

1.1.1 LINQ interview questions part

Question: Given an array of numbers, find if ALL numbers are a multiple of a provided number. For example, all of the following numbers - 30, 27, 15, 90, 99, 42, 75 are multiples of 3.

The trick here is to use the **Enumerable.All<TSource>** method which determines whether all elements of a sequence satisfy a condition.

```
static void Main(string[] args)
{
    int[] numbers = { 30, 27, 15, 90, 99, 42, 75 };

    bool isMultiple = MultipleTester(numbers, 3);
}

private static bool MultipleTester(int[] numbers, int divisor)
{
    bool isMultiple =
        numbers.All(number => number % divisor == 0);

    return isMultiple;
}
```

Question: Given an array of numbers, find if ANY of the number is divisible by 5. For example, one of the following numbers - 30, 27, 18, 92, 99, 42, 72 is divisible by 5.

Again, the key fact here is to use **Enumerable.Any<TSource>** method which determines whether any element of a sequence satisfies a condition. The code is very similar to the ALL case.

```
static void Main(string[] args)
{
    int[] numbers = { 30, 27, 18, 92, 99, 42, 72 };

    bool isDivisible = IsDivisible(numbers, 3);
}
```

```
}

private static bool IsDivisible(int[] numbers, int divisor)
{
    bool isDivisible =
        numbers.Any(number => number % divisor == 0);

    return isDivisible;
}
```

Another way to present a similar question that utilizes **Enumerable.Any<TSource>** method is shown below.

Question: Given a GPSManufacturer and GPSDevice data structure as defined below, find all the manufacturers that have at least 1 or more active GPS devices.

```
class GpsDevice
{
    public string Name;
    public bool IsActive;
}

class GpsManufacturer
{
    public string Name;
    public GpsDevice[] Devices;
}

static List<GpsManufacturer> gpsManufacturers =
    new List<GpsManufacturer>
    {
        new GpsManufacturer
        {
            Name = "manf1",
            Devices = new GpsDevice[]
            {
                new GpsDevice {Name = "device1", IsActive = true},
                new GpsDevice {Name = "device2", IsActive = false}
            }
        },
        new GpsManufacturer
        {
            Name = "manf2",
            Devices = new GpsDevice[]
            {
                new GpsDevice {Name = "device1", IsActive = false},
                new GpsDevice {Name = "device2", IsActive = false}
            }
        },
        new GpsManufacturer
    }
```

```
{Name = "manf3"}
};
```

The following function finds out the active GPS manufacturers.

```
public static void ActiveGpsManufacturers()
{
    // Determine which manufacturers have a active device.
    IEnumerable<string> activeManf =
        from manf in gpsManufacturers // for all manufacturers
        where manf.Devices != null && // they have a list of devices
            // and at least one of the device is active
            manf.Devices.Any(m => m.IsActive == true)
        select manf.Name; // select them

    // just for debugging
    foreach (string name in activeManf)
        Console.WriteLine(name);
}
```

As an extension question and using Lambda expressions, consider the following question:

Question: Find the count of all the manufacturers that have at least 1 or more active GPS devices.

```
public static void ActiveGpsManufacturersCount()
{
    // to count the number of active manufacturers
    int activeManfCount =
        gpsManufacturers.Count
            (manf => (manf.Devices != null &&
                manf.Devices.Any(m => m.IsActive == true)));

    // debugging
    Console.WriteLine("Active Manf count: {0}", activeManfCount);
}
```

MSDN has extensive documentation on LINQ. I recommend reading about the various functions and then trying out various queries on your own. LINQPad is an excellent tool to learn also.

Based on popular demand, I am posting a follow up post to my first [LINQ interview questions](#) article. In this post, we will follow a similar pattern of rapid fire interview questions and answers on LINQ concepts.

Question 1: How can you sort a result set based on 2 columns?

Assume that you have 2 tables – Product and Category and you want to sort first by category and then by product name.

```
var sortedProds = _db.Products.OrderBy(c => c.Category).ThenBy(n => n.Name)
```

Question 2: How can you use LINQ to query against a DataTable?

You cannot query against the `DataTable`'s `Rows` collection, since `DataRowCollection` doesn't implement `IEnumerable<T>`. You need to use the `AsEnumerable()` extension for `DataTable`. As an example:

```
var results = from myRow in myDataTable.AsEnumerable()  
where myRow.Field<int>("RowNo") == 1  
select myRow;
```

Question 3: LINQ equivalent of foreach for IEnumerable<T>

There is no `ForEach` extension for `IEnumerable`; only for `List`. There is a very good reason for this as explained by Eric Lippert [here](#).

