> titles =  from m in movies
                              select m.Title;

// Enter three additional objects
movies.Add(new Movie("Forrest Gump", 1994, 142, "Paramount Pictures"));
movies.Add(new Movie("True Grit", 2010, 110, "Paramount Pictures"));
movies.Add(new Movie("Dark City", 1998, 111, "New Line Cinema"));

// Iterate over the query result
foreach (var item in titles)
{
      Console.WriteLine(item);
}
```

Running this code will print out five elements, not two! Also, the query result is not "cached" in any way. If you later add additional **Movie** objects to the **movies** collection, and subsequently iterate over the **titles** variable again, the query result will now also include the recently added objects. If this is not the behavior you want, you can force the query to produce a result immediately. This result will however be a copy of the query result, and will not change after execution. You can force this behavior in a slightly cryptic way, by calling the **ToList** method in the query definition:

```
// Define the query
IEnumerable<string> titles =  from m in movies.ToList()
                              select m.Title;
```

The **ToList** method is actually used quite commonly, to execute a query and immediately map the result into a **List**. Suppose we wanted to implement a simple method that enables you to filter movies according to a given year:

```
List<Movie> FilterOnYear(int year)
{
    var filteredMovies = from m in movies
                         where m.Year == year
                         select m;

    return filteredMovies; // NB: Does NOT work!
}
```

This is – as stated – not a working implementation, since the type of **filteredMovies** is **IEnumerable<Movie>**, not **List<Movie>**. However, the problem is solved by calling **ToList** on the query result, like this:

```
List<Movie> FilterOnYear(int year)
{
    var filteredMovies = from m in movies
                         where m.Year == year
                         select m;

    return filteredMovies.ToList();
}
```

Mapping from **IEnumerable<T>** to **List<T>** is as such trivial, and **ToList** does therefore not take any parameters. A method **ToDictionary** also exist, which can map the query result into a **Dictionary**. This operation is a bit more complex, and we will return to it later on. For now, just be aware that **ToList** is not the only option for query result mapping.

The Fluent syntax

The overall purpose of LINQ is – as mentioned above – to enable us to select data from data structures in a "declarative" way, i.e. only specifying what data we wish to retrieve, without detailing how to retrieve it. The SQL-like syntax used above makes this possible. If you are used to writing SQL queries, it probably feels quite natural to express queries in this way. However, it is also possible to use LINQ functionality in a way that looks more like typical Object-Oriented code, since the LINQ functionality is also available as a set of **methods**, which can be called on collection objects. A simple example of LINQ in this format is given below:

```
List<int> numbers = new List<int>{12, 37, 8, 17};
IEnumerable<int> resultA = numbers.Where(i => i < 15);

// Corresponding SQL-like LINQ query
IEnumerable<int> resultB =    from i in numbers
                              where i < 15
                              select i;
```

It is probably not surprising that the **Where** method has the same purpose as the **where** LINQ operator; it filters out those items in the collection which match a given condition. Here, the condition is specified as a parameter to the **Where** method, in the form of an **anonymous method** of type **Func<int, bool>**. We can also call a **Select** method on a collection, like this:

```
List<Movie> movies = new List<Movie>();
var resultA = movies.Select(m => new {m.Title, m.Year});

// Corresponding SQL-like LINQ query
var resultB =     from m in movies
                  select new {m.Title, m.Year};
```

If you compare this query to the corresponding SQL-like query, you can hopefully see that we have essentially just used the part following the **select** operator as a parameter to the **Select** method, again in the form of an anonymous method.

We can also chain together calls of **Select** and **Where**, like this:

```
var resultA = movies.Select(m => new {m.Title, m.Year})
                    .Where(m => m.Year > 1995);
```

Note that even though this is indeed just traditional method calls, we have used a code layout similar to the SQL-like queries, breaking the statement into two lines of

text. As soon as you start chaining together such method calls, the statements tend to become quite long, and it usually improves readability to break the statement at each method call. It is, however, entirely a matter of taste.

Once we start chaining together such calls, we can in principle make as many calls as we like in a single statement:

```
// More peculiar than useful...
var resultA = movies.Select(m => new {m.Title, m.Year})
                    .Select(m => new { ShortYear = m.Year, m.Title})
                    .Where(m => m.ShortYear > 1995)
                    .Where(m => m.Title.Contains("The"));
```

In this way , you can perform quite sophisticated selections and filterings within a single statement. Note that even though we are back in a style of code looking more like traditional C# code, the code still has a declarative nature; the details of how to obtain the requested data are hidden within the LINQ methods.

How is it possible to chain together method calls in this way? Remember that we initially stated that any collection implementing the **IEnumerable<T>** interface can be queried. The query result has the type **IEnumerable<U>**; that is, the type of the items returned may be different than the type of the items being queried, but it still comes in the form of a collection implementing **IEnumerable**. This enables us to call LINQ methods on the query result as well! That call will – yet again – return a collection implementing **IEnumerable**, and so forth.

This style of syntax is used in several other contexts than LINQ queries, and is generally know as **Fluent syntax**. This syntax can be used on objects implementing so-called **Fluent interfaces**. An example of Fluent interfaces is exactly what we have above. We have some objects available which implement methods of a certain nature; methods, which have the interface itself as return type. This enables us to keep calling such methods in a single chain of method calls, since the return type keeps being of the same type.

There is a small catch, however. When we discussed the **IEnumerable<T>** interface above, we claimed that it is a very small interface, containing just a single method:

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

If this is indeed true (which it is ☺), how can we then call e.g. the **Where** method on a reference of this type, if it is not part of the interface!? This is made possible by a C# language feature known as **extension methods**[3]. In brief, an extension method is as such just an ordinary C# method, but an extension method is declared in such a way that it appears like you have indeed extended a class or interface definition with additional methods and properties.

LINQ methods are thus not defined as part of the **IEnumerable<T>** interface. Where are they then defined? As part of the general .NET class library named **System.Linq**. Consider the below definition of a very small class **LinqTest**:

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

public class LinqTest
{
	private List<int> _numbers;

	public LinqTest(List<int> numbers)
	{
		_numbers = numbers;
	}

	public IEnumerable<int> UseLINQ()
	{
		return _numbers.Where(i => i < 10);
	}
}
```

This code is valid, BUT it hinges on the inclusion of the **System.Linq** class library. If you remove the highlighted **using…** statement, the code is no longer valid, since the call of **Where** on **_numbers** is now invalid. It should, however, also be noted that if you are using the **top-level statement**[4] feature available in .NET version 6 or higher, the inclusion of **System.Linq** is implicit, i.e. you need not explicitly include it yourself.

The above discussion may give the impression that all LINQ methods are of the Fluent kind, i.e. returning references of an **IEnumerable** type. That is definitely not the case. The aggregation functions we saw earlier on (like **Min**, **Max**, **Average**, **Sum**, etc..) all have a numeric return type, and methods like e.g. **First, Last, ElementAt** have – when called on a reference of type **IEnumerable<T>** – the return type **T**. The easiest way of getting an overview of available LINQ methods is simply to declare a variable of a collection type (say, a **List<int>**), and see what pops up in the auto-completion box in

[3] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods
[4] https://learn.microsoft.com/en-us/dotnet/core/tutorials/top-level-templates

the Visual Studio editor, once you type a dot after the variable name. You will find that quite a lot of useful methods are available.

A final question to consider is then:

What LINQ syntax should I prefer?

