Visual Studio 2010 - Visual C#

**Lambda Expressions (C# Programming Guide)**

A *lambda expression* is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the lambda operator [=>](http://msdn.microsoft.com/en-us/library/bb311046.aspx), which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression x => x \* x is read "x goes to x times x." This expression can be assigned to a delegate type as follows:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl02_code');" \o "Copy Code)

delegate int del(int i);

static void Main(string[] args)

{

del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

}

To create an expression tree type:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

using System.Linq.Expressions;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

Expression<del> myET = x => x \* x;

}

}

}

The **=>** operator has the same precedence as assignment (**=**) and is right-associative.

Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx).

When you use method-based syntax to call the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) method in the [Enumerable](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.aspx) class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type [System..::.Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx). A lambda expression is the most convenient way to create that delegate. When you call the same method in, for example, the [System.Linq..::.Queryable](http://msdn.microsoft.com/en-us/library/system.linq.queryable.aspx) class (as you do in LINQ to SQL) then the parameter type is an [System.Linq.Expressions..::.Expression](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx)<Func> where Func is any Func delegates with up to sixteen input parameters. Again, a lambda expression is just a very concise way to construct that expression tree. The lambdas allow the **Where** calls to look similar although in fact the type of object created from the lambda is different.

In the previous example, notice that the delegate signature has one implicitly-typed input parameter of type **int**, and returns an **int**. The lambda expression can be converted to a delegate of that type because it also has one input parameter (x) and a return value that the compiler can implicitly convert to type **int**. (Type inference is discussed in more detail in the following sections.) When the delegate is invoked by using an input parameter of 5, it returns a result of 25.

Lambdas are not allowed on the left side of the [is](http://msdn.microsoft.com/en-us/library/scekt9xw.aspx) or [as](http://msdn.microsoft.com/en-us/library/cscsdfbt.aspx) operator.

All restrictions that apply to anonymous methods also apply to lambda expressions. For more information, see [Anonymous Methods (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifExpression Lambdas

A lambda expression with an expression on the right side is called an *expression lambda*. Expression lambdas are used extensively in the construction of [Expression Trees (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/bb397951.aspx). An expression lambda returns the result of the expression and takes the following basic form:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl69_ctl00_ctl01_code');" \o "Copy Code)

(input parameters) => expression

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl69_ctl00_ctl02_code');" \o "Copy Code)

(x, y) => x == y

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl69_ctl00_ctl03_code');" \o "Copy Code)

(int x, string s) => s.Length > x

Specify zero input parameters with empty parentheses:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl69_ctl00_ctl04_code');" \o "Copy Code)

() => SomeMethod()

Note in the previous example that the body of an expression lambda can consist of a method call. However, if you are creating expression trees that will be consumed in another domain, such as SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifStatement Lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl70_ctl00_ctl00_code');" \o "Copy Code)

(input parameters) => {statement;}

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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl70_ctl00_ctl01_code');" \o "Copy Code)

delegate void TestDelegate(string s);

…

TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };

myDel("Hello");

Statement lambdas, like anonymous methods, cannot be used to create expression trees.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifLambdas with the Standard Query Operators

Many Standard query operators have an input parameter whose type is one of the [Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx) family of generic delegates. The [Func<(Of <(T, TResult>)>)](http://msdn.microsoft.com/en-us/library/bb549151.aspx) delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. **Func** delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the following delegate type:

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl71_ctl00_ctl02_code');" \o "Copy Code)

public delegate TResult Func<TArg0, TResult>(TArg0 arg0)

The delegate can be instantiated as Func<int,bool> myFunc where **int** is an input parameter and **bool** is the return value. The return value is always specified in the last type parameter. **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 is invoked, will return true or false to indicate whether the input parameter is equal to 5:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl71_ctl00_ctl03_code');" \o "Copy Code)

Func<int, bool> myFunc = x => x == 5;

bool result = myFunc(4); // returns false of course

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

A standard query operator, the [Count](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count.aspx) method, is shown here:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl71_ctl00_ctl05_code');" \o "Copy Code)

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

int oddNumbers = numbers.Count(n => n % 2 == 1);

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 method will produce a sequence that contains all the elements in the numbers array that are to the left of the 9, because that is the first number in the sequence that does not meet the condition:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl71_ctl00_ctl06_code');" \o "Copy Code)

var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);

This example shows how to specify multiple input parameters by enclosing them in parentheses. The method returns all the elements in the numbers array until a number is encountered whose value is less than its position. Do not confuse the lambda operator (**=>**) with the greater than or equal operator (**>=**).

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl71_ctl00_ctl07_code');" \o "Copy Code)

var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifType Inference in Lambdas

When writing lambdas, you often do not have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the underlying delegate type, 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. So if you are 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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl72_ctl00_ctl00_code');" \o "Copy Code)

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

The general rules 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.

Note that lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or [Expression](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx) type to which the lambda expression is converted.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifVariable Scope in Lambda Expressions

Lambdas can refer to *outer variables* that are in scope in the enclosing method or type in which the lambda is defined. Variables that are captured in this manner are stored for use in the lambda expression even if 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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl73_ctl00_ctl00_code');" \o "Copy Code)

delegate bool D();

delegate bool D2(int i);

class Test

{

D del;

D2 del2;

public void TestMethod(int input)

{

int j = 0;

// Initialize the delegates with lambda expressions.

// Note access to 2 outer variables.

// del will be invoked within this method.

del = () => { j = 10; return j > input; };

// del2 will be invoked after TestMethod goes out of scope.

del2 = (x) => {return x == j; };

// Demonstrate value of j:

// Output: j = 0

// The delegate has not been invoked yet.

Console.WriteLine("j = {0}", j); // Invoke the delegate.

bool boolResult = del();

// Output: j = 10 b = True

Console.WriteLine("j = {0}. b = {1}", j, boolResult);

}

static void Main()

{

Test test = new Test();

test.TestMethod(5);

// Prove that del2 still has a copy of

// local variable j from TestMethod.

bool result = test.del2(10);

// Output: True

Console.WriteLine(result);

Console.ReadKey();

}

}

The following rules apply to variable scope in lambda expressions:

* A variable that is captured will not be garbage-collected until the delegate that references it goes out of scope.
* Variables introduced within a lambda expression are not visible in the outer method.
* A lambda expression cannot directly capture a **ref** or **out** parameter from an enclosing method.
* A return statement in a lambda expression does not cause the enclosing method to return.
* A lambda expression cannot contain a **goto** statement, **break** statement, or **continue** statement whose target is outside the body or in the body of a contained anonymous function.