Lambda expressions (C# reference)

11/08/202115 minutes to read

Is this page helpful?

In this article

Expression lambdas
Statement lambdas
Input parameters of a lambda expression
Async lambdas
Lambda expressions and tuples
Lambdas with the standard query operators
Type inference in lambda expressions
Natural type for lambda expressions
Declared return type
Attributes
Capture of outer variables and variable scope in lambda expressions
C# language specification
See also

You use a *lambda expression* 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:

  C# Copy

  ```
  (input-parameters) => expression
  ```

- Statement lambda that has a statement block as its body:

  C# Copy

  ```
  (input-parameters) => { <sequence-of-statements> }
  ```

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

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; otherwise, it can be converted to one of the `Func` delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an Action<T1,T2> delegate. A lambda expression that has one parameter and returns a value can be converted to a Func<T,TResult> delegate. In the following example, the lambda expression `x =>`

`x * x`, which specifies a parameter that's named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

C# Copy Run

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

Expression lambdas can also be converted to the expression tree types, as the following example shows:

C# Copy Run

```csharp
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

You can use lambda expressions in any code that requires instances of delegate types or expression trees, for example as an argument to the Task.Run(Action) method to pass the code that should be executed in the background. You can also use lambda expressions when you write LINQ in C#, as the following example shows:

C# Copy Run

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

When you use method-based syntax to call the Enumerable.Select method in the System.Linq.Enumerable class, for example in LINQ to Objects and LINQ to XML, the parameter is a delegate type System.Func<T,TResult>. When you call the Queryable.Select method in the System.Linq.Queryable class, for example in LINQ to SQL, the parameter type is an expression tree type Expression<Func<TSource,TResult>>. In both cases, you can use the same lambda expression to specify the parameter value. That makes the two `Select` calls to look similar although in fact the type of objects created from the lambdas is different.

Expression lambdas

A lambda expression with an expression on the right side of the `=>` operator is called an *expression lambda*. An expression lambda returns the result of the expression and takes the following basic form:

C# Copy

```
(input-parameters) => expression
```

The body of an expression lambda can consist of a method call. However, if you're creating expression trees that are evaluated outside the context of the .NET Common Language Runtime (CLR), such as in SQL Server, you shouldn't use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET Common Language Runtime (CLR).

Statement lambdas

A statement lambda resembles an expression lambda except that its statements are enclosed in braces:

C# Copy

```
(input-parameters) => { <sequence-of-statements> }
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

C# Copy Run

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

You can't use statement lambdas to create expression trees.

Input parameters of a lambda expression

You enclose input parameters of a lambda expression in parentheses. Specify zero input parameters with empty parentheses:

C# Copy

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

If a lambda expression has only one input parameter, parentheses are optional:

C# Copy

```
Func<double, double> cube = x => x * x * x;
```

Two or more input parameters are separated by commas:

C# Copy

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

Sometimes the compiler can't infer the types of input parameters. You can specify the types explicitly as shown in the following example:

C# Copy

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

Input parameter types must be all explicit or all implicit; otherwise, a CS0748 compiler error occurs.

Beginning with C# 9.0, you can use discards to specify two or more input parameters of a lambda expression that aren't used in the expression:

C# Copy

```
Func<int, int, int> constant = (_, _) => 42;
```

Lambda discard parameters may be useful when you use a lambda expression to provide an event handler.

Note

For backwards compatibility, if only a single input parameter is named `_`, then, within a lambda expression, `_` is treated as the name of that parameter.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the async and await keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

C# Copy

```
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);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows:

C# Copy

```
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);
    }
}
```

For more information about how to create and use async methods, see Asynchronous Programming with async and await.

Lambda expressions and tuples

Starting with C# 7.0, the C# language provides built-in support for tuples. You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple components.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

C#

Copy Run

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2,
2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, and so on. You can, however, define a tuple with named components, as the following example does.

C#

Copy Run

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 *
ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

For more information about C# tuples, see Tuple types.

Lambdas with the standard query operators

LINQ to Objects, among other implementations, have an input parameter whose type is one of the Func<TResult> family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the Func<T,TResult> delegate type:

C#

Copy

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a `Func<int, bool>` instance where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. For example, `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it's invoked, returns Boolean value that indicates whether the input parameter is equal to five:

C#

Copy Run

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

You can also supply a lambda expression when the argument type is an Expression<TDelegate>, for example in the standard query operators that are defined in the Queryable type. When you specify an Expression<TDelegate> argument, the lambda is compiled to an expression tree.

The following example uses the Count standard query operator:

C# Copy Run

