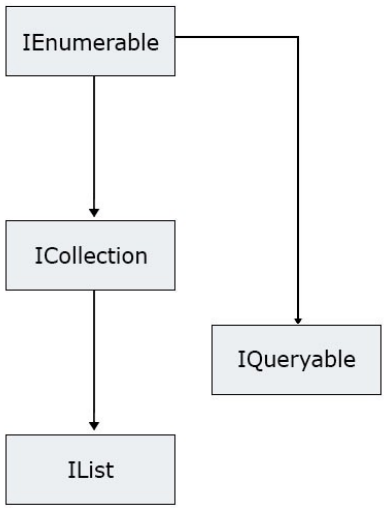
1. **IEnumerable vs ICollection vs IList vs IQueryable**



INHERITANCE

IEnumerable (namespace: System.Collections) -

- Holds data.
- Data can be iterated in forward only mode.
- Data not editable (immutable).
- Supports where clause.
- To fetch the record count data needs to be iterated. So count functionality will need O(n).
- Best for query data from in-memory collections like List, Array etc.
- It does not run a query until it is requested by iteration or enumerator. (Deferred Execution).

IEnumerable interface

```
public void IEnumerableExample()
{
    IEnumerable<EmployeeDetail> empDetail =
                        dbContext.EmployeeDetails
                        .Where(emp=>emp.Employee_Name.StartsWith("j"));
    var result = empDetail.Take(2);
}
```

SQL Query for the above LINQ query

```
SELECT
    [Extent1].[Eemployee_Number] AS [Eemployee_Number],
    [Extent1].[Employee_Name] AS [Employee_Name],
    [Extent1].[Department_Id] AS [Department_Id],
    [Extent1].[Salary] AS [Salary]
    FROM [dbo].[EmployeeDetails] AS [Extent1]
```

```
WHERE [Extent1].[Employee_Name] LIKE 'j%'
```

In the above example, we can see that IEnumerable filters did not execute on the server side and it fetched all the records which are matching all conditions. In the above query, the .Take(2) filter was not applied on the server side. The SQL query gets all records which match the where condition. **So, the filter is not applied on the server side but first, it fetches all the records from the server and applies on the client side.**

ICollection (namespace: System.Collections) -

- IEnumerable + functionality of Add, Remove, Update in the list.
- Can use Count() to directly fetch the record count in O(1).
- ICollection supports enumerating elements, filtering elements, adding, deleting, updating existing elements and getting counts of elements.
- Primarily used where EF table relationships are defined and we use this using the virtual keyword.
- It does not support indexing as IList does.
- ICollection<T> is a generic interface from which all generic collection based classes & interfaces are derived e.g. IList<T>,Stack<T>,Queue<T> etc.
  NOTE - The functionality of ICollection and IList are almost same with a small difference. IList works with an additional functionality when along with iteration we also need indexed based access as well.

IList (namespace: System.Collections) -

- ICollection + Insertion + Removal of elements from middle of list.
- We can use foreach to iterate.
- Element counting can be done in O(1).
- Supports indexing or getting elements at a particular location.
- As IEnumerable, IList is also an in-memory collection and helps query data.
- It supports deferred execution but not further filtering.

The IList interface is useful when you want to perform any operation like get an item, add a new item, or remove an item at a specific index from the collection.

LINQ Query for IList<T>

```
public void IListExample()
{
    IList<EmployeeDetail> empDetail
            = dbContext.EmployeeDetails
                       .Where(emp=> emp.Employee_Name.StartsWith("j"))
                       .ToList();

    empDetail.Add(new EmployeeDetail
    {
       Eemployee_Number = 10012,
       Employee_Name = "Peter",
```

```
            Department_Id = 3,
            Salary = 45000
        });

        empDetail.RemoveAt(4);
    }
```

In the above example, we have fetched the records from the database based on a condition. After that, we have added a new record to the same list and removed a record based on a specific index.

IQueryable (namespace: System.Linq) -

- Extends from IEnumerable.
- Generates LINQ to SQL to execute over DB.
- We can use List.AsQueryable().
- Element counting can be done in O(1).
- Supports custom query and deferred execution.

