FullStack.Cafe - Kill Your Tech Interview

Q1: Explain what is LINQ? Why is it required? ☆

Topics: LINQ

Answer:

Language Integrated Query or **LINQ** is the collection of standard query operators which provides query facilities into.NET framework language like C#, VB.NET. LINQ is required as it bridges the gap between the world of data and the world of objects.

Q2: What are the *types* of LINQ? ☆

Topics: LINQ

Answer:

- LINQ to Objects
- LINQ to XML
- LINQ to Dataset
- LINQ to SQL
- LINQ to Entities

Q3: What is LINQ in C#? ☆☆

Topics: C# LINQ

Answer:

LINQ stands for Language Integrated Query. LINQ has a great power of querying on any source of data. The data source could be collections of objects, database or XML files. We can easily retrieve data from any object that implements the `IEnumerable<T>` interface.

Q4: Explain how LINQ is useful than Stored Procedures? ☆☆

Topics: LINQ

Answer:

- **Debugging:** It is difficult to debug a stored procedure but as LINQ is part of.NET, visual studio debugger can be used to debug the queries
- **Deployment:** For stored procedure, additional script should be provided but with LINQ everything gets compiled into single DLL hence deployment becomes easy
- **Type Safety:** LINQ is type safe, so queries errors are type checked at compile time

Q5: List out the three main components of LINQ? ☆☆

Topics: LINQ

Answer:

Three main components of LINQ are

- Standard Query Operators
- Language Extensions
- LINQ Providers

Q6: Explain why `SELECT` clause comes after `FROM` clause in LINQ? ☆☆

Topics: LINQ

Answer:

With other programming language and C#, LINQ is used, it requires all the variables to be declared first. “FROM” clause of LINQ query defines the range or conditions to select records. So, FROM clause must appear before SELECT in LINQ.

Q7: In LINQ how will you find the index of the element using `where()` with Lambda Expressions? ☆☆

Topics: LINQ

Answer:

In order to find the index of the element use the overloaded version of `where()` with the lambda expression:

```
where(( i, ix ) => i == ix);
```

Q8: Mention what is the role of `DataContext` classes in LINQ? ☆☆

Topics: LINQ

Answer:

DataContext class acts as a bridge between SQL Server database and the LINQ to SQL. For accessing the database and also for changing the data in the database, it contains connections string and the functions. Essentially a DataContext class performs the following three tasks:

- Create connection to database.
- It submits and retrieves object to database.
- Converts objects to SQL queries and vice versa.

Q9: Explain what is the purpose of LINQ providers in LINQ? ☆☆

Topics: LINQ

Answer:

LINQ providers are set of classes that take an LINQ query which generates method that executes an equivalent query against a particular data source.

Q10: Explain what is *LINQ to Objects*? ☆☆

Topics: LINQ

Answer:

When LINQ queries any `IEnumerable(<T>)` collection or IEnumerable directly without the use of an intermediate LINQ provider or API such as LINQ to SQL or LINQ to XML is referred as **LINQ to Objects**.

Q11: What are *Extension Methods*? ☆☆

Topics: LINQ

Answer:

Extension methods are static functions of a static class. These methods can be invoked just like instance method syntax. These methods are useful when we can not want to modify the class. Consider:

```
public static class StringMethods
{
    public static bool IsStartWithLetterM(this string s)
    {
        return s.StartsWith("m");
    }
}
class Program
{
    static void Main(string[] args)
    {
        string value = "malslfds";
        Console.WriteLine(value.IsStartWithLetterM()); //print true;

        Console.ReadLine();
    }
}
```

Q12: What are *Anonymous Types*? ☆☆

Topics: LINQ

Answer:

Anonymous types are types that are generated by compiler at run time. When we create a anonymous type we do not specify a name. We just write properties names and their values. Compiler at runtime create these properties and assign values to them.

```
var k = new { FirstProperty = "value1", SecondProperty = "value2" };
Console.WriteLine(k.FirstProperty);
```

Anonymous class is useful in LINQ queries to save our intermediate results.

There are some restrictions on Anonymous types as well:

- Anonymous types can not implement interfaces.
- Anonymous types can not specify any methods.
- We can not define static members.
- All defined properties must be initialized.
- We can only define public fields.

Q13: What is *Anonymous function*? ☆☆

Topics: LINQ

Answer:

An Anonymous function is a special function which does not have any name. We just define their parameters and define the code into the curly braces.

