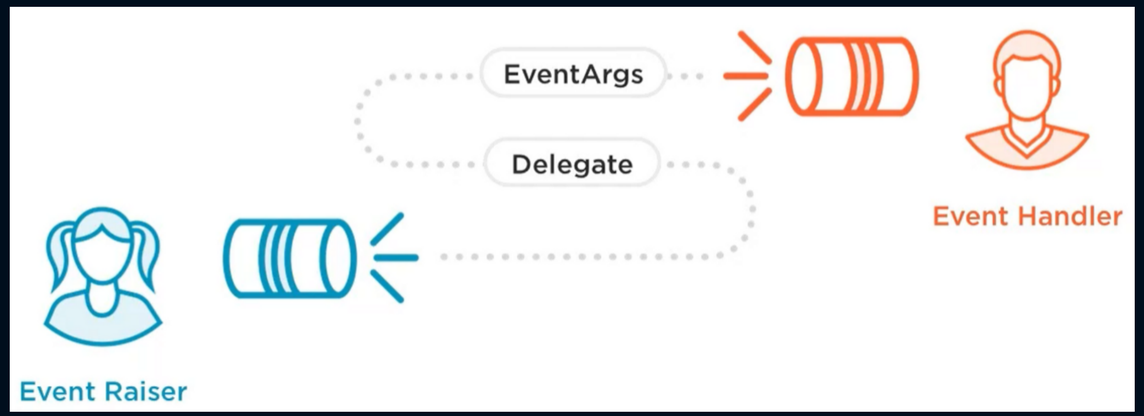
Delegates

A delegate is a specialized class often call a “Function pointer”.
The glue/pipeline between an event and an event handler. Delegates in C# are a type-safe way to encapsulate a method or a group of methods and treat them as objects.



Delegate role

Example

Delegate invocation list

A delegate has a base class known as MultiCast Delegate. A Multicast delegate can reference more than one delegate function, tracks delegate reference using an invocation list and invocations are sequential, this means they can reference multiple methods. When a multicast delegate is invoked, all the referenced methods are called.

Multicast Delegate is the base class

```
// Define a custom delegate
public delegate void CustomDelegate(string message);
// Create instances of the delegate
CustomDelegate delegate1 = PrintMessage;
CustomDelegate delegate2 = DisplayMessage;

// Add the delegates to the invocation list
CustomDelegate combinedDelegate = delegate1 + delegate2;

// Invoke the delegates in the invocation list
combinedDelegate("Hello, world!");
```

Example

Action

Built in Delegate provided by the BCL. Has no return type and it can have up to 16 generic parameters . Action is typically used when you need to define a callback or perform an action without returning a value.

```
public static void Main(string[] args)
{
    Action<string> messageTarget;
    if (args.Length > 1) messageTarget = ShowWindowsMessage;
    else messageTarget = Console.WriteLine;
    messageTarget("Invoking Action!");
}

private static void ShowWindowsMessage(string message)
{
    MessageBox.Show(message);
}
```

Action example

```
delegate int AddDelegate(int a, int b);

static void Main(string[] args){
    AddDelegate ad = (a,b) => a + b;
    int result = ad(1,1); //result = 2
}

delegate bool LogDelegate();

static void Main(string[] args)
{
    LogDelegate ld = () =>
    {
        UpdateDatabase();
        WriteToEventLog();
        return true;
    };
    bool status = ld();
}
```

Event handler using lambda expression

Func

Built in Delegate provided by the BCL. Has a generic return type and it can have up to 15 generic parameters, the last generic parameter represents the return type. Func is often used when you need to define a callback that returns a value or when you want to pass a function as an argument to another method.

```
public static void Main(string[] args)
{
    Func<string, bool> logFunc;
    if (args[0] == "EventLog") logFunc = LogToEventLog;
    else logFunc = LogToFile;
    bool status = logFunc("Log Message");
}

private static bool LogToEventLog(string message) { /* log */ }
private static bool LogToFile(string message) { /*log */ }
```

Func Example

Func

Ejemplos de Action y Func

```
public static void Main(string[] args)
{
    Action<string> messageTarget;
    if (args.Length > 1) messageTarget = ShowWindowsMessage;
    else messageTarget = Console.WriteLine;
    messageTarget("Invoking Action!");
}

private static void ShowWindowsMessage(string message)
{
    MessageBox.Show(message);
}
```

Delegate usando Action, el paramétro de las referencias debe ser string

Asignando un método al delegate dependiendo de la condición

Invocando el método referenciado por el delegate

Método que cumple con la firma del delegate

Delegate usando Func, el paramétro de las referencias debe ser string y el retorno bool

Asignando un método al delegate dependiendo de la condición

Invocando el método referenciado por el delegate

Métodos que cumplen con la firma del delegate

```
public static void Main(string[] args)
{
    Func<string, bool> logFunc;
    if (args[0] == "EventLog") logFunc = LogToEventLog;
    else logFunc = LogToFile;
    bool status = logFunc("Log Message");
}

private static bool LogToEventLog(string message) { /* log */ }
private static bool LogToFile(string message) { /*log */ }
```

Lambda expressions

The Lambda operator (=>) in C# is used to create anonymous functions or delegates. Lambda expressions provide a concise way to write code by reducing the need for explicit method declarations.

