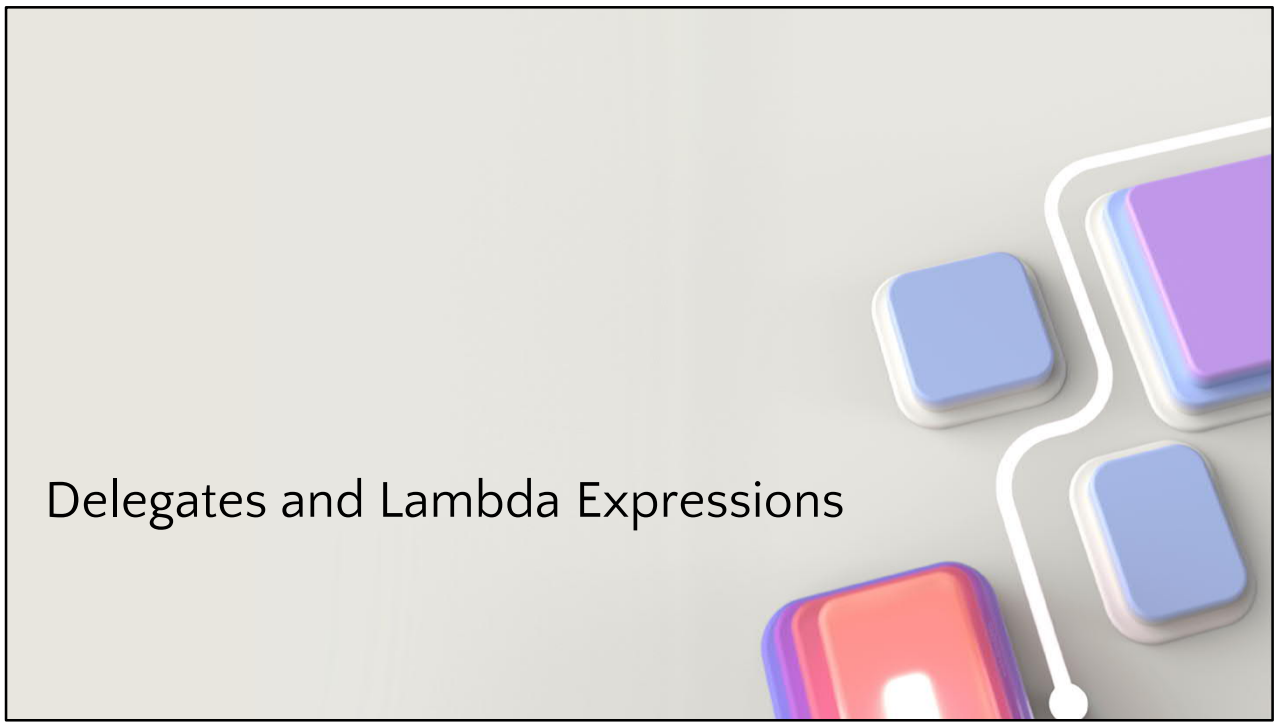
Microsoft

Language Integrated Query

Delegates and Lambda Expressions



Delegates (Function Pointers)

```
class Program
{
      // defining a function pointer that will accept an int parameter
      // and will return an int
      delegate int Transformer(int x);

      static int Square(int x)
      {
            return x * x;
      }
      public static void Main(string[] args)
      {
            Transformer transform = Square;
            Console.WriteLine(transform(10));
      }
}
```

https://www.csharptutorial.net/csharp-tutorial/csharp-anonymous-methods/

Anonymous Method

```
class Program
{
    delegate int Transformer(int x);

    public static void Main(string[] args)
    {
        Transformer transform =
              delegate (int x) { return x * x; };

        Console.WriteLine(transform(10)); // 100
    }
}
```

https://www.csharptutorial.net/csharp-tutorial/csharp-anonymous-methods/

Sometimes, defining a new method is not necessary because you only use it once. It creates unwanted overhead. C# allows you to create a delegate and immediately assign a code block to it. An anonymous method uses the delegate keyword followed by an optional parameter declaration and method body.