Given both the SQL-like syntax and the Fluent syntax to choose from, the answer is simply to choose the syntax which you feel enables you to express your queries in the easiest manner. There is no significant difference w.r.t. performance, so choose freely. There might of course be other factors dictating your choice – say, if you work in a company where a code-standard dictates the choice – but apart from that, the choice is yours. The best advice to a beginner is probably to experiment a bit with both syntaxes, until you get a feel of what suits you best.

LINQ (Language Integrated Query) - advanced

The previous chapter should illustrate that LINQ can be utilized to retrieve data from a given data structure, the only prerequisite being that the data structure in question implements the (very small) **IEnumerable<T>** interface. However, thinking of LINQ as a tool merely for data selection is a bit too limited. In general, you should rather think of LINQ as a tool for performing data transformation. We will unfold this idea further in this chapter. Also, we will take a look at some more advanced features of LINQ, in particular the **Aggregate** method, how to utilise LINQ for set-oriented operations on data, and finally on Parallel LINQ (PLINQ) which adds the possibility of parallel execution to LINQ.

LINQ as Data Transformation

It is tempting to percieve LINQ as a C# version of SQL, and there are indeed a lot of common traits. When we perform a selction with SQL, the result is (hopefully) the data we intended to describe with the query. The type of the result itself is worth a bit of attention. A query is performed on one or more tables, and the result itself is also a table. This resulting table may for simple queries be of the same type – i.e. have exactly the same columns – as the table on which the query was performed, but it could in principle be of any type we could imagine, as long as it still matches the general requirements for being a table. Likewise, we can perform LINQ queries on data structures which implement the **IEnumerable** interface, and the result of the query will also be a data structure implementing the **IEnumerable** interface, but the type of the data in the data structure may be different. More specifically, we may execute a query on a data structure implementing **IEnumerable<T>**, and the result will then be a data structure implementing **IEnumerable<V>**, where **T** and **V** need not be identical.

Transformation at object level

Let's look at some queries from the previous chapter again. We assume we have a **List<Movie>** object named **movies** available (which has been populated with some **Movie** objects), and perform the query:

```
var queryResult = movies.Select(m => m.Title);
```

The **movies** object implements **IEnumerable<Movie>** (since a **List<T>** implements **IEnumerable<T>**), but the return type of the query is **IEnumerable<string>**. So, what has the query actually done? It has transformed each **Movie** object (the term "transformed""is not 100 % precise, since the original **Movie** objects in **movies** are of course left unchanged…) into a **string** value.

For this simple example, it is easy still to perceive a LINQ query as pure selection, since we are just picking out a "slice" of data from each object, without altering the data itself in any way. This also holds if we add a selection condition to the query:

```
var queryResult = movies.Where(m => m.Year > 1995).Select(m => m.Title);
```

We have also seen that we can select more than one property from an object:

```
var queryResult = movies.Select(m => new {m.Title, m.Year});
```

This is a bit more towards transformation; the return type is still an **IEnumerable** data structure, but the type of the objects themselves is now described by Visual Studio as an **anonymous type**. This is correct, since we have not defined a class specifically for holding a query result of this type. Still, the individual data in these new objects is the same as in the originale **Movie** objects.

Suppose now that our movie management system has to interface with a different system, which contains the class **MovieInfo**:

```
public class MovieInfo
{
	public string Title { get; set; }
	public int YearSince1900 { get; set; }
	public double TimeHours { get; set; }

	public MovieInfo(string title, int yearSince1900, double timeHours)
	{
		Title = title;
		YearSince1900 = yearSince1900;
		TimeHours = timeHours;
	}
}
```

At some point, there is a need to convert (i.e. transform) a given list of **Movie** objects into a corresponding list of **MovieInfo** objects. This can indeed be done with a LINQ query:

```
List<MovieInfo> miList = movies
      .Select(m => new MovieInfo(
            m.Title,
            m.Year - 1900,
            m.DurationInMins / 60.0))
      .ToList();
```

This is hardly just "pure selection", since we are creating objects of a different type than the queried objects, and the data which goes into the new objects is also subjected to some level of transformation, like e.g. converting from minutes to hours.

Where is the "procedure" for the transformation done by the LINQ query precisely specified? By the argument to the **Select** metod. This argument has a **function type**, which in general will be the function type **Func<T,V>**, which means: A function taking one argument of type **T**, and returning a value of type **V**. In the first simple query, the argument was of type **Func<Movie, string>**. In our last example, the argument was of type **Func<Movie, MovieInfo>**. So, we can – in principle – transform our list of **Movie** objects to anything, as long as we are able to supply a function which describes the details of the transformation.

If you reflect a bit over how a transformation of a set of items in general should proceed, you will probably come up with something like this:

```
public List<V> TransformItems<T, V>(List<T> items)
{
      List<V> transformedItems = new List<V>();
      foreach (T item in items)
      {
            V transformedItem = TransformItem<T,V>(item);
            transformedItems.Add(transformedItem);
      }
      return transformedItems;
}

public abstract V TransformItem<T, V>(T item);
```

The method **TransformItems** describes a general algorithm for data transformation, but one step is situation-specific, and must therefore be declared **abstract** (Hello, **Template Method** design pattern!). In this formulation, a specific implementation of a transformatiom would then be supplied by means of **inheritance**. An alternative formulation of the above is this:

```
public static List<V> TransformItems<T, V>(List<T> items, Func<T,V> transformer)
{
	List<V> transformedItems = new List<V>();
	foreach (T item in items)
	{
		V transformedItem = transformer(item);
		transformedItems.Add(transformedItem);
	}
	return transformedItems;
}
```

Given this alternative formulation, how would we then invoke **TransformItems**? Like this:

```
List<MovieInfo> miList = Transformer.TransformItems<Movie, MovieInfo>(
	movies, m => new MovieInfo(
		m.Title,
		m.Year - 1900,
		m.DurationInMins / 60.0));
```

The syntax is somewhat more verbose here, but the essence of it is very close to the LINQ statement used for transformation above. We, as invokers of the functionality, need to supply two things:

- The data which is to be transformed.
- The algorithm for how to transform each item to a transformed item.

The rest of the transformation process is generic, and can therefore be implemented once and for all in the **Select** method in the LINQ library.

Transformation at the collection level

In the examples above, we have a couple of times ended the LINQ query with a call of the method **ToList**. This is not a method for performing a transformation of the individual items in the collection, but rather a method for transforming the collection data structure in which the items are stored – in this case from a collection implementing **IEnumerable<T>** – to a **List<T>**. Why perform this transformation at all? We will often need to perform additional operations on the collection returned by the query, and the **List** class does offer a much richer set of methods to work with, as compared to the minimal **IEnumerable** interface. The transformation performed by **ToList** is quite simple, and **ToList** itself does not need any arguments.

The LINQ method **ToDictionary** can perform a slightly more complex transformation. Suppose that a collection of objects is available, and furthermore that each object contains a property with a unique value, i.e. a key. For the **Movie** class, we could – perhaps a bit optimistic – assume that the value of the **Title** property is unique. If we often need to look up movies by title, we can do this much more efficiently if the **Movie** objects are stored in a **Dictionary** instead of e.g. a **List**. Invoking the method will look like this:

```
Dictionary<string, Movie> movieDict = movies.ToDictionary(m => m.Title);
```

This method does require one argument, which is the answer to the question "if we want the **Movie** objects to be the values in the **Dictionary**, what should the key value for each object then be?", wrapped up as a function of type **Func<Movie, string>**.