Consider:

```
delegate int func(int a, int b);
static void Main(string[] args)
{
    func f1 = delegate(int a, int b)
    {
        return a + b;
    };

    Console.WriteLine(f1(1, 2));
}
```

Q14: Define what is `let` clause? ☆☆☆

Topics: LINQ

Answer:

In a query expression, it is sometimes useful to store the result of a sub-expression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply.

Consider:

```
var names = new string[] { "Dog", "Cat", "Giraffe", "Monkey", "Tortoise" };
var result =
    from animalName in names
    let nameLength = animalName.Length
    where nameLength > 3
    orderby nameLength
    select animalName;
```

Q15: Explain what is the difference between `Skip()` and `SkipWhile()` extension method? ☆☆☆

Topics: LINQ

Answer:

- **Skip() :** It will take an integer argument and from the given `IEnumerable` it skips the top `n` numbers
- **SkipWhile ():** It will continue to skip the elements as far as the input condition is `true`. It will return all remaining elements if the condition is `false`.

Q16: Explain how *standard query operators* useful in LINQ? ☆☆☆

Topics: LINQ

Answer:

A set of extension methods forming a query pattern is known as LINQ Standard Query Operators. As building blocks of LINQ query expressions, these operators offer a range of query capabilities like filtering, sorting, projection, aggregation, etc.

LINQ standard query operators can be categorized into the following ones on the basis of their functionality.

- Filtering Operators (Where, OfType)
- Join Operators (Join, GroupJoin)
- Projection Operations (Select, SelectMany)
- Sorting Operators (OrderBy, ThenBy, Reverse, ...)
- Grouping Operators (GroupBy, ToLookup)
- Conversions (Cast, ToArray, ToList, ...)
- Concatenation (Concat)
- Aggregation (Aggregate, Average, Count, Max, ...)
- Quantifier Operations (All, Any, Contains)
- Partition Operations (Skip, SkipWhile, Take, ...)
- Generation Operations (DefaultIfEmpty, Empty, Range, Repeat)
- Set Operations (Distinct, Except, ...)
- Equality (SequenceEqual)
- Element Operators (ElementAt, First, Last, ...)

Q17: When to use `First()` and when to use `FirstOrDefault()` with LINQ? ☆☆☆

Topics: LINQ

Answer:

- Use `First()` when you know or expect the sequence to have at least one element. In other words, when it is an exceptional occurrence that the sequence is empty.
- Use `FirstOrDefault()` when you know that you will need to check whether there was an element or not. In other words, when it is legal for the sequence to be empty. You should not rely on exception handling for the check. (It is bad practice and might hurt performance).

`First()` will throw an exception if there's no row to be returned, while `FirstOrDefault()` will return the default value (NULL for all reference types) instead.

Q18: What is the difference between `First()` and `Take(1)`? ☆☆☆

Topics: LINQ

Problem:

Consider:

```
var result = List.Where(x => x == "foo").First();
var result = List.Where(x => x == "foo").Take(1);
```

Solution:

The difference between `First()` and `Take()` is that `First()` returns the element itself, while `Take()` returns a sequence of elements that contains exactly one element. (If you pass 1 as the parameter).

Q19: Could you compare *Entity Framework* vs *LINQ to SQL* vs *ADO.NET* with *stored procedures*? ☆☆☆

Topics: LINQ

Answer:

Stored procedures:

(++)

- Great flexibility
- Full control over SQL
- The highest performance available

(--)

- Requires knowledge of SQL
- Stored procedures are out of source control (harder to test)
- Substantial amount of "repeating yourself" while specifying the same table and field names. The high chance of breaking the application after renaming a DB entity and missing some references to it somewhere.
- Slow development

ORM (EF, L2SQL, ADO.NET):

(++)

- Rapid development
- Data access code now under source control
- You're isolated from changes in DB. If that happens you only need to update your model/mappings in one place.

(--)

- Performance may be worse
- No or little control over SQL the ORM produces (could be inefficient or worse buggy). Might need to intervene and replace it with custom stored procedures. That will render your code messy (some LINQ in code, some SQL in code and/or in the DB out of source control).
- As any abstraction can produce "high-level" developers having no idea how it works under the hood

Q20: Could you explian what is the exact deference between *deferred execution* and *Lazy evaluation* in C#? ☆☆☆

Topics: LINQ

Answer:

In practice, they mean essentially the same thing. However, it's preferable to use the term deferred.

- Lazy means "don't do the work until you absolutely have to."
- Deferred means "don't compute the result until the caller actually uses it."