The following statement defines an anonymous method and assigns it to the Transformer delegate:

delegate (int x) { return x * x; }

Lambda Expression

- A lambda expression is an anonymous method written in place of a delegate instance. A lambda expression is used to create an anonymous function.
- Use the lambda declaration operator => to separate the lambda's parameter list from its body
- A lambda expression can be of any of the following two forms:
  - Expression lambda that has an expression as its body:
    (input-parameters) => expression
  - Statement lambda that has a statement block as its body:
    (input-parameters) => { <sequence-of-statements> }

https://www.csharptutorial.net/csharp-tutorial/csharp-lambda-expression/

Lambda Expression: Examples

```
var square = (int x) => x * x;  // statement
var result = square(10);

Console.WriteLine(result); // 100
```

```
var square = (int x) => { return x * x; }; // expression
var result = square(10);

Console.WriteLine(result); // 100
```

```
var square = int (int x) => { return x * x; }; // with return type
var result = square(10);

Console.WriteLine(result); // 100
```

Lambda Expression → Delegate Type

- Any lambda expression can be converted to a delegate type.
- The types of its parameters and return value define the delegate type to which a lambda expression can be converted.
  - `Doesn't return a value → Converted to Action delegate type;`
  - `Else Converted to Func delegate type.`

Examples:

- Lambda expression that has two parameters and returns no value → Converted to an Action<T1,T2> delegate. It can up to 0 ➔ 16 parameters.
- A lambda expression that has one parameter and returns a value → converted to a Func<T,TResult> delegate. It can up to 0 ➔ 16 input parameters!

*C# provides **separate delegate types** — **`Func`**, **`Action`**, and **`Predicate`** — to make code **more readable**, **type-safe**, and **intention-revealing***

Delegate Types: Func

- Accepts input params and returns a value;
- Last value in the parameter list is the output param type.
- You can specify 0 → 16 input parameters, besides the output type.

```
class Program
{
    static void Main(string[] args)
    {
        Func<int, int, int> addFunc = new Func<int, int, int>(Add);
        int result = addFunc(3, 4);
        Console.WriteLine(result);
        Console.ReadLine();

    }
    static int Add(int a, int b)
    {
        return a + b;
    }
}
```

https://learn.microsoft.com/en-us/dotnet/api/system.func-2?view=net-9.0

Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter.

Type Parameters

T

The type of the parameter of the method that this delegate encapsulates. This type parameter is contravariant. That is, you can use either the type you specified or any type that is less derived.

TResult

The type of the return value of the method that this delegate encapsulates. This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived.

Func<T,Tresult>: Example

```
// Declare a Func variable and assign a lambda expression to the
// variable. The method takes a string and converts it to uppercase.
Func<string, string> selector = str ⇒ str.ToUpper();

// Create an array of strings.
string[] words = { "orange", "apple", "Article", "elephant" };

// Query the array and select strings according to the selector method.
IEnumerable<String> aWords = words.Select(selector);

// Output the results to the console.
foreach (String word in aWords)
    Console.WriteLine(word);
```

/*
This code example produces the following output:

ORANGE
APPLE
ARTICLE
ELEPHANT

*/

Delegate Type: Action

Represents a method that does not return a value

```
static void Main(string[] args)
{
    Action<string> log = new Action<string>(LogInfo);
    log.Invoke("Hi ALL");
    Console.ReadLine();

}

static void LogInfo(string message)
{
    Console.WriteLine(message);
}
```

Delegate Type: Predicate

- Accepts input params and returns a value;
- Always return bool.

```
class Program
{
    static void Main(string[] args)
    {
        Predicate<int> IsEven = new Predicate<int>(IsEvenNumber);
        Console.WriteLine(IsEven(10));
        Console.WriteLine(IsEven(1567));
        Console.ReadLine();

    }
    static bool IsEvenNumber(int number)
    {
        return number % 2 == 0;
    }
}
```

https://learn.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-9.0

IEnumerable vs. IQueryable

Feature	IEnumerable	IQueryable
Namespace	System.Collections	System.Linq
Purpose	For in-memory collection iteration	For querying remote data sources (e.g., databases)
Execution	Deferred execution, in-memory	Deferred execution, remote (e.g., SQL)

Aspect	IEnumerable	IQueryable
Where executed	On the client (in-memory)	On the server (e.g., SQL database)
Query translation	Not translated to SQL	Translated to SQL (via LINQ providers)
Performance	May be slower for large datasets	More efficient for large datasets

IEnumerable

- Namespace: System.Collections
- Represents a forward-only cursor over a collection.
- Used for in-memory data (like arrays, lists).
- Executes queries in memory (LINQ to Objects).
- Deferred execution applies, but filtering happens after data is loaded into memory.