Actions at the object level (without using LINQ)

Suppose that we are not interested in performing an actual transformation of data, but rather want to do something with it, like e.g. print it on the screen. The classic code idiom for this job look like this:

```
foreach (Movie m in movies)
{
      Console.WriteLine(m);
}
```

Could a construction like this also fit into the LINQ paradigm? In terms of using the **Select** metod, we come up short. We could try to write something like:

```
movies.Select(m => Console.WriteLine(m));
```

This is however not valid code! Remember that the return type of **Select** is of type **IEnumerable<T>**, but in this case **T** should be **void**, since **Console.WriteLine** does not return any value (i.e. it has the return type **void**). Using **void** as a type parameter is not permitted in C#. What then? LINQ doesn't really have anything to offer for this kind of task. The **List** class itself, however, contains the method **ForEach**, which is not a LINQ method, but does have some similaries to LINQ methods w.r.t. style. Printing all items in our movies collection can be done like this:

```
movies.ForEach(m => Console.WriteLine(m));
```

We only changed **Select** to **ForEach** in the example, but the latter line of code is perfectly valid, and gets the job done. What is the type of the function argument to **ForEach**? We need to tell **ForEach** what it should do with each item in the list, so we need to suppply a function of type **Action<Movie>**. **Action<T>** is the special case of **Func** where no value is returned by the function, i.e. like **Func<Movie, void>**. This is precisely the type of **Console.WriteLine**. The above line can even be expressed in a more compact form:

```
movies.ForEach(Console.WriteLine);
```

Again, do note that **ForEach** is not a LINQ method, but it is often used together with LINQ queries. Suppose we want to print out our collection of **Movie** objects, but for some reason don't want to add an override of **ToString** to the **Movie** class. This could be done like this:

```
movies
  .Select(m => $"{m.Title}, made in {m.Year}")
  .ToList()
  .ForEach(Console.WriteLine);
```

It may look tempting to remove the call of **ToList**, but this will indeed break the code, since we can not call **ForEach** on a collection which only implements **IEnumerable**. The **ForEach** method can only be called on a **List** object.

The Aggregate Method

The idea of performing transformations using LINQ can be pushed a bit further, by taking another look at the aggregation functions described earlier in this chapter. Some of these functions are conceptually simple, like the **Sum** function:

```
List<int> numbers = [ 21, 8, 14, 45, 30, 9, 22 ];
int sum = numbers.Sum();
```

The **Sum** function simply calculates the sum of the numbers in the collection it is invoked on. This is also a transformation, in this case a transformation of a set of integers to a single integer. However, this transformation has a different nature than the per-object transformations or collection transformations we just discussed. This transformation is from a set of items into a "value", e.g. an integer. This value has a couple of noteworthy features:

- It will usually depend on all of the items in the collection on which it is calculated.
- It will usually – but not always – have a simple type, like an integer or string.

The aggregation methods we saw earlier were all of a numerical nature, like **Sum**, **Max** and **Average**, even though some of them – e.g. the **Max** function – can be invoked on e.g. a collection of strings. But could we not imagine other useful aggregation functions? Suppose we want to calculate the product of a set of integers. There is no **Product** method in the LINQ library, so we could write such a method ourselves:

```
private int Product(IEnumerable<int> numbers)
{
    int product = 1;
    foreach (int value in numbers)
    {
        product = product * value;
    }
    return product;
}
```

Not very complicated… but still not quite as handy as the **Sum** method, which we could just invoke directy on a collection of integers. A more fundamental problem is that we have to implement the steps involved in calculating an aggregate value over and over, if we want to implement additional functions of this nature. What "steps" are those? Consider if we had to implement the **Sum** function by hand as well:

```
private int Sum(IEnumerable<int> numbers)
{
    int sum = 0;
    foreach (int value in numbers)
    {
        sum = sum + value;
    }
    return sum;
}
```

If you compare this to the **Product** function, you can hopefully begin to see a pattern in the implementations: we seem to do the same kind of steps, but with a different "implementation" in each of the specialised cases. Doesn't that sound familiar by now? This seems like another case for the **Template Method** pattern! In the class **AggregateCalculator** defined below, we have tried to generalise the steps seen in the two examples:

```
public abstract class AggregateCalculator<T, V>
{
	public T Aggregate(IEnumerable<V> collection)
	{
		T result = InitialAggregateValue();
		foreach (V item in collection)
		{
			result = UpdateAggregateValue(result, item);
		}
		return result;
	}

	protected abstract T InitialAggregateValue();
	protected abstract T UpdateAggregateValue(T value, V item);
}
```

The method **Aggregate** is intended to being able to calculate any sort of aggregated value for a collection of items, no matter its specific type or the algorithm for the calculation. All we have to do is to "plug in" actual implementations of the two abstract functions. How will this look for e.g. the **Product** function? Like this:

```
public class ProductCalculator : AggregateCalculator<int, int>
{
	protected override int InitialAggregateValue()
	{
		return 1;
	}

	protected override int UpdateAggregateValue(int value, int item)
	{
		return value * item;
	}
}
```

In this case, the implementation of the two methods is quite simple. Using the new class then looks like this:

```
	List<int> numbers = new List<int>{ 21, 8, 14, 45 };
	ProductCalculator prodCalc = new ProductCalculator();
	Console.WriteLine($"Product is {prodCalc.Aggregate(numbers)}");
```

Very nice! What about a function for concatenating a collection of strings into one single string?

```
public class StringConcatenator : AggregateCalculator<string, string>
{
	protected override string InitialAggregateValue()
	{
		return "";
	}

	protected override string UpdateAggregateValue(string value, string item)
	{
		return value + item;
	}
}
```

You get the idea now… Now we don't have to re-implement the fundamental aggregation algorithm over and over. Still, it may seem a bit of a hassle to have to create a new class for each new aggregation function. Let's rewrite the **Aggregate** method using function-type parameters:

```
	public static T Aggregate(
		IEnumerable<V> collection,
		Func<T> initialValueFunc,
		Func<T,V,T> updateValueFunc)
	{
		T value = initialValueFunc();
		foreach (V item in collection)
		{
			value = updateValueFunc(value, item);
		}
		return value;
	}
```

The algorithm is of course the same, but now we don't need to create a new class for using the function. Before seeing an example, let's just be sure that we understand the types of the two functions, The first function **initialValueFunc** is just **Func<T>**, which is fairly straightforward; it should just supply an initial value of type **T**. The second function **updateValueFunc** has the type **Func<T, V, T>**, meaning:

- The function takes two parameters: one of type **T**, and one of type **V**
- The function returns a value of type **T**

The parameters to **updateValueFunc** are the aggregate *value-so-far* (of type **T**) and the next item (of type **V**) in the collection, and the return value (of type **T**) is then the updated aggregate value.