LINQ Query for using IQueryable.

```
public void IQueryableExample()
{
      IQueryable<EmployeeDetail> empDetail =
                  dbContext.EmployeeDetails
                  .Where(emp => emp.Employee_Name.StartsWith("j"));
      var result = empDetail.Take(2);
}
```

SQL Query for the above LINQ query,

```
SELECT TOP (2)
    [Extent1].[Eemployee_Number] AS [Eemployee_Number],
    [Extent1].[Employee_Name] AS [Employee_Name],
    [Extent1].[Department_Id] AS [Department_Id],
    [Extent1].[Salary] AS [Salary]
    FROM [dbo].[EmployeeDetails] AS [Extent1]
    WHERE [Extent1].[Employee_Name] LIKE 'j%'
```

In the above example, we see that IQueryable executed all the filters on the server-side and fetched the records that are matching all conditions and filters. In the above query, .Take(2) filter is applied on server-side and SQL Query takes the Top (2) records. **So, the filter is applied on the server side.**

Summery

Also IEnumerable and IQueryable always hit the data source whenever you access them.That is if you are accessing SQL database, on each access of IEnumerable and IQueryable database will be accessed. Use this when data can change and you want fresh data always to be queried

from data source.But incase you don't want to go to database every time make sure you convert the IEnumerable or IQueryable to a IList as shown below and do further operation on IList.

```
IList<EmployeeProfile> profileEnumerable =
                    dc.EmployeeProfiles
                    .AsEnumerable()
                    .Where(x => x.EmployeeJobSector == "In Consulting")
                    .ToList();
```

Type	**IEnumerable**	**ICollection**	**IList**	**IQueryable**
NameSpace	namespace: System.Collections	namespace: System.Collections	namespace: System.Collections	namespace: System.Linq
Data Fetch Process	This can be used to query data from DB directly but if there are any conditions then the conditions will not be applied on the DB. All the data will be fetched from the DB to an in-app collection and then the filter will be applied.	Data fetch process is the same as IEnumerable.	Data fetch process is the same as IEnumerable.	If data is fetched from DB using a filter. Then the DB execution will run with the filter condition. So at client side we will receive filtered data.
Data Read	Forward Only	Forward Only	Forward Only + Read by Index	Forward Only
Data Operation	Read Only **from DB**. To read from DB we use **GetEnumerator()** and this method will be available to inherited interface	Add, Remove, Update. Read using GetEnumerator() of IEnumerable	Add, Remove, Update + Use of Index to do operations in the middle of the collection. Read using GetEnumerator() of IEnumerable	Read Only **from DB.** Read using GetEnumerator() of IEnumerable.

Data Count	Move through all the count element - O(n)	Dedicated Count() - O(1)	Dedicated Count() - O(1)	Dedicated Count() - O(1)
Objective	Read from DB	Operation on the DB data	Indexed operation on the DB data	Read from DB
Mixed Operation	To operate on the data once data is fetched, the data is needed to be converted to ICollection or IList for further operation.	Operation on DB fetched data.	Indexed operation on DB fetched data.	To operate on the data once data is fetched, the data is needed to be converted to ICollection or IList for further operation.

Popular Methods
GetEnumerator() - Returns an enumerator that iterates through a collection.

Extension Methods
AsParallel(IEnumerable) - Enables parallelization of a query.
AsQueryable(IEnumerable) - Converts an IEnumerable to an IQueryable.

IEnumerable vs List
If you iterate elements from IEnumerable or List only one time, so both are the same, no differences.
If you iterate elements many times, or if you get an one element multiple times, IEnumerable will be slow.
Because of IEnumerable it does not contain result, so that it creates a new instance of result when it's used. It means, when you access the same element multiple times, each will be a different object!