Consist of three parts:

- Parameters: Specifies the input parameters of the lambda expression.
- Lambda Operator: The ”=>” operator separates the parameters from the expression body.
- Expression Body: Specifies the code to be executed when the lambda expression is invoked.

Provides: Concise syntax, inline definition, enhanced expresiveness

```csharp
Func<int, int> square = delegate(int x)
{
    return x * x;
};

int result = square(5); // Output: 25


Func<int, int> squareLambda = x => x * x;

int resultLambda = squareLambda(5); // Output: 25
```

Anonymous method syntax and Lambda expression

```fsharp
let numbers = [1; 2; 3; 4; 5]

// Filter even numbers using a Lambda expression
let evenNumbers = numbers |> List.filter (fun x -> x % 2 = 0)

// Output: [2; 4]
printfn "%A" evenNumbers
```

Lambdas in F#

```csharp
List<string> names = new List<string> { "John", "Alice", "Bob", "David" };

// Sort names in ascending order using a Lambda expression
List<string> sortedNames = names.OrderBy(x => x).ToList();

// Output: [Alice, Bob, David, John]
Console.WriteLine(string.Join(", ", sortedNames));
```

Using lambdas in LINQ

Anonymous types

Provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.

The type name is generated by the compiler and is not available at the source code level, the type of each property is inferred by the compiler.

Creating an Anonymous Type

```
var subset = new
{
    order.OrderNumber,
    order.Total,
    AveragePrice = order.LineItems.Average(item => item.Price)
};
```

Using **LINQ** with **Anonymous Types**

```
// Method Syntax
var totals = orders.Select(order => new { order.Total });
```

Anonymous types

Temporary class, use it to create a representation of data used in the current method.

Do not return an anonymous type or use it as a parameter, it is only to be used in the current method

Commonly used with LINQ, create a representation of a subset of the properties

Introducing the **with** Expression

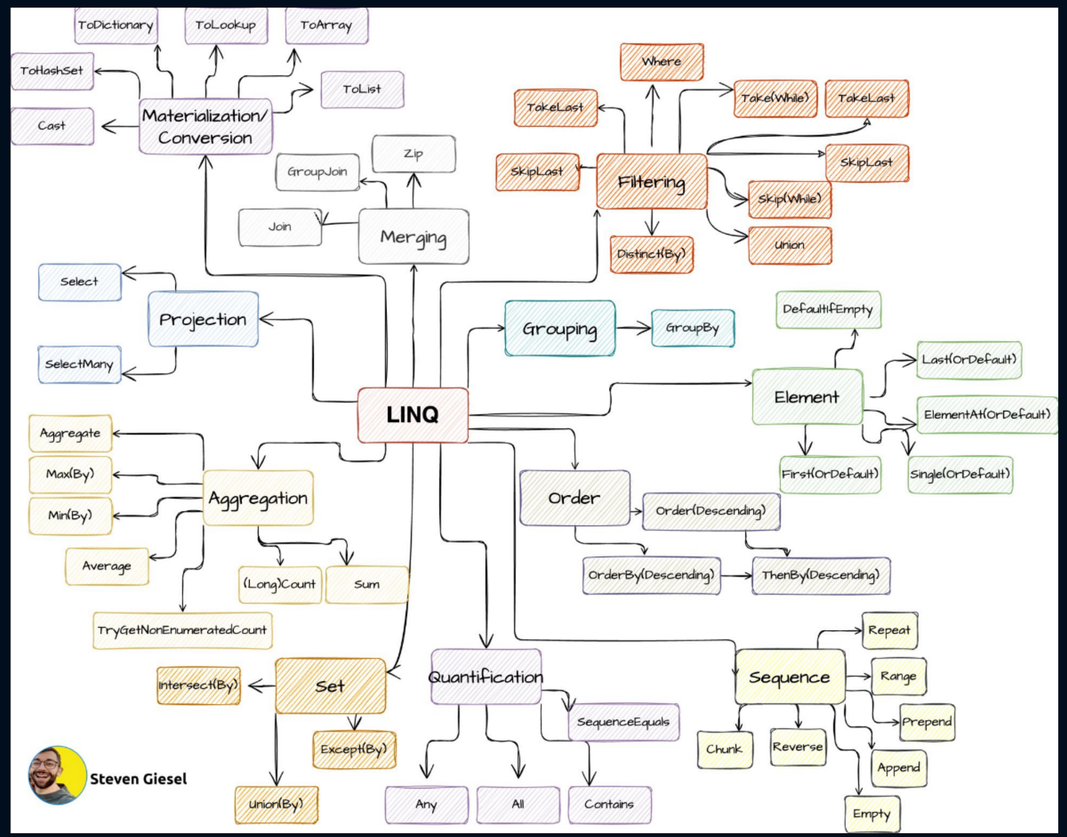
```
var instance = new
{
    order.OrderNumber,
    order.Total,
    AveragePrice = order.LineItems.Average(item => item.Price)
};

var copy = instance with { Total = 50 };
```

```
// <>f__AnonymousType0<<OrderNumber>j__TPar,<Total>j__TPar,<AveragePrice>j__TPar>
using ...

[CompilerGenerated]
[DebuggerDisplay("\\{ OrderNumber = {OrderNumber}, Total = {Total}, AveragePrice = {AveragePrice} }", Type
internal sealed class <>f__AnonymousType0<<OrderNumber>j__TPar, <Total>j__TPar, <AveragePrice>j__TPar>
{
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly <OrderNumber>j__TPar <OrderNumber>i__Field;

    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly <Total>j__TPar <Total>i__Field;

    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly <AveragePrice>j__TPar <AveragePrice>i__Field;

    public <OrderNumber>j__TPar OrderNumber
    ...

    public <Total>j__TPar Total
    ...

    public <AveragePrice>j__TPar AveragePrice
    ...
```