We can now invoke **Aggregate** in this way:

```
List<int> numbers = [ 21, 8, 14, 45 ];
int product = AggregateCalculator<int,int>.Aggregate(
      numbers,
      () => 1,
      (val, item) => val * item);
Console.WriteLine($"Product is {product}");
```

For string concatenation, we get:

```
List<string> words = [ "This ", "is ", "Sparta!"];
string concatStr = AggregateCalculator<string, string>.Aggregate(
      words,
      () => "",
      (val, item) => val + item);
Console.WriteLine(concatStr);
```

So, we now have a generally applicable **Aggregate** method…but surely we cannot be the first to come up with such a method? Definitely not, and the LINQ library does indeed contain an **Aggregate** method, which can be invoked on a collection. The LINQ **Aggregate** method has three overloads, which are exemplified below along with our own **Aggregate** method:

```
List<int> numbers = [ 21, 8, 14, 45 ];

int sosA = AggregateCalculator<int, int>.Aggregate(
      numbers,
      () => 0,
      (val, item) => val + (item * item));
Console.WriteLine($"Sum-of-squares is {sosA}");

int sosB = numbers.Aggregate(
      (val, item) => val + (item * item)); // Accumulator function
Console.WriteLine($"Sum-of-squares is {sosB}");

int sosC = numbers.Aggregate(
      0, // Seed value
      (val, item) => val + (item * item)); // Accumulator function
Console.WriteLine($"Sum-of-squares is {sosC}");

double sosAverage = numbers.Aggregate(
      0, // Seed value
      (val, item) => val + (item * item), // Accumulator function
      val => (val * 1.0)/numbers.Count); // Result selector function
Console.WriteLine($"Sum-of-squares average is {sosAverage}");
```

The three overloads have one, two and three parameters, respectively.

- The first overload only takes an "accumulator function" as parameter, which serves the same purpose as our own update-value function: given the aggregate *value-so-far* and an item, find the new aggregate value.
- The second overload includes a "seed value" parameter, which serves the same purpose as our initial-value function (we leave it as a small exercise to add an overload to our own **Aggregate** method, such that we can specify an initial value directly instead of a function).
- The third overload adds an "afterburner" to the parameter list, if some final processing needs to be done to the aggregated value.

In the example above, we use our own **Aggregate** method along with the three overloads of LINQ **Aggregate** to calculate the sum of the squares of the integer values in **numbers** (the last example actually calculates the average of the squares). Running the example will – a bit surprising, perhaps – not yield the same result for the first three cases! The call of LINQ **Aggregate** without a seed value does not produce the correct result. You could argue that in lack of a seed value, the value 0 (zero) seems like an obvious choice for integers, but that is certainly not always correct. Consider our **Product** aggregate function; what would the result always become, if the initial value was set to zero…? Being able to call **Aggregate** without an initial value seems risky, and is not generally recommendable.

We have now seen that LINQ provides a general mechanism for calculating – in a broad sense – aggregate values of any type, as long as we are able to come up with meaningful definitions of how to aggregate a new value into an existing aggregated value. The algorithm for performing an aggregation as such is thus once-and-for-all embedded in the LINQ **Aggregate** method, which we can then invoke by supplying the missing pieces for a specific aggregation.

Set-oriented operation with LINQ

In addition to functionality for transformation and calculation, LINQ also contains a set of methods for performing set-oriented operations on data collections. A "set" is a mathematical term, which for our purposes is practically the same as a collection. Some operations on collections are a bit tricky to express with a traditional programming style, for instance this operation: Given two sets (i.e. collections) A and B of integers, find the set of integers which are in A but not in B.

```
List<int> setA = [ 2, 6, 12, 9, 3, 7 ];
List<int> setB = [ 12, 8, 3, 71, 13 ];
List<int> setResult = new List <int>();

foreach (int val in setA)
{
      if (!setB.Contains(val))
      {
            setResult.Add(val);
      }
}

Console.WriteLine("In A but not in B: ");
foreach (int val in setResult)
{
      Console.WriteLine(val);
}
```

Again, it's not super-complex code, but it would be convenient not to have to implement this set-operation logic again and again. For this specific operation, LINQ contains the **Except** method:

```
List<int> setA = [ 2, 6, 12, 9, 3, 7 ];
List<int> setB = [ 12, 8, 3, 71, 13 ];

Console.WriteLine("In A but not in B: ");
foreach (int val in setA.Except(setB))
{
      Console.WriteLine(val);
}
```

A prerequisite for having **Except** work correctly is that we can decide if two given items from the set are "equal". We are using **int** values in the example, where the definition of equality is well-defined. Suppose we use **Except** on a collection of **Car** objects instead (NB: **carA** and **carC** are intentionally created with the same values):

```
Car carA = new Car("AA 111", 10000, "Car A");
Car carB = new Car("BB 222", 20000, "Car B");
Car carC = new Car("AA 111", 10000, "Car A");
Car carD = new Car("DD 444", 40000, "Car D");
Car carE = new Car("EE 555", 50000, "Car E");

List<Car> carSetA = [carA, carB, carD, carE ];
List<Car> carSetB = [carB, carC, carE ];
foreach (Car val in carSetA.Except(carSetB))
{
      Console.WriteLine(val);
}
```

Except will in this case return **carA** and **carD**, which is as such correct, if we define two references to **Car** objects as being equal, if and only if they refer to the same object. Two distinct objects containing the same values are thus not considered to be equal. If we wish to change this definition of equality, we can provide an extra argument to **Except**, of type **IEqualityComparer<Car>**. In order to do this, we must then implement a class which implements this interface:

```
public class CarComparer : IEqualityComparer<Car>
{
	public bool Equals(Car x, Car y)
	{
		return x.Plate == y.Plate;
	}

	public int GetHashCode(Car obj)
	{
		return new {obj.Plate, obj.Price, obj.Description}.GetHashCode();
	}
}
```

For now, it is the implementation of **Equals** that is most important. In this case, we define two **Car** objects to be equal if the **Plate** property has the same value. We can now call **Except** with this specific comparer:

```
	foreach (Car val in carSetA.Except(carSetB, new CarComparer()))
	{
		Console.WriteLine(val);
	}
```

With this modification, **Except** will now only return **carD**. The total set of set-oriented methods in LINQ is given below:

Method	Called on	Argument	Description
Except	A	B	Returns items which are in A but not in B.
Intersect	A	B	Returns items which are in both A and B.
Union	A	B	Returns items which are in either A or B.
Distinct	A	(none)	Removes duplicate items from A
Concat	A	B	Concatenates B to A.

Note that the methods **Except**, **Intersect** and **Union** will only return those unique items which fulfill the condition, i.e. the returned set will never contain duplicate items, even though A and B may themselves contain duplicates.

Parallel LINQ (PLINQ)

We can definitely utilize LINQ to implement many kinds of useful functionality in a convenient way. But what about efficiency? We have in a previous chapter discussed a couple of approaches to how to parallelize execution of code. The concept of **tasks** helped us achieve this, and this concept is also part of the LINQ paradigm. This was not the case from the outset of LINQ, and this part of the LINQ library is therefore often referred to as **Parallel LINQ** or just **PLINQ**. All of the PLINQ functionality is however also part of the **System.Linq** namespace, and can therefore be used just as any other part of LINQ.

In the discussion of the **task** concept, the overall goal was to divide the computational effort needed for completing a calculation (in a broad sense) into "chunks", and then execute each "chunk" of computation on some hardware which allows parallel execution. In practice, this hardware will be a modern multicore-CPU, which will typically contain two to 16 processing cores (this will – of course – change over time). PLINQ contains several elements to support this, and we will here only give a brief introduction to the main ideas in PLINQ[5].