Having said that, there are two ways you can solve this:

Convert the items to list and then do a foreach on it:

```
items.ToList().ForEach(i => i.DoStuff());
```

Or, alternatively, you can write an extension method of your own.

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Question 4: When to use .First and when to use .FirstOrDefault with LINQ?

You should use First when you know or expect the sequence to have at least one element. In other words, when it is an exceptional if 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.

Question 5: Write a LINQ expression to concatenate a List<string> in a single string separated by a delimiter.

First of all, it is better to use string.Join to tackle this problem. But for interview purposes, this problem can be approached as follows:

```
string delimiter = ",";
List<string> items = new List<string>() { "foo", "boo", "john", "doe" };
Console.WriteLine(items.Aggregate((i, j) => i + delimiter + j));
```

Question 6: Write a LINQ expression to concatenate a List<MyClass> objects in a single string separated by a delimiter. The class provides a specific property (say Name) that contains the string in question.

```
items.Select(i => i.Name).Aggregate((i, j) => i + delimiter + j)
```

In this post, we will do a quick review of LINQ operators Any() and All().

Question 1: Using LINQ, determine if any word in a list contains the substring “ei”.

```
string[] words = { "believe", "relief", "receipt", "field" };
bool iAfterE = words.Any(w => w.Contains("ei"));
Console.WriteLine("list has words that contains 'ei': {0}", iAfterE);
```

Question 2: Using LINQ, return a grouped list of products only for categories that have at least one product that is out of stock.

In this case, we run any over a grouping as shown below:

```
List<Product> products = GetProductList();
var productGroups =
    from p in products
    group p by p.Category into g
    where g.Any(p => p.UnitsInStock == 0)
    select new { Category = g.Key, Products = g };
```

Question 3: Determine if an array of numbers contains all odds.

```
int[] numbers = { 1, 11, 3, 19, 41, 65, 19 };
bool onlyOdd = numbers.All(n => n % 2 == 1);
Console.WriteLine("The list contains only odd numbers: {0}", onlyOdd);
```

Question 4: Using LINQ, return a grouped list of products only for categories that have all of their products in stock.

```
List<Product> products = GetProductList();

var productGroups =
    from p in products
    group p by p.Category into g
    where g.All(p => p.UnitsInStock > 0)
    select new { Category = g.Key, Products = g };
```

he above LINQ expression:

```
var booktitles =
from b in books
select b.Title;
```

Is equivalent to the following SQL query:

```
SELECT Title from Books
```

2 LINQ Operators:

Apart from the operators used so far, there are several other operators, which implement all query clauses. Let us look at some of the operators and clauses.

3 The Join clause:

The 'join clause' in SQL is used for joining two data tables and displays a data set containing columns from both the tables. LINQ is also capable of that. To check this, add another class named Salesdetails.cs in the previous project:

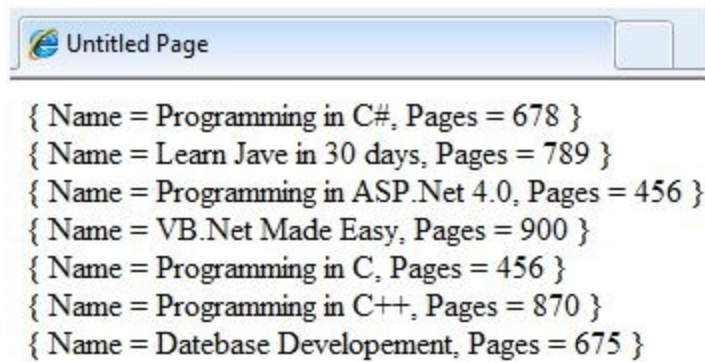
```
public class Salesdetails
{
    public int sales { get; set; }
    public int pages { get; set; }
    public string ID {get; set;}

    public static IEnumerable<Salesdetails> getsalesdetails()
    {
        Salesdetails[] sd =
        {
            new Salesdetails { ID = "001", pages=678, sales = 110000},
            new Salesdetails { ID = "002", pages=789, sales = 60000},
            new Salesdetails { ID = "003", pages=456, sales = 40000},
            new Salesdetails { ID = "004", pages=900, sales = 80000},
            new Salesdetails { ID = "005", pages=456, sales = 90000},
            new Salesdetails { ID = "006", pages=870, sales = 50000},
            new Salesdetails { ID = "007", pages=675, sales = 40000},
        };
        return sd.OfType<Salesdetails>();
    }
}
```