When the caller decides to use the result of an evaluation (i.e. start iterating through an `IEnumerable<T>`), that is precisely the point at which the "work" needs to be done (such as issuing a query to the database).

The term *deferred* is more specific/descriptive as to what's actually going on. When I say that I am lazy, it means that I avoid doing unnecessary work; it's ambiguous as to what that really implies. However, when I say that execution/evaluation is deferred, it essentially means that I am not giving you the real result at all, but rather a ticket you can use to claim the result. I defer actually going out and getting that result until you claim it.

FullStack.Cafe - Kill Your Tech Interview

Q1: Filter out the first 3 even numbers from the list using LINQ ☆☆

Topics: C# LINQ

Answer:

```
var evenNumbers = List
   .Where(x => x % 2 ==0)
   .Take(3)
```

Q2: Explain what is *Lambda Expressions* in LINQ? ☆☆☆

Topics: LINQ

Answer:

Lambda expression is referred as a unique function use to form delegates or expression tree types, where right side is the output and left side is the input to the method. For writing LINQ queries particularly, Lambda expression is used.

Q3: Get the indexes of top `n` items where item `value = true` ☆☆☆

Topics: LINQ

Problem:

I have a `List<bool>`. I need to get the indexes of top n items where item `value = true`.

```
10011001000

TopTrueIndexes(3) = The first 3 indexes where bits are true are 0, 3, 4
TopTrueIndexes(4) = The first 4 indexes where bits are true are 0, 3, 4, 7
```

Solution:

Assuming you have some easily-identifiable condition, you can do something like this, which will work for any `IEnumerable<T>`:

```
var query = source.Select((value, index) => new { value, index })
                  .Where(x => x.value => Condition(value))
                  .Select(x => x.index)
                  .Take(n);
```

The important bits are that you use the overload of `Select` to get index/value pairs before the Where, and then another Select to get just the indexes after the Where... and use Take to only get the first n results.

Q4: Explain what are LINQ *compiled queries*? ☆☆☆

Topics: LINQ

Answer:

There may be scenario where we need to execute a particular query many times and repeatedly. LINQ allows us to make this task very easy by enabling us to create a query and make it compiled always. Benefits of Compiled Queries:

- Query does need to compiled each time so execution of the query is fast.
- Query is compiled once and can be used any number of times.
- Query does need to be recompiled even if the parameter of the query is being changed.

Consider:

```
static class MyCompliedQueries {
    public static Func <DataClasses1DataContext, IQueryable <Person>> CompliedQueryForPerson =
        CompiledQuery.Compile((DataClasses1DataContext context) = >from c in context.Persons select c);
}
```

Q5: Using LINQ to remove elements from a `List<T>` ☆☆☆

Topics: LINQ

Problem:

Given that `authorsList` is of type `List<Author>`, how can I delete the `Author` elements that are equalling to `Bob`?

Solution:

Consider:

```
authorsList.RemoveAll(x => x.FirstName == "Bob");
```

Q6: When trying to decide between using the *Entity Framework* and *LINQ to SQL* as an ORM, what's the difference? ☆☆☆

Topics: LINQ

Answer:

- **LINQ to SQL** only supports 1 to 1 mapping of database tables, views, sprocs and functions available in Microsoft SQL Server. It's a great API to use for quick data access construction to relatively well designed SQL Server databases. LINQ2SQL was first released with C# 3.0 and .Net Framework 3.5.

- **LINQ to Entities** (ADO.Net Entity Framework) is an ORM (Object Relational Mapper) API which allows for a broad definition of object domain models and their relationships to many different ADO.Net data providers. As such, you can mix and match a number of different database vendors, application servers or protocols to design an aggregated mash-up of objects which are constructed from a variety of tables, sources, services, etc. ADO.Net Framework was released with the .Net Framework 3.5 SP1.

Q7: What is *Expression Trees* and how they used in LINQ? ☆☆☆

Topics: LINQ

Answer:

An **Expression Tree** is a data structure that contains Expressions, which is basically code. So it is a tree structure that represents a calculation you may make in code. These pieces of code can then be executed by "running" the expression tree over a set of data.

In LINQ, expression trees are used to represent structured queries that target sources of data that implement `IQueryable<T>`. For example, the LINQ provider implements the `IQueryable<T>` interface for querying relational data stores. The C# compiler compiles queries that target such data sources into code that builds an expression tree at runtime. The query provider can then traverse the expression tree data structure and translate it into a query language appropriate for the data source.