Calculation of primes in interval – a candidate for parallelisation

One of the classic examples of parallelization is **prime calculation**. Given a number, determine whether or not the number is a prime number, i.e. only divisible with itself and 1 (one). The algorithm for determining whether or not a specific number is a prime number is not the target for parallelization, however. Instead, it is the task of finding all prime numbers up to a given limit, which we will seek to parallelize. An algorithm for determining if a given number is a prime number is given below:

```
private bool IsPrime(int number)
{
      int limit = Convert.ToInt32(Math.Sqrt(number));
      bool isPrime = true;

      for (int i = 2; i <= limit && isPrime; i++)
      {
            isPrime = (number % i) != 0;
      }

      return isPrime;
}
```

[5] For a more thorough introduction to PLINQ, see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq

This is by no means the cleverest of algorithms for determining this, but this algorithm does have one advantageous feature: it does not depend on the evaluation of other numbers. The consequence is that all such evaluations can in principle be executed in parallel. Let us first see how we can use the **IsPrime** method in conjunction with LINQ, in order to find all prime numbers up to 1,000,000:

```
IEnumerable<int> primes = Enumerable.Range(2, 1000000).Where(IsPrime);
```

Quite compact! The **Enumerable.Range** method generates a collection of all **int** values int the interval [2; 1,000,000], and the **Where** method then simply picks out those values for which **IsPrime** returns **true**.

The interesting question is now: how long does it take to calculate this? In order to measure this, we can use the **StopWatch** class from the .NET class library. A first attempt of measuring the time could look like this:

```
Stopwatch watch = new Stopwatch();
watch.Restart();
IEnumerable<int> primes = Enumerable.Range(2, 1000000).Where(IsPrime);
watch.Stop();
Console.WriteLine($"Primes up to 1,000,000: {primes.Count()}");
Console.WriteLine($"Time spent: {watch.ElapsedMilliseconds} ms");
```

Running this code will yield a very short running time – e.g. a few milliseconds – even though the application does seem to take a little while to complete… If we e.g. try to increase the limit to 10,000,000, the reported running time is still very short, even if the application now spends several seconds before printing the result…? The pitfall here is that the line of code where the query is specified is indeed only a specification of the query, but not an execution of the query! The query is not executed before it really is necessary, which in this case is when **primes.Count()** is called, at which point we have already stopped the stopwatch… Let's make a (seemingly) very small change to the code:

```
Stopwatch watch = new Stopwatch();
watch.Restart();
IEnumerable<int> primes = Enumerable.Range(2, 1000000).Where(IsPrime);
int primesCount = primes.Count();
watch.Stop();
Console.WriteLine($"Primes up to 1,000,000: {primesCount}");
Console.WriteLine($"Time spent: {watch.ElapsedMilliseconds} ms");
```

Running the code now will produce the actual time spent on executing the query, which on the author's hardware is about 700 ms. A always, such measurements of running time are very "noisy", since they depend heavily on the current state of the computer, with regards to other running processes. You should run the code several times to get a reasonable average for the running time.

The AsParallel method (and its brother AsSequential)

So far, so good…but this is just standard LINQ. How do we get PLINQ into play? This is actually quite simple, at least for a simple scenario like this. We need only make one tiny change to the previous LINQ query (since the query now grows beyond the limit for code one-liners, we have changed the formatting a bit):

```
IEnumerable<int> primes = Enumerable.Range(2, 1000000)
      .AsParallel()
      .Where(IsPrime);
```

The addition of the call of **AsParallel** makes all the difference. Running the code will now yield a typical running time of 300-400 ms, i.e. about half of the running time measured for the plain LINQ version. This is consistent with the author's hardware, which includes a dual-core CPU. Increasing the limit to 10,000,000 yields a similar speedup ratio.

What is going on here? The documentation of PLINQ reads that *"…**AsParallel** specifies that the rest of the query should be parallelized, if it is possible"*. So, by calling **AsParallel**, we are instructing the .NET run-time to try to execute the subsequent query in a way which is "as parallel as possible", but we are not providing any further details about how this is to be achieved. Figuring out the details is left to the .NET run-time. This does, however, not relieve us of all responsibility with regards to how to parallelize the task. In order for this to work, the actions done in the parallel part of the query should indeed be independent. This was why it was important that the **IsPrime** method does not rely on other evaluations, i.e. on other calls of **IsPrime**.

Suppose we have a very complex query, where certain parts of the query can be parallelized, while others cannot. PLINQ allows you to "switch off" parallel execution at a certain point in the query, so choosing between serial or parallel execution is thus not restricted to the query as a whole. Switching off parallel execution is done by calling the method **AsSequential**, which will cause the subsequent part of the query to be executed sequentially. You can think of **AsParallel** and **AsSequential** as a closely related pair of methods for controlling the execution model of a query.

The AsOrdered method (and its brother AsUnordered)

In the example above, we have so far only been interested in calculating the number of primes up to 1,000,000. The actual primes do of course need to be found in order to do this, but we have not e.g. printed out the prime numbers themselves. Printing out all primes up to 1,000,000 (there are 78,498 primes up to 1,000,000…) and then inspecting them by hand would be a barbaric job, but if it was done, you would likely find that the order of the prime numbers is somewhat random… If we think of the original query as a sort of "filter", we would probably expect that the order of the numbers would be preserved, i.e. it should never occur that a prime number is lower than the previous prime number. Still, this is likely to happen when executing the parallelised version of the query. But does it matter? Probably not in this case, but it might very well be important in other scenarios. For this reason, it is possible to "augment" the **AsParallel** call with a call of the method **AsOrdered**:

```
IEnumerable<int> primes = Enumerable.Range(2, 1000000)
      .AsParallel()
      .AsOrdered()
      .Where(IsPrime);
```

The call of **AsOrdered** is thus an instruction to PLINQ that the order of the items must be preserved, perhaps at the expense of how efficiently the query can be executed in parallel. Note that **AsOrdered** should not be confused with the **OrderBy** method, which has the almost opposite purpose of reordering a query result according to the function provided by the caller.

In a fashion quite analog to the **AsParallel/AsSequential** method pair, PLINQ also contains an **AsUnordered** method, which can "switch off" ordering at a certain point in a query. Maintaining order in a query result will almost always result in a less efficient query, making it quite useful to be able to turn ordering off whenever it is not strictly needed.

The ForAll method (as compared to the foreach-loop)

A final point in this very brief introduction concerns the processing of a query result. A very typical processing of a query result will involve iterating over the query result by use of a **foreach**-loop:

```
IEnumerable<int> primes = Enumerable.Range(2, 1000000)
      .AsParallel()
      .Where(IsPrime);

foreach (int val in primes)
{
      // Do something with val...
}
```

The query itself will – just as before – not be executed before it really needs to, so the execution will happen as part of the **foreach**-loop. However, since a **foreach**-loop is inherently sequential, the query needs to be completed before the processing done by the loop can proceed. This makes perfect sense in many cases. Suppose we just want to print out the items in the query result by calling **Console.WriteLine**. This is a sort of processing that needs to be sequential, since **Console.WriteLine** does not handle concurrent calls well… Another – perhaps slightly contrived – example could be that we wish to transfer the query result to a collection, e.g. a **List**, like this:

```
IEnumerable<int> primes = Enumerable.Range(2, 1000000)
      .AsParallel()
      .Where(IsPrime);

List<int> primeList = new List<int>();
foreach (int val in primes)
{
      primeList.Add(val);
}
Console.WriteLine($"No. of items in primesList : {primeList.Count}");
```