Add the codes in the Page_Load event handler to query on both the tables using the join clause:

```
protected void Page_Load(object sender, EventArgs e)
{
    IEnumerable<Books> books = Books.GetBooks();
    IEnumerable<Salesdetails> sales =
        Salesdetails.getsalesdetails();
    var booktitles = from b in books
        join s in sales
        on b.ID equals s.ID
        select new { Name = b.Title, Pages = s.pages };
    foreach (var title in booktitles)
        lblbooks.Text += String.Format("{0} <br />", title);
}
```

The resulted Page:

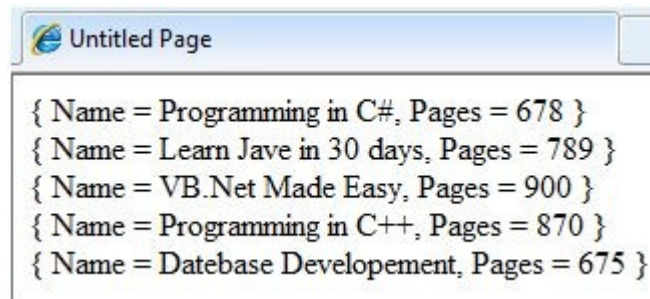


4 The Where clause:

The 'where clause' allows adding some conditional filters to the query. For example, if you want to see the books, where the number of pages are more than 500, change the Page_Load event handler to:

```
var booktitles = from b in books
    join s in sales
    on b.ID equals s.ID
    where s.pages > 500
    select new { Name = b.Title, Pages = s.pages };
```

The query returns only those rows, where the number of pages is more than 500:

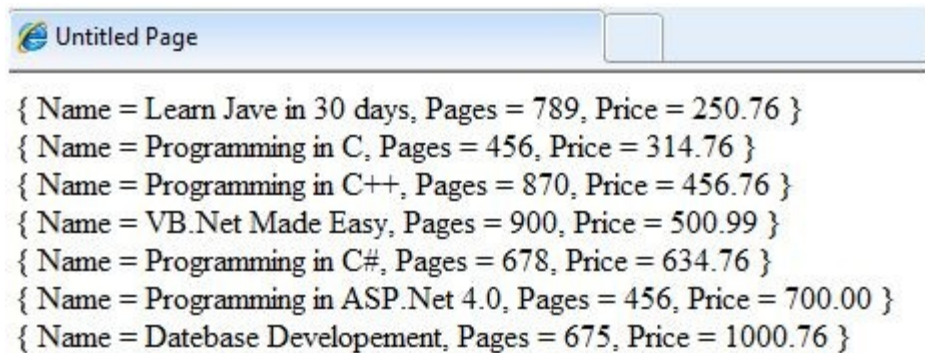


5 The Orderby and Orderbydescending clauses:

These clauses allow sorting the query results. To query the titles, number of pages and price of the book, sorted by the price, write the following code in the Page_Load event handler:

```
var booktitles = from b in books
                 join s in sales
                 on b.ID equals s.ID
                 orderby b.Price
                 select new { Name = b.Title,
                             Pages = s.pages, Price = b.Price};
```

The returned tuples are:



```
{ Name = Learn Java in 30 days, Pages = 789, Price = 250.76 }
{ Name = Programming in C, Pages = 456, Price = 314.76 }
{ Name = Programming in C++, Pages = 870, Price = 456.76 }
{ Name = VB.Net Made Easy, Pages = 900, Price = 500.99 }
{ Name = Programming in C#, Pages = 678, Price = 634.76 }
{ Name = Programming in ASP.Net 4.0, Pages = 456, Price = 700.00 }
{ Name = Database Developement, Pages = 675, Price = 1000.76 }
```

6 The Let clause:

The let clause allows defining a variable and assigning it a value calculated from the data values. For example, to calculate the total sale from the above two sales, you need to calculate:

```
TotalSale = Price of the Book * Sales
```

To achieve this, add the following code snippets in the Page_Load event handler:

The let clause allows defining a variable and assigning it a value calculated from the data values. For example, to calculate the total sale from the above two sales, you need to calculate:

```
var booktitles = from b in books
                 join s in sales
                 on b.ID equals s.ID
                 let totalprofit = (b.Price * s.sales)
                 select new { Name = b.Title, TotalSale = totalprofit};
```

The resultant query page looks like:

```
Untitled Page
{ Name = Programming in C#, TotalSale = 69823600.00 }
{ Name = Learn Jave in 30 days, TotalSale = 15045600.00 }
{ Name = Programming in ASP.Net 4.0, TotalSale = 28000000.00 }
{ Name = VB.Net Made Easy, TotalSale = 40079200.00 }
{ Name = Programming in C, TotalSale = 28328400.00 }
{ Name = Programming in C++, TotalSale = 22838000.00 }
{ Name = Datebase Development, TotalSale = 40030400.00 }
```