Anonymous types to intermediate language

LINQ

LINQ (Language Integrated Query) is a powerful feature in C# that allows you to query and manipulate data from different data sources using a unified syntax. It provides a consistent and expressive way to query collections, databases, XML documents, and other data sources.

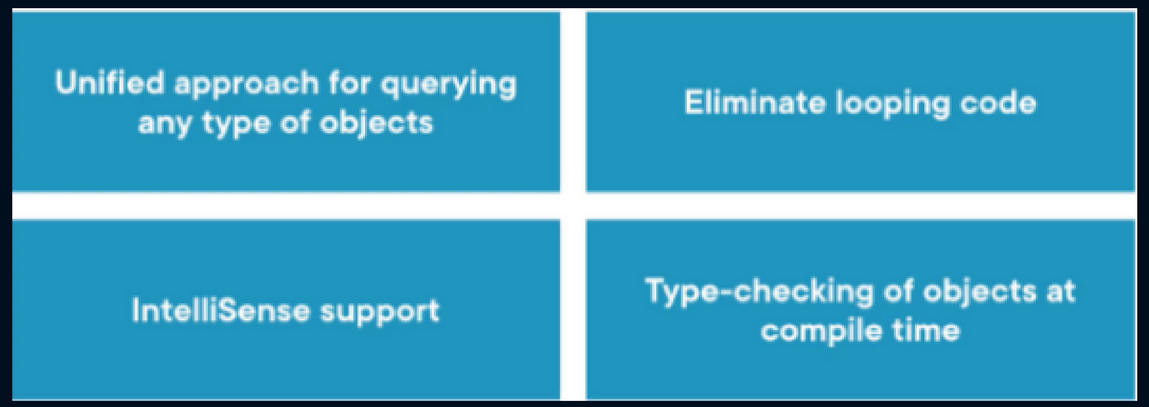


LINQ Mindmap

Why?

It offers several benefits that make it a valuable tool in software development:
Readability: LINQ uses a declarative syntax that closely resembles natural language.
Productivity: LINQ simplifies common data-related tasks, such as filtering, sorting, grouping, and transforming data
Type safety: LINQ is strongly typed, meaning it performs compile-time checking to ensure type safety
Integration with C# language features: LINQ seamlessly integrates with other language features of C#, such as lambda expressions and anonymous types.
Extensibility: LINQ is extensible, allowing developers to create custom query operators and providers.