Q8: Explain the difference between `Select` and `Where` ☆☆☆☆

Topics: C# LINQ

Problem:

Consider:

```
ContextSet().Select(x=> x.FirstName == "John") // A
ContextSet().Where(x=> x.FirstName == "John") // B
```

When should I use `.Select` vs `.Where`?

Solution:

- **Select** is a projection, so what you get is the expression `x=> x.FirstName == "John"` evaluated for each element in ContextSet() on the server. i.e. lots of true/false values (the same number as your original list). If you look the select will return something like `IEnumerable<bool>` (because the type of `x=> x.FirstName == "John"` is a bool). Use `Select` when you want to keep all results, but change their type (project them).
- **Where** filters the results, returning an enumerable of the original type (no projection). Use `Where` when you want to filter your results, keeping the original type

Q9: What is an equivalent to the `let` keyword in chained LINQ extension method calls? ☆☆☆☆

Topics: LINQ

Answer:

Essentially `let` creates an anonymous tuple. It's equivalent to:

```
var result = names.Select(animal => new { animal = animal, nameLength = animal.Length })
    .Where(x => x.nameLength > 3)
    .OrderBy(y => y.nameLength)
    .Select(z => z.animal);
```

Q10: When should I use a `CompiledQuery`? ☆☆☆☆

Topics: LINQ

Answer:

You should use a `CompiledQuery` when all of the following are true:

- The query will be executed more than once, varying only by parameter values.
- The query is complex enough that the cost of expression evaluation and view generation is "significant" (trial and error)
- You are not using a LINQ feature like `IEnumerable<T>.Contains()` which won't work with `CompiledQuery`.
- You have already simplified the query, which gives a bigger performance benefit, when possible.
- You do not intend to further compose the query results (e.g., restrict or project), which has the effect of "decompiling" it.

`CompiledQuery` does its work the first time a query is executed. It gives no benefit for the first execution. Like any performance tuning, generally avoid it until you're sure you're fixing an actual performance hotspot. Note EF 5 will do this automatically.

Q11: Name some advantages of LINQ over Stored Procedures ☆☆☆☆

Topics: LINQ

Answer:

Some advantages of LINQ over sprocs:

1. **Type safety**: I think we all understand this.
2. **Abstraction**: This is especially true with LINQ-to-Entities. This abstraction also allows the framework to add additional improvements that you can easily take advantage of. PLINQ is an example of adding multi-threading support to LINQ. Code changes are minimal to add this support. It would be MUCH harder to do this data access code that simply calls sprocs.
3. **Debugging support**: I can use any .NET debugger to debug the queries. With sprocs, you cannot easily debug the SQL and that experience is largely tied to your database vendor (MS SQL Server provides a query analyzer, but often that isn't enough).
4. **Vendor agnostic**: LINQ works with lots of databases and the number of supported databases will only increase. Sprocs are not always portable between databases, either because of varying syntax or feature support (if the database supports sprocs at all).
5. **Deployment**: It's easier to deploy a single assembly than to deploy a set of sprocs. This also ties in with #4.
6. **Easier**: You don't have to learn T-SQL to do data access, nor do you have to learn the data access API (e.g. ADO.NET) necessary for calling the sprocs. This is related to #3 and #4.

Q12: What are the benefits of a *Deferred Execution* in LINQ? ☆☆☆☆☆

Topics: C# LINQ

Answer:

In LINQ, queries have two different behaviors of execution: **immediate** and **deferred**.

Deferred execution means that the evaluation of an expression is delayed until its realized value is actually required. It greatly improves performance by avoiding unnecessary execution.

Consider:

```
var results = collection.Select(item => item.Foo).Where(foo => foo < 3).ToList();
```

With deferred execution, the above iterates your collection one time, and each time an item is requested during the iteration, performs the map operation, filters, then uses the results to build the list.

If you were to make LINQ fully execute each time, each operation (Select / Where) would have to iterate through the entire sequence. This would make chained operations very inefficient.

Q13: Why use `AsEnumerable()` rather than casting to `IEnumerable<T>`? ☆☆☆☆☆

Topics: LINQ

Answer:

Readability is the main issue here. Consider that

```
Table.AsEnumerable().Where(somePredicate)
```

is far more readable than

```
((IEnumerable<TableObject>)Table).Where(somePredicate).
```

Q14: What is the difference between returning `IQueryable<T>` vs. `IEnumerable<T>`? ☆☆☆☆☆

Topics: LINQ

Problem:

What is the difference between returning `IQueryable<T>` vs. `IEnumerable<T>`?

```
IQueryable < Customer > custs = from c in db.Customers
where c.City == "<City>"
select c;

IEnumerable < Customer > custs = from c in db.Customers
where c.City == "<City>"
select c;
```

Will both be deferred execution and when should one be preferred over the other?

