Lambda Expressions

.NET

Lambda expressions *are anonymous functions that contain expressions or a sequence of operators. It uses the lambda operator, =>.*
The left side of the lambda operator specifies the input parameters and the right side holds an expression or a code block that works with the input parameters.

HTTPS://WWW.C-SHARPCORNER.COM/UPLOADFILE/BD6C67/LAMBDA-EXPRESSIONS-IN-C-SHARP/

Lambda Expression

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

---

There are two Lambda expression forms.

- Expression Lambda - `(input-parameters) => expression`
- Statement Lambda - `(input-parameters) => { <sequence-of-statements> }`

Specify input parameters to the left of the => and an expression or statement block to the right.

Lambda Statement Form

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions#stateme0nt-lambdas

---

The body of a statement lambda can consist of any number of statements.

Statement lambdas cannot be used to create expression trees.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Lambda Expression Form

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions#expression-lambdas

---

Expression lambdas are used extensively in the construction of expression trees. An expression lambda returns the result of the expression.

- Zero parameters have empty ( ).

```
Action line = () => Console.WriteLine();
```

- >2 parameters are separated by commas enclosed in ( ).

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

- You can optionally specify the types explicitly.

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

- Parameter types must be all explicit or all implicit.

Lambda Conversion to Delegate

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

- Any ***lambda expression*** can be converted to a ***delegate*** type.
- The ***delegate*** type to which a ***lambda expression*** can be converted is defined by the ***types*** of its parameters and return value.
- If a ***lambda expression*** doesn't return a value,
  - It can be converted to one of the ***Action<> delegate types***.
    - A ***lambda expression*** that has two parameters and returns void can be converted to an ***Action<T1,T2> delegate***.
- If a ***lambda expression*** returns a value,
  - It can be converted to a ***Func<> delegate type***.
    - A ***lambda expression*** that has one parameter and returns a value can be converted to a ***Func<T,TResult> delegate***.

Lambda Conversions - Examples

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

---

The lambda expression x => x * x, which specifies a parameter, x, and returns x*x, is assigned to a variable of a delegate type.

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

You can use lambda expressions when you use ***LINQ.***

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Lambda Async/Await

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions#async-lambdas

---

Create lambda expressions and statements that incorporate ***asynchronous*** processing by using the async and await keywords.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Predicate<T> Delegate

https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-5.0

A *Predicate <T>* variable represents a method that defines a set of criteria and iterates over a set to determine whether a specified object in that set meets those criteria.

This type parameter is contravariant (of the type specified or less derived).

Returns Boolean ***true*** if the obj meets the criteria defined within the method represented by this delegate; otherwise returns, ***false***.

```
using System;
using System.Drawing;

public class Example
{
   public static void Main()
   {
      // Create an array of Point structures.
      Point[] points = { new Point(100, 200),
                         new Point(150, 250), new Point(250, 375),
                         new Point(275, 395), new Point(295, 450) };

      // Define the Predicate<T> delegate.
      Predicate<Point> predicate = FindPoints;

      // Find the first Point structure for which X times Y
      // is greater than 100000.
      Point first = Array.Find(points, predicate);

      // Display the first structure found.
      Console.WriteLine("Found: X = {0}, Y = {1}", first.X, first.Y);
   }

   private static bool FindPoints(Point obj)
   {
      return obj.X * obj.Y > 100000;
   }
}
// The example displays the following output:
//        Found: X = 275, Y = 395
```

Predicate<T> Delegate

https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-5.0

It's customary (and much more common) to use a ***lambda expression*** rather than to explicitly define a delegate of type Predicate<T>.

Each element of the points array is passed to the lambda expression until the expression finds an element that meets the search criteria. The lambda expression returns ***true*** if the product of the x and y fields is greater than 100,000.

```
using System;
using System.Drawing;

public class Example
{
   public static void Main()
   {
      // Create an array of Point structures.
      Point[] points = { new Point(100, 200),
                         new Point(150, 250), new Point(250, 375),
                         new Point(275, 395), new Point(295, 450) };

      // Find the first Point structure for which X times Y
      // is greater than 100000.
      Point first = Array.Find(points, x => x.X * x.Y > 100000 );

      // Display the first structure found.
      Console.WriteLine("Found: X = {0}, Y = {1}", first.X, first.Y);
   }
}
// The example displays the following output:
//        Found: X = 275, Y = 395
```