.NET/C# Syntax.

Modern .NET/C# Course

.NET Academy

Content

- Methods.
  - Delegates.
  - Lambda expressions.
  - Default Values for Lambda Parameters.
  - Static modifier for Lambda Expression.
- Input parameters: in, out, ref.

.NET Academy

Methods.

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

The Main method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started. In an application that uses top-level statements, the Main method is generated by the compiler and contains all top-level statements.

Methods.

Methods are declared in a class, struct, or interface by specifying the access level such as public or private, optional modifiers such as abstract or sealed, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method.

By default, when an instance of a value type is passed to a method, its copy is passed instead of the instance itself. Therefore, changes to the argument have no effect on the original instance in the calling method. To pass a value-type instance by reference, use the ref keyword. For more information, see Passing Value-Type Parameters.

Methods.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

Methods can return a value to the caller. If the return type (the type listed before the method name) is not void, the method can return the value by using the return statement. A statement with the return keyword followed by a value that matches the return type will return that value to the method caller.

Methods.

The value can be returned to the caller by value or by reference. Values are returned to the caller by reference if the ref keyword is used in the method signature and it follows each return keyword. For example, the following method signature and return statement indicate that the method returns a variable named estDistance by reference to the caller.

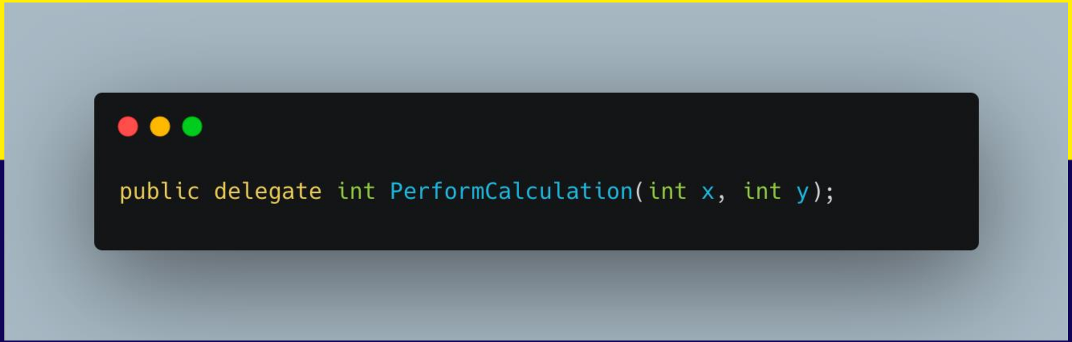
The return keyword also stops the execution of the method. If the return type is void, a return statement without a value is still useful to stop the execution of the method. Without the return keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the return keyword to return a value.

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using =>.

.NET Academy

Delegates.

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.
Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:



```
public delegate int PerformCalculation(int x, int y);
```

.NET Academy

Delegates.

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. You can write a method that compares two objects in your application. That method can be used in a delegate for a sort algorithm. Because the comparison code is separate from the library, the sort method can be more general.

Function pointers were added to C# 9 for similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

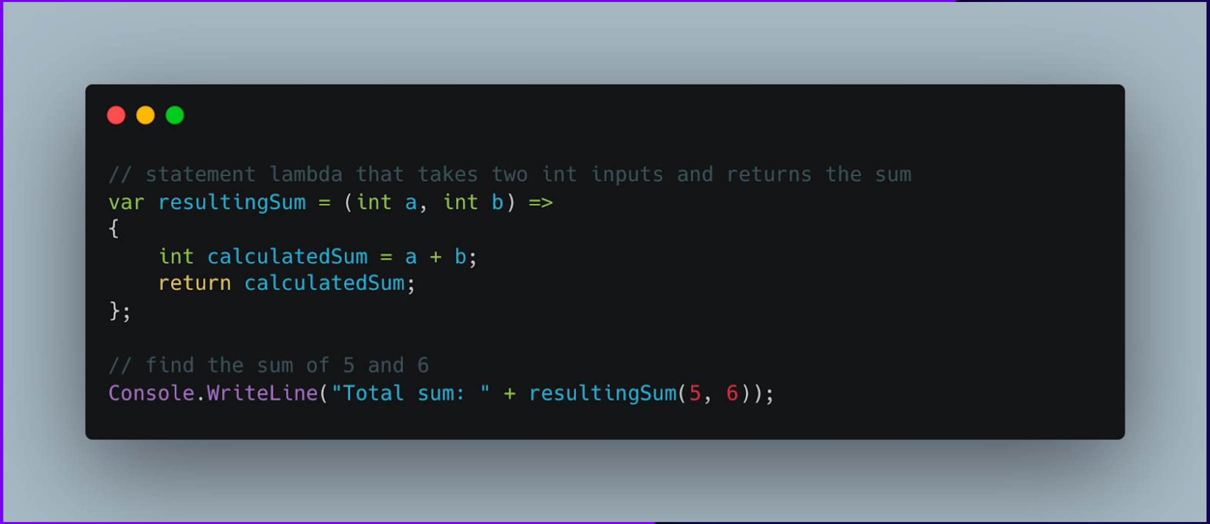
.NET Academy

Lambda.

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.

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.