Solution:

The difference is that `IQueryable<T>` is the interface that allows LINQ-to-SQL (LINQ.-to-anything really) to work. So if you further refine your query on an `IQueryable<T>`, that query will be executed in the database, if possible.

Consider:

```
IQueryable<Customer> custs = ...;
// Later on...
var goldCustomers = custs.Where(c => c.IsGold);
```

That code will execute SQL to only select gold customers. On the other hand, will execute the original query in the database, then filtering out the non-gold customers in the memory:

```
IEnumerable<Customer> custs = ...;
// Later on...
var goldCustomers = custs.Where(c => c.IsGold);
```

This is quite an important difference, and working on `IQueryable<T>` can in many cases save you from returning too many rows from the database. Another prime example is doing paging: If you use `Take` and `Skip` on `IQueryable`, you will only get the number of rows requested; doing that on an `IEnumerable<T>` will cause all of your rows to be loaded in memory.

Q15: What is the difference between `Select` and `SelectMany`? ☆☆☆☆☆

Topics: LINQ

Answer:

- **Select** is a simple one-to-one projection from source element to a result element.
- **SelectMany** is used when there are multiple from clauses in a query expression: each element in the original sequence is used to generate a new sequence. It also *flattens queries that return lists of lists*.

Consider:

```
public class PhoneNumber
{
    public string Number { get; set; }
}

public class Person
{
    public IEnumerable<PhoneNumber> PhoneNumbers { get; set; }
    public string Name { get; set; }
}

IEnumerable<Person> people = new List<Person>();

// Select gets a list of lists of phone numbers
IEnumerable<IEnumerable<PhoneNumber>> phoneLists = people.Select(p => p.PhoneNumbers);

// SelectMany flattens it to just a list of phone numbers.
IEnumerable<PhoneNumber> phoneNumbers = people.SelectMany(p => p.PhoneNumbers);

// And to include data from the parent in the result:
// pass an expression to the second parameter (resultSelector) in the overload:
var directory = people
   .SelectMany(p => p.PhoneNumbers,
               (parent, child) => new { parent.Name, child.Number });
```

Q16: Name some disadvantages of LINQ over Stored Procedures ☆☆☆☆☆

Topics: LINQ

Answer:

Some disadvantages of LINQ vs sprocs:

1. **Network traffic**: sprocs need only serialize sproc-name and argument data over the wire while LINQ sends the entire query. This can get really bad if the queries are very complex. However, LINQ's abstraction allows Microsoft to improve this over time.
2. **Less flexible**: Sprocs can take full advantage of a database's featureset. LINQ tends to be more generic in it's support. This is common in any kind of language abstraction (e.g. C# vs assembler).
3. **Recompiling**: If you need to make changes to the way you do data access, you need to recompile, version, and redeploy your assembly. Sprocs can *sometimes* allow a DBA to tune the data access routine without a need to redeploy anything.

Security and manageability are something that people argue about too.

1. **Security**: For example, you can protect your sensitive data by restricting access to the tables directly, and put ACLs on the sprocs. With LINQ, however, you can still restrict direct access to tables and instead put ACLs on updatable table *views* to achieve a similar end (assuming your database supports updatable views).
2. **Manageability**: Using views also gives you the advantage of shielding your application non-breaking from schema changes (like table normalization). You can update the view without requiring your data access code to change.

Q17: Can you provide a concise distinction between *anonymous method* and *lambda expressions*? ☆☆☆☆☆

Topics: LINQ

Answer:

Consider the following queries:

```
Customers.Where(delegate(Customers c) { return c.City == "London";}); // delegate
Customers.Where(c => c.City == "London"); // lambda
```

The first generates:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[ContactTitle], [t0].[Address],
[t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
    FROM [Customers] AS [t0]
```

Whereas the second generates:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[ContactTitle], [t0].[Address],
[t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
    FROM [Customers] AS [t0]
    WHERE [t0].[City] = @p0
```

The compiler is able to determine that the lambda expression is a simple single-line expression which can be retained as an expression tree, whereas the anonymous delegate is not a lambda expression and thus not wrappable as an `Expression<Func<T>>`. As a result in the first case, the best match for the Where extension method is the one that extends IEnumerable rather than the IQueryable version which requires an `Expression<Func<T, bool>>`.