Ref: https://stackoverflow.com/questions/19689328/why-ienumerable-slow-and-list-is-fast

//Code 1

```
var dic = new Dictionary<int, string>();
for (int i = 0; i < 20000; i++)
{
      dic.Add(i, i.ToString());
}

var list = dic.Where(f => f.Value.StartsWith("1"))
                  .Select(f => f.Key); //.....................1
```

```
Console.WriteLine(list.GetType());

var list2 = dic.Where(f => list.Contains(f.Key))
                   .ToList(); //.............................2

Console.WriteLine(list2.Count());
```

Here (1) is IEnumerable and (2) is List

//Code 1 - IEnumerable - O(n^2)
//The logic behind the second Where without ToList looks like this:
// The logic is expanded for illustration only.

```
var list2_a = new List<KeyValuePair<int, string>>();
foreach (var d in dic)
{
    var list_a = new List<int>();
    // This nested loop does the same thing on each iteration,
    // redoing n times what could have been done only once.
    foreach (var f in dic)
    {
        if (f.Value.StartsWith("1"))
        {
            list_a.Add(f.Key);
        }
    }
    if (list_a.Contains(d.Key))
    {
        list2_a.Add(d);
    }
}
```

Because when we don't have the .ToList() call, the list2 instantiation will iterate through the whole list enumerable for every item in the dictionary. So you go from O(n) to O(n^2) if you use deferred execution.

//Code 2 - List - O(2n)
//The logic with ToList looks like this:
// The list is prepared once, and left alone

```
var list_b = new List<int>();
foreach (var f in dic)
{
    if (f.Value.StartsWith("1"))
    {
        list_b.Add(f.Key);
```

```
    }
}
var list2_b = new List<KeyValuePair<int, string>>();
// This loop uses the same list in all its iterations.
foreach (var d in dic)
{
    if (list.Contains(d.Key))
    {
        list2_b.Add(d);
    }
}
```

As you can see, the ToList transforms an O(n^2) program with two nested loops of size n into O(2*n) with two sequential loops of size n each.

It is caused by deferred execution. IEnumerable does not have to be a static collection. Generally it is some data source (dic in your case) + all methods and expressions (Where, Contains, etc.) that lead to the final set. .ToList() executes all these methods and expressions and generates the final result.
So in case you use ToList() it generates a standard .NET List (array of integers) and does all operations on that list. If we don't call ToList() (or any other To-method) the IEnumerable can be enumerated several times.

IList vs List
List is the specialized implementation of the IList interface.

How to choose:

IEnumerable	IEnumerable is useful when we want to get data from DB using simple query without filters + in-process memory process to iterate the data set. Note - The in-memory process holds good if we only have a single iteration through foreach or other loops. If we have more than one iteration or multiple foreach then in memory we need to reiterate and allocate memory for every operation. Moreover if there are filters then this will execute a query in the DB without filters and then filter would be applied on this data on the backend code. No Count(), or other Upsert operations possible. IEnumerable is suitable for LINQ to Object and LINQ to XML queries. IEnumerable supports deferred execution. IEnumerable doesn't support lazy loading.
IQueryable	Best when we deal with DB queries with filters. But they cannot use an in-memory process to iterate here so need to convert it to another form of Collection using ToList(). It creates a query using an Expression Tree.

	IQueryable is suitable for LINQ to SQL queries. IQueryable supports deferred execution. IQueryable supports lazy loading. Hence it is suitable for paging like scenarios.
ICollection	ICollection is better if we have any non-indexed operation and we need to switch between List, Queue, Stacks. But it cannot be used to query from DB. But any kind of non-indexed Count, Read or Upsert, Delete operation are needed to be performed. Except Count() for other operations we can have deferred execution.
IList	IList is useful when we want to perform any operation like Add, Remove or Get item at specific index position in the collection. For Add, Remove or other operations we need to transform IEnumerable or IQueryable to IList/ICollection. Except Count() for other operations we can have deferred execution.
List	Use of List is preferable if we want immediate execution and faster response. With list we have a definite implementation of the Interface IList hence all the methods have proper implementation. Fast Response - As for the interfaces most of the operations will have deferred execution as such they have to track and operate on the final collection. Moreover as the collection is not final in the case of deferred execution so for each iteration they might have to allocate memory. Faster response holds good if we need to iterate through the List. Note - With IEnumerable the memory consumption is more and performance is slow. Memory consumption is more as for each operation we need GetEnumerator() run. List is best as it is materialized. Both in terms of speed and memory. Best if data is in memory. So if we have data in memory and we need to iterate, the best approach is to convert to list and iterate.