Lambda.



```
// statement lambda that takes two int inputs and returns the sum
var resultingSum = (int a, int b) =>
{
    int calculatedSum = a + b;
    return calculatedSum;
};

// find the sum of 5 and 6
Console.WriteLine("Total sum: " + resultingSum(5, 6));
```

.NET Academy

Default Values for Lambda Parameters.

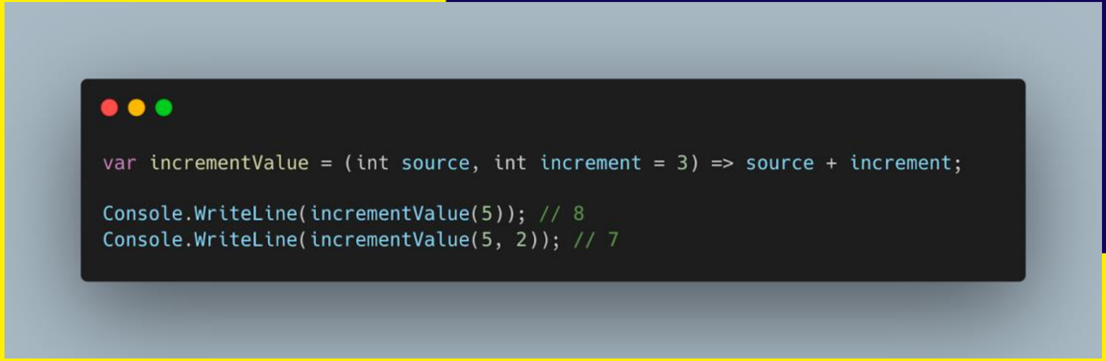
Default Values for Lambda Expression is one of the coolest features from C# 12. This feature is similar to default values for regular method parameters, and it allows to create more flexible and concise lambda expressions.

With C# 12, we can have a default value for lambda expressions just by adding an equal sign and the value you want to assign to the parameter after its declaration.

.NET Academy

Default Values for Lambda Parameters.

For example, int increment equals 3 Lambda expression, 3 set as a default value for increment which add with the source parameter when no value is provided. When the lambda expression is called with this, you won't have to rely on the system's default parameter value. It makes the code more readable as well.



```
var incrementValue = (int source, int increment = 3) => source + increment;

Console.WriteLine(incrementValue(5)); // 8
Console.WriteLine(incrementValue(5, 2)); // 7
```

.NET Academy

Static modifier for Lambda Expression.

Allow a 'static' modifier on lambdas and anonymous methods, which disallows capture of locals or instance state from containing scopes.

Avoid unintentionally capturing state from the enclosing context, which can result in unexpected retention of captured objects or unexpected additional allocations.

A lambda or anonymous method may have a static modifier. The static modifier indicates that the lambda or anonymous method is a static anonymous function. A static anonymous function cannot capture state from the enclosing scope. As a result, locals, parameters, and this from the enclosing scope are not available within a static anonymous function.

.NET Academy

Static modifier for Lambda Expression.

A static anonymous function cannot reference instance members from an implicit or explicit this or base reference. A static anonymous function may reference static members from the enclosing scope. A static anonymous function may reference constant definitions from the enclosing scope. nameof() in a static anonymous function may reference locals, parameters, or this or base from the enclosing scope. Accessibility rules for private members in the enclosing scope are the same for static and non-static anonymous functions. No guarantee is made as to whether a static anonymous function definition is emitted as a static method in metadata. This is left up to the compiler implementation to optimize. A non-static local function or anonymous function can capture state from an enclosing static anonymous function but cannot capture state outside the enclosing static anonymous function. Removing the static modifier from an anonymous function in a valid program does not change the meaning of the program.

Input parameters: in, out, ref.

Method parameters have modifiers available to change the desired outcome of how the parameter is treated. Each method has a specific use case:

- **ref** is used to state that the parameter passed may be modified by the method.
- **in** is used to state that the parameter passed cannot be modified by the method.
- **out** is used to state that the parameter passed must be modified by the method.

Both the **ref** and **in** require the parameter to have been initialized before being passed to a method. The **out** modifier does not require this and is typically not initialized prior to being used in a method.

Questions for the study of the topic

1. What is the purpose of the return statement in a method, and how does it affect the method's execution?
2. What is the difference between value types and reference types in C# regarding method parameter passing?
3. Can you provide an example of how delegates can be used to implement event handling in C#?
4. Explain the concept of multicast delegates and how they can be used to call multiple methods with a single delegate instance.
5. How can lambda expressions make code more concise and readable in comparison to traditional methods?
6. How does the use of default values for lambda parameters affect the behavior of lambda expressions in C# 12?
7. Explain the key differences between the in, out, and ref modifiers for method parameters and when to use each.

Homework

1. Create a C# console application demonstrating the use of a method. The method should accept two integers as input parameters and return their sum. Call this method from the Main method and display the result on the screen.
2. Implement three methods, each performing different operations (e.g., addition, subtraction, multiplication), using a delegate. Test these methods by calling them through the delegate and display the results on the screen.
3. Create a lambda expression that includes the static modifier. Discuss and document how this modifier affects the lambda's ability to grab variables from the scope. Test its behaviour with instance variables and static variables.
4. Create another method that accepts an integer parameter with the 'in' modifier. Attempt to modify the 'in' parameter inside the method. Discuss the results and explain why the 'in' modifier prevents modification.

Thank you for your attention.

.NET Academy