```csharp
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",
numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following example produces a sequence that contains all elements in the `numbers` array that precede the 9, because that's the first number in the sequence that doesn't meet the condition:

C# Copy Run

```csharp
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

The following example specifies multiple input parameters by enclosing them in parentheses. The method returns all the elements in the `numbers` array until it finds a number whose value is less than its ordinal position in the array:

C# Copy Run

```csharp
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

You don't use lambda expressions directly in query expressions, but you can use them in method calls within query expressions, as the following example shows:

C# Copy Run

```csharp
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you're querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

C# Copy

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Natural type for lambda expressions

Lambda expressions in themselves don't have a type because the common type system has no intrinsic concept of "lambda expression." However, it's sometimes convenient to speak informally of the "type" of a lambda expression. That informal "type" refers to the delegate type or Expression type to which the lambda expression is converted.

Beginning with C# 10, some lambda expressions have a *natural type*. Instead of forcing you to declare a delegate type, such as `Func<...>` or `Action<...>` for a lambda expression, the compiler may infer the delegate type from the parameters and the type of the expression. For example, consider the following declaration:

C# Copy

```csharp
var parse = (string s) => int.Parse(s);
```

The compiler can infer `parse` to be a `Func<string, int>`. In general, the compiler will use an available `Func` or `Action` delegate, if a suitable one exists. Otherwise, it will synthesize a delegate type. For example, the type must be synthesized if the lambda expression has `ref` parameters. When a lambda expression has a natural type, it can be assigned to a less explicit type, such as System.Object, or System.Delegate:

C# Copy

```csharp
object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Method groups (that is, method names without argument lists) with exactly one overload have a natural type:

C# Copy

```csharp
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

If you assign a lambda expression to System.Linq.Expressions.LambdaExpression, or System.Linq.Expressions.Expression, and the lambda has a natural delegate type, the expression has a natural type of System.Linq.Expressions.Expression<TDelegate>, with the natural delegate type used as the argument for the type parameter:

C# Copy

```csharp
LambdaExpression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       // Expression<Func<string, int>>
```

Many lambda expressions won't have a natural type. Consider the following declaration:

C# Copy

```csharp
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

The compiler can't infer a parameter type for `s`. When the compiler can't infer a natural type, you must declare the type:

C# Copy

```csharp
Func<string, int> parse = s => int.Parse(s);
```

Declared return type

Typically, the return type of the lambda expression is obvious and inferred. For some expressions, that wouldn't work:

c# Copy

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

Beginning with C# 10, you can specify the return type on a lambda expression before the parameters. When you specify an explicit return type, the parameters must be parenthesized, so that it's not too confusing to the compiler or other developers:

c# Copy

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Attributes

Beginning with C# 10, you can apply attributes to lambda expressions. The attributes are added before the parameter declaration. The lambda expression's parameter list must be parenthesized when there are attributes:

c# Copy

```
Func<string, int> parse = [Example(1)] (s) => int.Parse(s);
var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";
```

You can apply any attribute that is valid on AttributeTargets.Method.

Lambda expressions are invoked through the underlying delegate type. That is different than methods and local functions. The delegate's `Invoke` method won't check attributes on the lambda expression. Attributes don't have any effect when the lambda expression is invoked. Attributes on lambda expressions are useful for code analysis, and can be discovered via reflection. One consequence of this decision is the System.Diagnostics.ConditionalAttribute cannot be applied to a lambda expression.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

C#

Copy

```csharp
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable:
{equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured won't be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression aren't visible in the enclosing method.
- A lambda expression can't directly capture an in, ref, or out parameter from the enclosing method.
- A return statement in a lambda expression doesn't cause the enclosing method to return.
- A lambda expression can't contain a goto, break, or continue statement if the target of that jump statement is outside the lambda expression block. It's also an error to have a jump statement outside the lambda expression block if the target is inside the block.

Beginning with C# 9.0, you can apply the `static` modifier to a lambda expression to prevent unintentional capture of local variables or instance state by the lambda:

C# Copy

```csharp
Func<double, double> square = static x => x * x;
```

A static lambda can't capture local variables or instance state from enclosing scopes, but may reference static members and constant definitions.

C# language specification

For more information, see the Anonymous function expressions section of the C# language specification.

For more information about features added in C# 9.0, see the following feature proposal notes:

- Lambda discard parameters
- Static anonymous functions

See also

- C# reference
- C# operators and expressions
- LINQ (Language-Integrated Query)
- Expression Trees
- Local functions vs. lambda expressions
- Visual Studio 2008 C# Samples (see LINQ Sample Queries files and XQuery program)