Summery 2
All interfaces are inherited from IEnumerable. This helps us to use foreach statements.

IEnumerable - Has GetEnumerator() which reads values but not write.

ICollection - Basic interface of all. It supports Count().

IList - Supports addition and removal of items, retrieving items by index etc. It's the most commonly used interface.

IQueryable - IEnumerable interface that supports LINQ. We can always create IQueryable to IList and use LINQ to Objects. But IQueryable is also used for deferred execution of SQL statements in LINQ to SQL and LINQ to Entities.

Note - var list = List.Take<Emp>(10); //Take first 10 values of the list.

IEnumerable describes behavior, while list is an implementation. When we use IEnumerable, we give the compiler a chance to defer the work until later. But using ToList() we use the compiler to get the result immediately.

Example:

//CODE 1

```
static void Main()
{
	var names = new List<string>
		    {"sam", "alexia", "simon", "sumanth", "tony", "sam", "amr",
		    "mark", "drew"};

	var moreThanFiveLetters = names.Where(w => w.Length > 5);
	names[0] = "benjamin";

	foreach (var name in moreThanFiveLetters)
	{
		Console.WriteLine(name);
	}
}
```

This will by default **IEnumerator** so there will be deferred execution (SQL formed but executed when used i.e when it is iterated) so when foreach runs it will run on the state of the last updated names list.
So output will be :
benjamin
alaxia
sumanth

//CODE 2

```
static void Main()
{
	var names = new List<string>
		    {"sam", "alexia", "simon", "sumanth", "tony", "sam", "amr",
		    "mark", "drew"};
	var moreThanFiveLetters = names.Where(w => w.Length > 5).ToList();
	names[0] = "benjamin";

	foreach (var name in moreThanFiveLetters)
```

```
        {
                Console.WriteLine(name);
        }
}
```

This will be List as we have used ToList(). So when we have used Where the output list will have the final data. As we do not have deferred execution here.
So the output will be
alaxia
sumanth

2. Deferred Execution vs Immediate Execution

LINQ provides a common query syntax to query any data source. In a LINQ query, you always work with objects. The object might be in-memory of the program or remote object i.e. out of memory of the program. Based on the object, LINQ query expression is translated and executed.
There are two ways of LINQ query execution as given below:

Deferred Execution
In case of deferred execution, a query is not executed at the point of its declaration. It is executed when the Query variable is iterated by using loop for, foreach etc.
A LINQ query expression often causes deferred execution. Deferred execution provides the facility of query reusability since it always fetches the updated data from the data source which exists at the time of each execution.

Immediate Execution
In case of immediate execution, a query is executed at the point of its declaration. The query which returns a singleton value (single value or a set of values) like **Average, Sum, Count, List etc. caused Immediate Execution.**
You can force a query to execute immediately by calling ToList, ToArray methods.
Immediate execution doesn't provide the facility of query re-usability, since it always contains the same data which is fetched at the time of query declaration.

3. Eager loading vs Lazy loading

Eager Loading: Eager Loading helps you to load all your needed entities at once. i.e. related objects (child objects) are loaded automatically with its parent object.
When to use:

1. Use Eager Loading when the relations are not too much. Thus, Eager Loading is a good practice to reduce further queries on the Server.
2. Use Eager Loading when you are sure that you will be using related entities with the main entity everywhere.

Lazy Loading: In case of lazy loading, related objects (child objects) are not loaded automatically with its parent object until they are requested. **By default LINQ supports lazy loading.**

When to use:

1. Use Lazy Loading when you are using one-to-many collections.
2. Use Lazy Loading when you are sure that you are not using related entities instantly.