Running this code will always produce a **List** with exactly 78,498 primes, as it should. Lurking on the sideline, however, is the PLINQ method **ForAll**, which indeed sounds like it has a close relationship with the **foreach**-loop, which it does. The main difference is that **ForAll** allows parallel execution of the processing we wish to perform for each item in the query result. It's fairly easy to rewrite the example above to using **ForAll** instead:

```
List<int> primeList = new List<int>();
Enumerable.Range(2, 1000000)
      .AsParallel()
      .Where(IsPrime)
      .ForAll(primeList.Add);
Console.WriteLine($"No. of items in primesList : {primeList.Count}");
```

It looks pretty innocent, but running this code will yield a count not equal to the expected count of 78,498, and the count will even vary between runs! The reason is that we are using the **List** class in an unintended way. The **List** class is not thread-safe, i.e. the code is not designed to handle multiple concurrent calls of e.g. **Add**. A set of thread-safe collection classes are available in the .NET class library, for instance the **ConcurrentBag** class (yes, a slightly odd name…). Rewriting the example above to using **ConcurrentBag** instead is quite easy:

```
ConcurrentBag<int> primeBag = new ConcurrentBag<int>();
Enumerable.Range(2, 1000000)
      .AsParallel()
      .Where(IsPrime)
      .ForAll(primeBag.Add);
Console.WriteLine($"No. of items in primeBag : {primeBag.Count}");
```

Running this code will always yield the correct count. So, even though PLINQ does make it tantalizingly easy to parallelize your query and/or the post-processing of the query result, you still need to consider if the actions involved are robust enough to being subjected to parallel execution.

Exercises

Exercise	**PRO.3.1**
Project	DataStructureCompare
Purpose	Observe a comparative test of three collection classes, when exposed to various types of use.
Description	The project contains classes that enable you to measure the performance (in terms of run-time) of various operations, when performed on various collection classes.
Steps	1. The class **TimedTester** is a general class for measuring the run-time of a method invocation. Have a look at the class, and notice the very useful class **Stopwatch** from the .NET library. 2. The interface **IDataStructureTester** and the class **DataStructureTesterBase** are general-purpose classes for collection class test. Study the classes, until you feel you understand the general structure of the test. 3. The classes **ListTester**, **LinkedListTester** and **HashSetTester** contain the specific test code for each collection class, i.e. the methods which have a specific implementation for this particular class. Compare how the test is done for each class, i.e. how are the **…Statement** methods implemented for each collection class. 4. **Program.cs** contains the code which executes the test. Get an overview of the test, and try to run the application. 5. Given the discussion about pros and cons of the various collection classes, do the actual run-times reported by the tests make sense? Or are there any surprising results? Note that it is not the absolute run-times that are of interest here – it is the relative measurements of performing the same operation of different collection classes, or different operations on the same collection class. 6. Try to increase the value of **noOfInserts**, in order to increase the number of times the various operations are invoked. What are your expectations to the running time of the various operations, if you e.g. double the value of **noOfInserts**? Do the results match your expectations? If not, try to think about plausible reasons for this.

Exercise	**PRO.3.2**
Project	Palindrome
Purpose	Solve a simple problem using a recursive approach
Description	A **palindrome** is a phrase that reads the same backwards and forwards, like “Racecar” or “Amore Roma”. Note that spaces and upper/lowercase is ignored in this definition. The **Palindrome** project contains an interface **IPalindromeChecker** and a class **PalindromeChecker**.
Steps	1. Study the interface **IPalindromeChecker** and the class **Palindrome-Checker**. They are both quite simple. 2. In **Program.cs**, some test code has been provided. The test code makes it easy to check if your palindrome checker works properly. Take a moment to study the test code itself, and then try to run the application, and see the results. 3. In **PalindromeChecker**, the method **IsPalindromeInternal** is not imple-mented properly. Implement a version that actually works, using a recursive approach. a. Think about how you can divide the original problem into smaller problems, and also about when the problem is trivially solved. b. You will probably need to use the method **Substring**, which can be called on variables of type **string**. c. Once you think the implementation is correct, you can just run the application again, and study the test output. 4. Write a non-recursive version of IsPalindromeInternal.

Exercise	**PRO.3.3**
Project	BackPacking
Purpose	Solve a real-life problem using a recursive approach
Description	The project contains several classes to support solving of the so-called **Backpacking** problem: Given • A **backpack** with limited weight capacity • An **item vault** with a set of **items**, each with a weight and a value Select a set of items from the item vault, such that: • The items can fit into the backpack (i.e. the weight capacity of the backpack is not exceeded). • The items have as high a total value as possible.
Steps	1. Study the single class **BackPackItem** in the **Item** folder. It should be fairly straightforward. 2. Study the classes in the **Containers** folder (start with **BackPackItem-Container**), until you understand their purpose and functionality. 3. Study the classes in the **Algorithms** folder (start with **IBackPackingSolver**), until you understand their purpose and functionality. Where does recursion come into play? 4. Study the code in **Program.cs** – it uses the "stupid" solver to solve a specific backpacking problem. 5. Run the program, and study the output. Are there some obvious indications that the algorithm does not produce the best possible result? 6. Now create a new class **BackPackingSolverSmart**, which should inherit from **BackPackingSolverBase**. Implement a smarter version of **Solve**, i.e. an algorithm which is smarter than the one found in **BackPackingSolverStupid**. The crucial step is to figure out a better way to pick the next item from the vault. See if you can beat the result produced by the stupid algorithm. 7. Once you have implemented a better algorithm, reconsider if the structure for the **BackPackingSolver…** classes is optimal. Could you move some (duplicated) code into the **BackPackingSolverBase** class? 8. Try out other criteria for picking the "best" item of the remaining items, and see if you can beat your first attempt.

Exercise	**PRO.3.4**
Project	LINQDrink
Purpose	Use LINQ queries on a single collection of objects
Description	The project contains a **Drink** class, which is fairly straightforward. In **Program.cs**, a **List** of **Drink** objects is created.
Steps	For each of the below cases, do the following: • Write a LINQ query that returns the specified result. You can use either the SQL-like syntax or the Fluent syntax, or both! • Print out the result of the query, using e.g. a **foreach**-loop: 1. The names of all drinks. 2. The names of all drinks without alcohol. 3. The name, alcohol part and alcohol amount for all drinks with alcohol. 4. The names of all drinks in alphabetical order. 5. The total amount of alcohol in the drinks. 6. The average amount of alcohol in drinks with alcohol. 7. The name and alcohol amount of each drink, grouped by name of alcohol part (NB: We have not discussed grouping in class! Seek information about the **group** LINQ operator online in order to solve this case ☺)

Exercise	**PRO.3.5**
Project	LINQHotels
Purpose	Use LINQ queries on two collections of objects
Description	The project contains the classes **Hotel** and **Room**, which have a *one-to-many* relationship. In **Program.cs**, a **List** of **Hotel** objects and a **List** of **Room** objects are created.
Steps	For each of the below cases, do the following (Note the two helper methods **PrintEnumerableQueryResult** and **PrintNumericQueryResult**): • Write a LINQ query that returns the specified result. You can use either the SQL-like syntax or the Fluent syntax, or both! • Print out the result of the query, using e.g. a **foreach**-loop: 1. The full details of all hotels 2. The full details of all hotels in Roskilde 3. Names of all hotels in Roskilde 4. All double rooms with a price below 400 kr. 5. All double or familiy rooms with a price below 400 kr., in order of price 6. All hotels which start with “P” 7. The number of hotels 8. The number of hotels in Roskilde 9. The average price of hotel rooms 10. The total price of all double hotel rooms

Exercise	**PRO.3.6**
Project	SchoolAdministrationV15
Purpose	Rewrite old-school procedural logic to new, shiny declarative logic using LINQ!
Description	The project is essentially a solution to an old exercise (Pro.2.12, using the **SchoolAdministrationV10** project). In its current form, the project uses classic, procedural logic to answer questions about score averages for individual students, and for all students.
Steps	1. Study the project in its current form. Pay particular attention to the properties **ScoreAverage** in **Student**, and **TotalAverage** in **StudentCatalog**. 2. Run the application, and take note of the output. 3. Now rewrite the logic in the two properties mentioned above, using LINQ where you find it appropriate (Hint: concentrate on the **else**-part of both of the properties mentioned above). 4. Run the application again – the output should of course be exactly the same as before.