Usage Scenarios:

- Best for small datasets already in memory.
- Cannot translate queries to SQL or remote sources.
- Every operation runs on the client side.

IEnumerable (Code Sample)

```
List<string> names =
    new List<string> { “Babar", "Ahmed", "Sara", “Lucy" };

// Filtering happens in memory
IEnumerable<string> result =
              names.Where(n ⇒ n.StartsWith("A"));

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

https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-9.0

IQueryable

- Namespace: System.Linq
- Extends IEnumerable and adds the ability to build expression trees.
- Used for remote data sources (like Entity Framework, LINQ to SQL).
- Queries are translated to SQL and executed on the database server.
- Supports deferred execution and query composition.

Key Points

- Best for large datasets or remote data sources.
- Reduces memory usage by filtering at the source.
- Enables lazy loading and server-side execution.

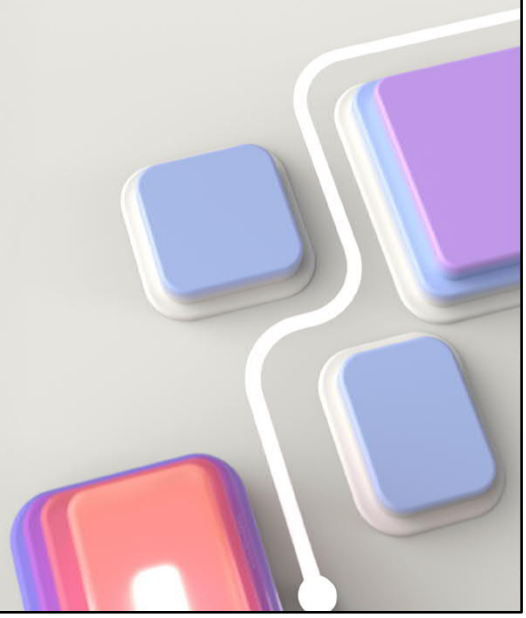
IQueryable (Code Example)

```
using (var context = new MyDbContext())
{
    IQueryable<Customer> query =
         context.Customers.Where(c ⇒ c.City == "Jeddah");

    foreach (var customer in query)
    {
        Console.WriteLine(customer.Name);
    }
}
```

https://learn.microsoft.com/en-us/dotnet/api/system.linq.iqueryable?view=net-9.0

Language Integration Query



Language Integrated Query

- Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language.
- It provides language-level querying capabilities, that enable you to write expressive declarative code.
  - Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support.
  - Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on.
- With LINQ, a query is a first-class language construct, just like classes, methods, and events.

LINQ Queries

Traditional Code

```
var petLookup = new Dictionary<int, Pet>();

foreach (var pet in pets)
{
    petLookup.Add(pet.RFID, pet);
}
```

LINQ

```
var petLookup = pets.ToDictionary(pet => pet.RFID);
```

LINQ Queries

```
// Filtering a list.
var germanShepherds = dogs.Where(dog =>
                    dog.Breed == DogBreed.GermanShepherd);

// Using the query syntax.
var queryGermanShepherds =
               from dog in dogs
               where dog.Breed == DogBreed.GermanShepherd
               select dog;

// Mapping a list from type A to type B.
var cats = dogs.Select(dog => dog.TurnIntoACat());
```

LINQ Queries

```
// Summing the lengths of a set of strings.
int seed = 0; // initial value
int sumOfStrings = strings.Aggregate(seed,
    (partialSum, nextString) =>
                    partialSum + nextString.Length);
```

```
strings.Sum(s => s.Length);
```

Using null-conditional operator (?.) *and the* ***null-coalescing operator (??)***:

```
int totalLength = strings.Sum(s => s?.Length ?? 0);
```

https://learn.microsoft.com/en-us/dotnet/standard/linq/

s?.Length → returns null if s is null, otherwise returns Length.
?? 0 → replaces null with 0.
So, if a string is null, it contributes 0 to the sum.

This way, your code won't throw a NullReferenceException if the collection contains null values.