3a. Eager Loading, Lazy Loading And Explicit Loading In Entity Framework

Eager Loading

Eager Loading helps you to load all your needed entities at once; i.e., all your child entities will be loaded at a single database call. This can be achieved, using the Include method, which returns the related entities as a part of the query and a large amount of data is loaded at once.
For example, you have a User table and a UserDetails table (related entity to User table), then you will write the code given below. Here, we are loading the user with the Id equal to userId along with the user details.

```
User usr = dbContext.Users.Include(a => a.UserDetails).FirstOrDefault(a =>
a.UserId == userId);
```

If you have multiple levels of child entities, then you can load, using the query given below.

```
User usr = dbContext.Users.Include(a => a.UserDetails.Select(ud =>
ud.Address)).FirstOrDefault(a => a.UserId == userId);
```

Lazy Loading

It is the default behavior of an Entity Framework, where a child entity is loaded only when it is accessed for the first time. It simply delays the loading of the related data, until you ask for it.
For example, when we run the query given below, UserDetails table will not be loaded along with the User table.

```
User usr = dbContext.Users.FirstOrDefault(a => a.UserId == userId);
```

It will only be loaded when you explicitly call for it, as shown below.

```
UserDeatils ud = usr.UserDetails; // UserDetails are loaded here
```

Lazy loading can cause unneeded extra database roundtrips to occur (the so-called **N+1 problem**), and care should be taken to avoid this. See the performance section for more details. With lazy loading, a User is only (lazily) loaded when its UserDetails property is accessed; as a result, each iteration in the inner foreach triggers an additional database query, in its own roundtrip. As a result, after the initial query loading all the data, we then have another query *call*, loading all its posts; this is sometimes called the *N+1* problem, and it can cause very significant performance issues.

Explicit Loading

There are options to disable Lazy Loading in an Entity Framework. After turning Lazy Loading off, you can still load the entities by explicitly calling the Load method for the related entities. There are two ways to use the Load method: Reference (to load a single navigation property) and Collection (to load collections), as shown below.

```
User usr = dbContext.Users.FirstOrDefault(a => a.UserId == userId);


dbContext.Entry(usr).Reference(usr => usr.UserDetails).Load();
```

Ref: https://www.youtube.com/watch?v=gjBn5f23Y1k

When to use what

- Use Eager Loading when the relations are not too much. Thus, Eager Loading is a good practice to reduce further queries on the Server.
- Use Eager Loading when you are sure that you will be using related entities with the main entity everywhere.
- Use Lazy Loading when you are using one-to-many collections.
- Use Lazy Loading when you are sure that you are not using related entities instantly.
- When you have turned off Lazy Loading, use Explicit loading when you are not sure whether or not you will be using an entity beforehand.

How to enable eager loading and lazy loading?

Eager loading

You can use the Include method to specify related data to be included in query results. In the following example, the blogs that are returned in the results will have their Posts property populated with the related posts.

Lazy Loading

Ref: https://learn.microsoft.com/en-us/ef/core/querying/related-data/lazy
Ref: https://www.learnentityframeworkcore.com/lazy-loading
The simplest way to use lazy-loading is by installing the **Microsoft.EntityFrameworkCore.Proxies** package and enabling it with a call to **UseLazyLoadingProxies**. For example:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

Or when using AddDbContext:

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

EF Core will then enable lazy loading for any navigation property that can be overridden--that is, it must be virtual and on a class that can be inherited from. For example, in the following entities, the **Post.Blog** and **Blog.Posts** navigation properties will be lazy-loaded.

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public virtual Blog Blog { get; set; }
}
```

Lazy loading without proxies

Lazy-loading without proxies work by injecting the ILazyLoader service into an entity, as described in Entity Type Constructors. For example:

```
public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
```

```
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}
```

NOTE - Both LINQ and LAMBDA expression supports Lazy Loading by default

For more Lazy Loading concepts :
https://docs.google.com/document/d/1qnS3MjkUv6wRIyjDB0jL9HGaEny3nKlUXszfAJl03Z4/edit

3. IEnumerator vs IEnumerable

IEnumerable and IEnumerator both are interfaces in C#. IEnumerable is an interface defining a single method GetEnumerator() that returns an IEnumerator interface. This works for read only access to a collection that implements that IEnumerable can be used with a foreach statement. IEnumerator has two methods: MoveNext and Reset. It also has a property called Current.