Exercise	**PRO.3.7**
Project	LINQCocktails
Purpose	Use LINQ queries – including **join** – on two collections of objects
Description	The project contains an **Ingredient** class and a **Cocktail** class. A **Cocktail** object contains a collection of (references to) **Ingredient** objects. In **Program.cs**, a **List** of **Ingredient** objects and a **List** of **Cocktail** objects are created.
Steps	First study the **Ingredient** class and the **Cocktail** class, to ensure you understand their structure. How does a **Cocktail** object refer to **Ingredient** objects? It might help to create a class diagram to understand the relation between cocktails and ingredients. Then, for each of the below cases, write a LINQ query that returns the specified result, and print out the result of the query (NB: Note that some queries will return collections of collections, so you may need a nested loop to print the query result properly): 1. The names of all cocktails. 2. For each cocktail: The name of the cocktail, and the name and amount of all ingredients 3. For each cocktail: The name of the cocktail, and the name of all ingredients with an alcohol percentage above 10 % 4. For each cocktail: The name and the price of the cocktail (note that the price (per cl.) for an ingredient can be found in the **Ingredient** object collection). 5. For each cocktail: The name and the alcohol percentage of the cocktail.

Exercise	**PRO.3.8**
Project	InvoiceGenerator
Purpose	Use LINQ queries when relevant, in a more complex setup
Description	The project contains several classes, which fall in three groups (details about each class can be found in the source code): • ***DomainModel***, with classes **Constants**, **Discounts**, **Product**, **Order** and **OrderLine** • ***Printing***, with classes **PrintTool**, **InvoiceTool** and **OverviewTool** • The **GUI** class The project is intended to implement screen printing functionality at two levels: printing of an invoice for a specific order, and printing of overview information for the total set of orders. Most of the classes are complete, but most of the functionality in the classes **OverviewTool** and **InvoiceTool** is <u>not</u> implemented; this is <u>your</u> job!
Steps	1. Get an overview of the application. Try to run it, and see what happens. Then take a closer look at the various classes in the project. Once you have an overview of the project, focus your attention on the three classes in the **Printing** folder. 2. First focus on the **OverviewTool** class. The method **PrintOverview** should print overview information about the total set of orders. The method should print (see below for the correct result): a. Total number of orders. b. Total number of order lines. c. Total amount for ordered products, i.e. the sum of the price for all orders (a bit more difficult than **a** and **b**). d. For each product: the description of the product, and the total number ordered of this product (a bit more difficult than **c**)).

```
Total number of orders: 5

Total number of order lines: 17

Total amount ordered: 6132,65 kr.

PRODUCTS ORDERED
--------------------------------------------------------------------------------
Belt                6 ordered.
Gloves              3 ordered.
Keyring             3 ordered.
Shirt               9 ordered.
Sneakers            3 ordered.
Sunglasses          1 ordered.
Sweater             4 ordered.
Wallet              3 ordered.
```

3. Next focus on the **InvoiceTool** class. The method **PrintInvoice** should print an invoice for a specific order. The invoice print should contain (see below for an example of a correct result):
    a. Name, address, zip code and city for customer
    b. For each order line: the description of the product, the quantity ordered, the unit price, the discount percentage, the discount (for one product) and finally the total price for the order line.
    c. The sub-total, defined as the sum of the price of the ordered products without tax
    d. The tax for the order
    e. The total amount, defined as the sub-total plus tax.

```
Anton Jensen
Foldager 17
3520 Farum


DESCRIPTION         QUANTITY    UNIT PRICE  DISC. (%)   DISCOUNT    TOTAL PRICE
--------------------------------------------------------------------------------
Shirt               2           149,00 kr.  10 %        14,90 kr.   268,20 kr.
Sweater             1           299,00 kr.  0 %         0,00 kr.    299,00 kr.
Wallet              1           199,00 kr.  0 %         0,00 kr.    199,00 kr.
Keyring             2           229,00 kr.  0 %         0,00 kr.    458,00 kr.
--------------------------------------------------------------------------------

                                                        SUB TOTAL   1224,20 kr.
                                                        TAX         306,05 kr.
                                                        TOTAL       1530,25 kr.
```

In order to implement the functionality, you should use:

- The available methods in the **PrintTool** base class, and in the two derived classes.
- LINQ (where appropriate)
- Common sense. Think about where you add extra functionality. It is not forbidden to add code to e.g. some of the domain classes.

Exercise	**PRO.3.9**
Project	TicketToBoardingPasses
Purpose	Use LINQ queries for data transformation
Description	The project contains a very simplified model of an Airport Management System. The feature of interest here is generation of boarding passes based on tickets. The two classes **Ticket** and **BoardingPass** represents these two concepts (note that both classes inherit from **CommonInfo**). The (unfinished) functionality for generating boarding passes from tickets is placed in the class **AirportSystem**, specifically in the method **GenerateBoarding-Passes**. Once implemented, the method should be able to convert a set of tickets to a set of corresponding boarding passes. The conversion functionality contains a slight complication with regards to deciding the seating area, which is defined as follows: • All passengers are per default given an "Economy" seating area • However, if a passenger has a bonus point score higher than a "cut-off" limit (which is calculated in the code), the passenger is upgraded to the "Business" seating area.
Steps	1. Get an overview of the project. Make sure you understand the commonalities and differences between **Ticket** and **BoardingPass**. Also inspect **Program.cs**, and see how the functionality is tested. 2. Run the application. You will find that no boarding passes are generated yet. 3. In the **AirportSystem** class, finish the implementation of the **Generate-BoardingPasses** method, so it conforms to the specification above. Try to implement the conversion with the use of the LINQ **Select** method. 4. Test you implementation. You should see that passengers affiliated with DSV or Carlsberg (a total of four passengers) are now seated in Business, while the rest of the passengers are seated in Economy. 5. Try implementing the same functionality in a traditional loop-oriented style, and compare the implementations. What do you prefer?