LINQ use cases

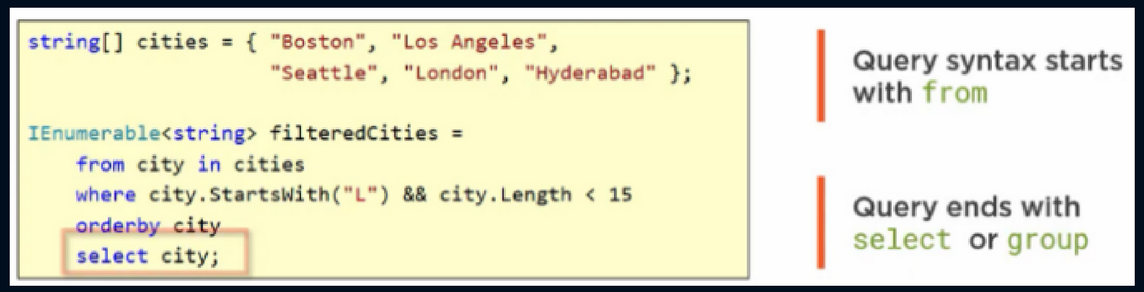


LINQ avantages

Syntaxis

Query syntax resembles SQL-like syntax and is designed to make queries more readable and expressive. It uses keywords such as from, where, select, group by, order by, and join to construct queries.

Method syntax, also known as fluent syntax, uses extension methods to chain together operations on collections. It is more concise and suited for functional programming style.



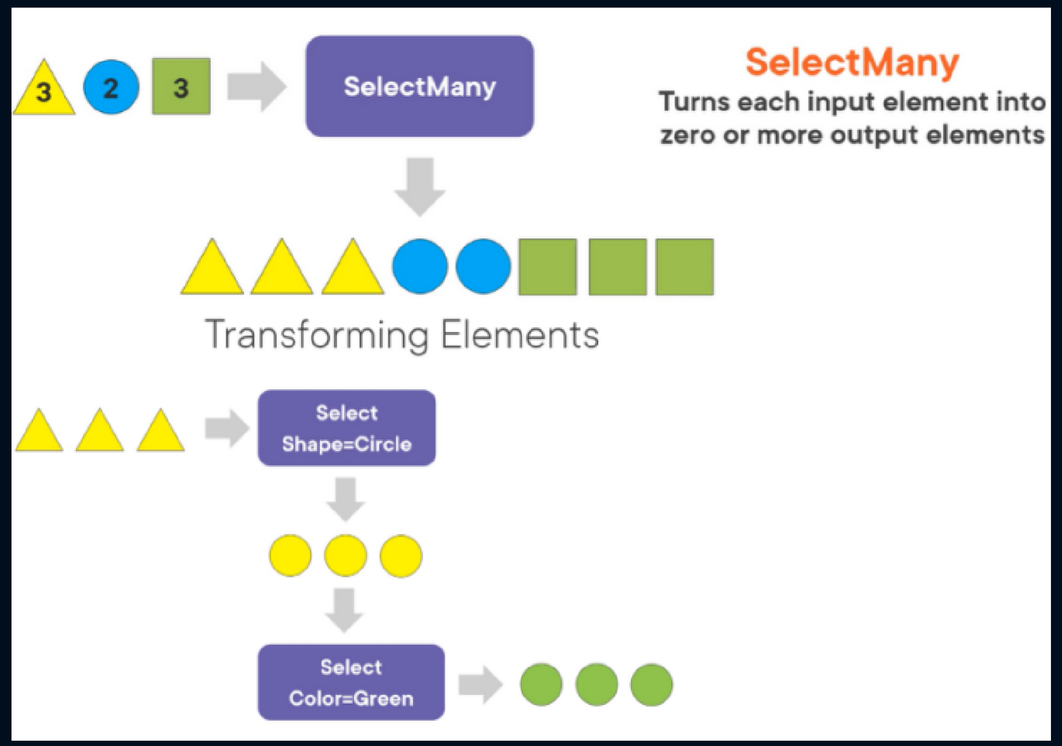
Query syntax

```
var query = developers.Where(e => e.Name.Length == 5)
                      .OrderByDescending(e => e.Name)
                      .Select(e => e);

var query2 = from developer in developers
             where developer.Name.Length == 5
             orderby developer.Name descending
             select developer;
```

Method syntax

Projections

Projections refer to the process of transforming the elements of a collection into a new form or shape. It allows you to extract and manipulate specific data from a collection based on your requirements. Projections in LINQ are typically achieved using the Select clause or method.



Select Many and select

```
var query =
    from car in cars
    where car.Manufacturer == "BMW" && car.Year == 2016
    orderby car.Combined descending, car.Name ascending
    select car;
```

Select example

Task #21

Exercise

Create a class called "EventManager" that manages a collection of events. The class should have methods to add and remove events, as well as a method to invoke all the registered events. Use **Func<string, bool>** delegates to represent the events. Write code to add multiple events of different types and invoke them, the order should be the following, the last event is the first being executed.

Every event will be represented as **Func<string, bool>** returns a bool that represents if the operation was successful and a string that holds the information of that operation (for complex operations consider using this string to represent objects using JSON)

The main goal is to use the EventManager as a playbook, to reproduce the events captured in its collection.