|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Programming, Part III |

|  |
| --- |
| By Per Laursen  01-07-2018 |

[Introduction 2](#_Toc517341632)

[Run-time complexity 3](#_Toc517341633)

[Data Structures revisited 7](#_Toc517341634)

[The LinkedList class 7](#_Toc517341635)

[The Queue and Stack class 9](#_Toc517341636)

[The HashSet class 10](#_Toc517341637)

[Recursion – iteration without loops 11](#_Toc517341638)

[LINQ (Language In-Line Query) 16](#_Toc517341639)

[Sample Data 17](#_Toc517341640)

[Selection – single property 18](#_Toc517341641)

[Selection – several properties 19](#_Toc517341642)

[Selection – collections containing collections 20](#_Toc517341643)

[Filtering 21](#_Toc517341644)

[Ordering 22](#_Toc517341645)

[Aggregation functions 22](#_Toc517341646)

[Joining 23](#_Toc517341647)

[Deferred evaluation 24](#_Toc517341648)

[Exercises 26](#_Toc517341649)

[PRO.3.1 26](#_Toc517341650)

[PRO.3.2 27](#_Toc517341651)

[PRO.3.3 28](#_Toc517341652)

[PRO.3.4 29](#_Toc517341653)

[PRO.3.5 30](#_Toc517341654)

[PRO.3.6 31](#_Toc517341655)

[PRO.3.7 32](#_Toc517341656)

# Introduction

The previous chapters have armed us with tools to create rather sophisticated appli­cations, with feature-rich GUIs and 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 called **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 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 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 the 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 In-Line 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
* 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 is not so interesting, 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 (often called 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**(**n2**)

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(**n2**). Even though the O(**n**)-algorithm might actually be slower than the O(**n2**)-algorithm for small values of **n**, the O(**n**)-algorithm will at some point beat the O(**n2**)-algorithm.

In this way, we consider an O(**n2**)-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:

**T** = **O**(**1**)

This may look a bit weird the first time you see it, but it 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**p** elements. This gives us a rela­tion between the problem size and the running time:

**n** = c \* 2**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:

**T** = **O**(log(**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 ≈ 210 | ≈ 10 |
| 1.000.000 ≈ 220 | ≈ 20 |
| 1.000.000.000 ≈ 230 | ≈ 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.

# 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. 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 called E, and you wish to insert it just after an element called X. Before insertion, element Y follows after element X:

Y

E

X

Insertion of E requires just two steps:

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

E

Y

X

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 pro-vide 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 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** game. The problem to be solved in this game is as follows:



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:
   1. Only one disk can be moved at a time
   2. A disk can only be placed on a larger disk
   3. 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 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**n**)!

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 In-Line Query)

The main idea in **Language In-Line Queries** (or just **LINQ**) is to provide a way of selecting data, which focuses on speci­fying the data subset to retrieve, without spefiying how to retrieve it. This idea is not new; the **Structured Query Language[[1]](#footnote-1)** (SQL) used for retrieving data from relational databases also relies on this idea. In fact, the syntax used in LINQ is 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** 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>** interfaceas the absolutely minimal require­ment needed for being able to iterate over a collection. 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 single property, it becomes possible to iterate over the collection with a **foreach** loop:

**IEnumerable<int> collection = new List<int>();**

**foreach (var element in collection)**

**{**

**// do something with the element**

**}**

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**

|  |  |  |
| --- | --- | --- |
| ***StudioName*** | ***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 to store the **Movie** and **Studio** objects:

**List<Movie> movies = new List<Movie>();**

**List<Studio> studios = new List<Studio>();**

## Selection – single property

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.
* We use a couple of so-called **LINQ operators** (highlighted), which perform cer­tain operations on data. We will dissect them 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 element into it.

A translation of the above LINQ query to human language would read “from the col­lec­tion called **movies**, select the property **Title** from each object”. In other words, we are 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 element in collection**

**select element.PropertyName;**

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

**foreach (var element in collection)**

**{**

**// ...do something with element**

**}**

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

**foreach (var element in titles)**

**{**

**Console.WriteLine(element);**

**}**

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 called to **project** the objects to a set of properties.

## Selection – several properties

The obvious next step is to consider how to select – or project to – several properties. Suppose we wish to select 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 result is a bit obscure, it is pretty straight­forward to use it in a **foreach**-loop:

**foreach (var element in titlesAndYears)**

**{**

**Console.WriteLine(element.Title + " made in " + element.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 the object of an anonymous type is constructed by “trivial” 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 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 element in titlesAndYears)**

**{**

**Console.WriteLine(element.Summary + " " + element.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.

## Selection – collections containing collections

The queries in the previous examples return a collection of objects, where each object contains a couple of properties with simple types, like **int** or **string**. It is straightfoward 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 loop

**foreach (var element in titlesAndActors)**

**{**

**Console.WriteLine(element.Title + " -> " + element.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 element in titlesAndActors)**

**{**

**Console.WriteLine(element.Title);**

**foreach (var actor in element.Actors)**

**{**

**Console.WriteLine(actor.Name);**

**}**

**}**

## Filtering

If we relate the above queries to the data tables with the sample data, you 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 – called 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:

**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());**

This line will on the other hand work just fine:

**Console.WriteLine((from m in movies select m.Title).Max());**

## Joining

The examples above have all been concerned with selection from a single table. How­ever, you can construct (more or less) sensible questions which “transcend” a single table, 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.StudioName**

**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 two **StudioName** properties (one from **Movie**, one from **Studio**) are equal. 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 two 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 element in titles)**

**{**

**Console.WriteLine(element);**

**}**

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;**

There is more to LINQ than described in this chapter, for instance methods for finding intersections, unions etc. between data sets, and also more sophisticated methods for process­ing data. As usual, there are plently of sources online providing more in-depth treatments of LINQ.

# 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 **Palin­drome­Checker**. 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. Try to run the application, and see the results. 3. In **PalindromeChecker**, the method **IsPalindromeInternal** is not imple­men­ted properly. Implement a version that actually works, using a recursive approach.    1. Think about how you can divide the original problem into small­er problems, and also about when the problem is trivially solved.    2. You will probably need to use the method **Substring**, which can be called on variables of type **string**.    3. Once you think the implementation is correct, you can just run the application again, and study the test output. |

|  |  |
| --- | --- |
| **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 **BackPackItem­Container**), 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 * 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 * 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 an ER-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. |

1. https://en.wikipedia.org/wiki/SQL [↑](#footnote-ref-1)