C# IEnumerable vs List and Array

When you should use one over the other and what I learned trying to work this out…

**TL;DR**

It depends! As is often the case with programming.

IEnumerables delay execution of the query until the code is looped over or a value type is extracted. Basically IEnumerables have a method to get the next item in the collection, so they look at items “one at a time”, they don’t need the whole collection to be in memory and don’t know how many items are in it, foreach just keeps getting the next item until it runs out. IEnumerables also help ensure immutability, as you are always querying the source there are no unintended side effects.

Lists and Arrays create objects in memory and allow access to a whole lot of methods associated with those types ( [Lists](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netframework-4.8#methods) | [Arrays](https://docs.microsoft.com/en-us/dotnet/api/system.array?view=netframework-4.8#methods) ). Lists and Arrays know how many items are in the collection and have more acknowledgment of their whole overall structure.

When you might use an IEnumerable:

* A massive database table, you don’t want to copy the entire thing into memory and cause performance issues in your application.

When you might use a List or Array:

* You need the results right away and are possibly mutating the structure you are querying later on.

**My Story**

My name is Ben Muller and I am an aspiring software engineer. I work at MYOB in Sydney, Australia and I have been in the [Future Makers Academy](https://www.myob.com/au/careers/graduate-program" \t "_blank) (“FMA”) for just over two months. As part of the FMA protégés have the incredible opportunity to present their code at milestones referred to as quorum reviews (“QR”). At the QR a protégé presents a coding challenge and walks through their solution, business logic, design choices and overall understanding of C# and the language design principles associated with it. The QR is done in front of six senior developers, a principal developer ( lead developer ) and the FMA coordinator.

The QR went well, I was definitely a little bit nervous at first but after a few minutes I realised that everyone in the room was here to help me grow and learn, not to shoot me down for anything I had done wrong. It also dawned upon me that this is as close to a once in a lifetime opportunity as they come, a chance to learn from the devs in the room and see all of their opinions on how I can better my craft. I often find myself seeking out others opinions on my code to see if I am doing things in the “correct way”, and here I am with the vehicle to do that with eight different opinions.

At work I use [Jetbrains Rider IDE](https://www.jetbrains.com/rider/" \t "_blank) for all of my C# development, it is incredibly powerful and user friendly once you get the hang of it. It is so smart that it can often recommend certain types of refactoring to make to your code. Starting out as a developer this can often be a trap. You find yourself accepting Riders refactoring suggestions and not understanding why. I was slightly guilty of this. In my QR I received some feedback to go away and make sure I reinforce my understanding of Collections at a deeper level and in particular when to use each one and the differences between them eg when to use IEnumerable vs List vs Array.

After the QR I set out to up skill in the topic and gain a much deeper understanding. I found that there was a lack of straight forward up to date information on the topic, so I decided to write a post about it for future aspiring developers in my position.

**What I learned**

When I started looking into when to use IEnumerable vs List vs Array this was one of the first things I came across in a SO post:

*“IEnumerable describes behaviour, while List is an implementation of that behaviour.*

*When you use IEnumerable, you give the compiler a chance to defer work until later,*

*possibly optimising along the way.*

*If you use ToList() you force the compiler to reify the results right away.”*

At first, this made a bit of sense to me, but I didn’t really understand the statement about:

“When you use IEnumerable, you give the compiler a chance to defer work until later, possibly optimising along the way.”

So let’s explore this a little more with a bit of code…

* In the above code we have some sample names in a List.
* Next we have a LINQ query to find the words that have more than 5 letters.
* Then we reassign the first word in the names list to benjamin.

Now, what would you expect this code to output to the console?

I know that at first I thought the program would output:

**alexia sumanth**

Then I thought further about the SO post.

*“When you use IEnumerable, you give the compiler a chance to defer work until later,*

*possibly optimising along the way.”*

Defer work until later, hey?

So what is actually happening here?

When you run a LINQ it returns an IEnumerable\<T>, so it doesn’t actually save the results of the query into memory. It just saves the actual query and delays the execution until later when it is looped over or a value type is extracted from the variable it is saved in.

So in this specific example when the code is iterating over the **moreThanFiveLetters** IEnumerable it executes the query on the **names** List and its current contents at the time of the loops’ execution. In the above example the loop is executed after the **names** list has had the first element assigned to “benjamin”.

This means that the above code actually outputs

benjamin  
alexia  
sumanth

to the console.

To really drum home the concept let’s consider the following code ( it is the same as the above code, only I have cast the IEnumerable into a List with .ToList() ):

In the above example what do you think is output to the console?

If you guessed:

alexia   
sumanth

then you are correct!

In this example casting the IEnumerable via **.ToList()** means that the **moreThanFiveLetters** variable is now a List which is a reference type and **moreThanFiveLetters** is an instance of the List type.

That is a bit of a mouthful. Think of it like this, List is a class and it has a bunch of properties and methods associated to that class and therefor those properties and methods are available to **moreThanFiveLetters** when we cast it to a list.

Because **moreThanFiveLetters** is now an instance of a List and its own object, when we execute:

names[0] = “benjamin”;

**moreThanFiveLetters** takes up its own space in memory and technically doesn’t know about the **names** variable and therefor is unaffected.

So when we iterate over **moreThanFiveLetters**,

alexia  
sumanth

is written to the console.

**Key findings**

* IEnumerable is read-only and List is not.
* IEnumerable types have a method to get the next item in the collection. It doesn’t need the whole collection to be in memory and doesn’t know how many items are in it, foreach just keeps getting the next item until it runs out.
* List implements IEnumerable, but represents the entire collection in memory.
* LINQ expressions return an enumeration, and by default the expression executes when you iterate through it using a foreach, but you can force it to iterate sooner using **.ToList(**) or **.ToArray()**.

**Useful links**

* [Multiple enumeration](http://twistedoakstudios.com/blog/Post7694_achieving-exponential-slowdown-by-enumerating-twice)
* [IEnumerable vs List](https://stackoverflow.com/questions/3628425/ienumerable-vs-list-what-to-use-how-do-they-work/3628462#3628462)
* [When To Use IEnumerable, ICollection, IList And List](https://www.claudiobernasconi.ch/2013/07/22/when-to-use-ienumerable-icollection-ilist-and-list/)

I hope this helps some people understand a little more about the difference between the levels of abstraction of a List and an IEnumerable. There is a lot more to learn from this topic, but this is a tricky starting point and one that took me a little while to get my head around.