Deferred vs Immediate Query Execution in LINQ

Posted by: [Suprotim Agarwal](#) , on 8/12/2011, in Category [LINQ](#)

Views: 102785

<http://www.dotnetcurry.com/linq/750/deferred-vs-immediate-query-execution-linq>

Abstract: In LINQ, queries have two different behaviors of execution: immediate and deferred. In this article, we will take a quick overview of how Deferred query execution and Immediate Query Execution works in LINQ

In LINQ, queries have two different behaviors of execution: immediate and deferred. In this article, we will take a quick overview of how Deferred query execution and Immediate Query Execution works in LINQ

If you do LINQ programming, do check my article [50 LINQ Examples, Tips and How To's](#)

6.1 Deferred Query Execution

To understand Deferred Query Execution, let's take the following example which declares some Employees and then queries all employees with Age > 28:

```
class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

© DotNetCurry.com

```
static void Main(string[] args)
{
    var empList = new List<Employee>(
        new Employee[]
        {
            new Employee {ID = 1, Name = "Jack", Age = 32},
            new Employee {ID = 2, Name = "Rahul", Age = 35},
            new Employee {ID = 3, Name = "Sheela", Age = 28},
            new Employee {ID = 4, Name = "Mary", Age = 25}
        });

    var lst = from e in empList
              where e.Age > 28
              select new { e.Name };

    foreach (var emp in lst)
        Console.WriteLine(emp.Name);

    Console.ReadLine();
}
```

OUTPUT: Jack, Rahul

Looking at the query shown above, it appears that the query is executed at the point where the arrow is pointing towards. However that's not true. The query is actually executed when the query variable is *iterated* over, not when the query variable is created. This is called *deferred execution*.

Now how do we prove that the query was not executed when the query variable was created? It's simple. Just create another Employee instance **after** the query variable is created

```
static void Main(string[] args)
{
    var empList = new List<Employee>(
        new Employee[]
        {
            new Employee {ID = 1, Name = "Jack", Age = 32},
            new Employee {ID = 2, Name = "Rahul", Age = 35},
            new Employee {ID = 3, Name = "Sheela", Age = 28},
            new Employee {ID = 4, Name = "Mary", Age = 25}
        });

    var lst = from e in empList
              where e.Age > 28
              select new { e.Name };

    empList.Add(new Employee {ID = 5, Name = "Bill", Age = 39});

    foreach (var emp in lst)
        Console.WriteLine(emp.Name);
}
```

Notice we are creating a new Employee instance *after* the query variable is created. Now had the query been executed when the query variable is created, the results would be the same as the one we got earlier, i.e. only two employees would meet the criteria of Age > 28. However the output is not the same

OUTPUT: Jack, Rahul, Bill.

What just happened is that the execution of the query was *deferred* until the query variable was iterated over in a foreach loop. This allows you to execute a query as frequently as you want to, like fetching the latest information from a database that is being updated frequently by other applications. You will always get the latest information from the database in this case.

6.2 Immediate Query Execution

You can also force a query to execute immediately, which is useful for caching query results. Let us say we want to display a count of the number of employees that match a criteria.

```
static void Main(string[] args)
{
    var empList = new List<Employee>(
        new Employee[]
        {
            new Employee {ID = 1, Name = "Jack", Age = 32},
            new Employee {ID = 2, Name = "Rahul", Age = 35},
            new Employee {ID = 3, Name = "Sheela", Age = 28},
            new Employee {ID = 4, Name = "Mary", Age = 25}
        });

    var lst = (from e in empList
               where e.Age > 28
               select e).Count();

    empList.Add(new Employee {ID = 5, Name = "Bill", Age = 39});

    Console.WriteLine("Total Employees who age is > 28 are {0} ", lst);

    Console.ReadLine();
}
```

© DotNetCurry.com

← **Immediate Execution**

In the query shown above, in order to count the elements that match the condition, the query must be executed, and this is done automatically when Count() is called. So adding a new employee instance *after* the query variable declaration does not have any effect here, as the query is already executed. The output will be 2, instead of 3.

The basic difference between a Deferred execution vs Immediate execution is that Deferred execution of queries produce a sequence of values, whereas Immediate execution of queries return a singleton value and is executed immediately. Examples are using Count(), Average(), Max() etc.

Note: To force immediate execution of a query that does not produce a singleton value, you can call the ToList(), ToDictionary() or the ToArray() method on a query or query variable. These are called conversion operators which allow you to make a copy/snapshot of the result and access is as many times you want, without the need to re-execute the query.

Hopefully novice developers will now know the basic difference between Deferred and Immediate Execution of queries.