Exercise	**PRO.3.10**
Project	BoundingRectangle
Purpose	Use LINQ functionality for calculating aggregated values.
Description	The project contains two geometry-related classes **Point** and **Rectangle**. Both classes are fairly simple. The **Rectangle** class does, however, contain the method **CalculateBoundingRectangle**, which can calculate the "bounding rectangle" for two given rectangles. The "bounding rectangle" is defined as the smallest rectangle which can fully contain the two given rectangles.
Steps	1. Take a look at the code in the project, and in particular the **CalculateBoundingRectangle** method in **Rectangle**. It is not crucial that you understand the algorithm as such for calculating the bounding rectangle, but it is important that you understand the way you call the method, and what the method returns to the caller. 2. In **Program.cs**, use LINQ functionality – and perhaps also the **ForEach** method – to implement code for the below: • Printing information about each of the created **Rectangle** objects. • Finding and printing the total area of the rectangles. You could do this in two ways: Either by using the **Aggregate** method, or using the **Sum** method with a little twist… [The expected result is 66.0] • Finding and printing the bounding rectangle for the entire set of created **Rectangle** objects. This will require that you try to perceive that calculation process as an "aggregation" process: given the aggregated value-so-far and the next item, what is then the result of aggregating the next item into the aggregated value? [The expected result is the rectangle [(1,9), (9,2)], with area 56.0)

Exercise	**PRO.3.11**
Project	FateV10
Purpose	Use LINQ functionality for data transformation and aggregation in a more complex scenario.
Description	The project **FateV10** contains a couple of elements from a typical MMORPG, with particular focus on "gear", i.e. items owned by a game participant (a **Hero**) and used in e.g. combat scenarios. An important feature of gear is the **Gear slot**. A gear slot indicates where the gear item "fits" on a Hero. If a gear item has e.g. gear slot set to "hands", it can only be used in that gear slot on the Hero. A Hero may own as many gear items as (s)he wishes, but can only equip (i.e. use) one piece of gear per slot at any time. Another feature of a gear item is the **Power Level**. The Power Level is an indicator of the "strength" of the gear item, i.e. the higher the Power Level, the stronger the gear. In this simple version of the game, the Power Level is only used to be able to rank gear items against each other. A main feature of the game world is the concept of "affinity". Three kinds of affinity exist: Sun, Moon and Stars. The idea is that all gear items have one specific affinity, and can only be used in scenarios which are of the same affinity. Example: If a piece of gear has affinity Moon, it can only be used in e.g. dungeons which have the same affinity. Given a Hero which owns a set of gear items, an interesting question is then: *What is the best possible set of gear items the Hero could equip (from the set of owned gear items), given a specific affinity?* This information could e.g. be used to assess if a Hero (or a group of Heroes) is adequately equipped for entering e.g. a dungeon with a power level restriction (e.g. only Heroes with a power level of at least 600 can enter).
Steps	1. Spend some time getting an overview of the **FateV10** project. Study each class, but pay special attention to the class **Hero**, where most of the work will be done. 2. Run the application. Initially, it will just print out information about a single Hero (this is done in **Program.cs**), and the gear owned by the Hero. Some of the values for power level will not be correct at this point. 3. Return to the **Hero** class. The first method to implement is the **BestGearInSlot** method (the version indicated in the code). Implement it according to the description in the code. Once you have implemented it, you can try to

	write a small test for it in **Program.cs**. Note that once you have implemented the method, the two other overloads of the method should work as well. 4. The next method to implement is the **PowerLevel** method (the version indicated in the code). Implement it according to the description in the code. Once you have implemented it, you can try to write a small test for it in **Program.cs**. Note that once you have implemented the method, the two other overloads of the method should work as well. 5. Follow the same strategy to implement the **BestGear** method. 6. Follow the same strategy to implement the **BestGearBuild** method. 7. Feel free to extend the application further, and perhaps use it to try to answer more complex questions: One such question could be: *If a Hero has picked up N Gear items, what would the average Power Level for his best build be (in general, and also for a specific affinity)?*

Exercise	**PRO.3.12**
Project	PrimesSieve
Purpose	Use set-oriented LINQ functions.
Description	The project only contains a single (besides the classes for testing, etc..) class **PrimesCalculator**, with a single public (static) method **NoOfPrimesUpToN**. This method is intended to calculate the number of primes in the interval from 2 up to a given limit. The algorithm (to be) implemented in **NoOfPrimesUpToN** uses the principle called *Sieve of Eratosthenes* (see https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). The algorithm starts out with a full list of numbers from 2 to an upper limit, and then successively remove values from this list. The main principle is to remove multiples of known prime numbers: If e.g. 7 is known to be a prime number, we can remove all multiples of 7 (i.e. [14, 21, 28, 35, 42, …]) from the list, since they can obviously not be prime numbers.
Steps	1. The method **NoOfPrimesUpToN** in **PrimesCalculator** uses the above principle, but in a list-based manner. Study the (incomplete) implementation, and be sure you understand how the algorithm works. **NB**: If you run the application *as-is*, it will be stuck in a infinite loop, due to the incomplete implementation. 2. One you feel you understand the algorithm, implement the two steps which are left out. Both steps can be implemented quite neatly using the set-oriented methods from LINQ. 3. Run the application; the test is initially set up to test a single case, where the upper limit is set to 1.000 (see **Program.cs**). The expected result of this case is 168 primes. 4. Once you get the correct result, you can try out a few more cases, as indicated in **Program.cs**. The expected results for the three additional cases are 10.000 -> 1.229, 100.000 -> 9.592, 1.000.000 -> 78.498. **NB**: Be aware that the last case may take a few seconds to complete. 5. (For the daring): Run a case where the upper limit is set to 10.000.000, while having the **Diagnostic Tools** window (found under **Debug/Windows**) open. Observe how the memory consumption of the application develops over time.

Exercise	**PRO.3.13**
Project	ModernFamily
Purpose	Use a number of LINQ functionalities in a more complex scenario
Description	The project is intended to model individuals and their relation to each other. The three central classes in the project are **Person**, **Relation** and **FamiliyTreeNode**: • **Person**: A quite simple representation of a single person. • **Relation**: Represents a relation between two persons A and B. The possible types of relations are found in the enumeration **RelationType**. • **FamilyTreeNode**: Intended to represent an entry in a family tree, for one specific person. The node will have references to parents, children and spouse for that specific person. The challenge is now to use LINQ funtionality to calculate various sets of persons, as specified below.
Steps	1. Study the three classes mentioned above. **Person** and **Relation** are fairly straightforward, while **FamilyTreeNode** is a bit more complex. 2. Open **Program.cs**. In **Program.cs**, we first create a number of **Person** objects, a number of **Relation** objects, and populate two collections with these objects. **Program.cs** also contains a number of helper methods in the *Helper methods…* section. Also take a look at those methods, since you will need to use them in the subsequent steps. 3. In *Step 1*, we imagine that Alice and her (current) spouse wants to throw a party for *all persons who are friends with both Alice and her spouse*. Implement code for this in the place indicated in the code. 4. In *Step 2*, we imagine that Carol and her (current) spouse wants to throw a party for *all persons who are friends with either Carol or her spouse (or both)*. Implement code for this in the place indicated in the code. 5. It turns out that a single person in the "friends" group (i.e. those persons who are only part of "friend" relations) is not invited to either of the two parties. Write code in *Step 3* to find that single person. Also, it is decided that this "uninvited" person should go out on a "secret date" with that person who is a friend with the "uninvited" person. Write additional code in *Step 3* to find this person as well. 6. In *Step 4*, we wish to generate **FamilyTreeNode** objects for all persons in the scenario. That is, we wish to generate a single, "standalone" family tree object for each person, i.e. the family tree objects are not supposed to be connected to each other. In order to do this, you need to add code in two places: a. Implement the helper method **GenerateFamilyTreeNode** correctly. b. Implement *Step 4* in the place indicated in the code.

	7. [Challenge] Now that you can create stand-alone family tree nodes, see if you can figure out how to put a set of disconnected family tree nodes together to form "real" family trees. In order to test this, you may need to add further **Person** and **Relation** objects to the scenario.