C# - Anonymous Method

As the name suggests, an anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type.

Example: Anonymous Method

public delegate void Print(int value);

static void Main(string[] args)

{

Print print = delegate(int val) {

Console.WriteLine("Inside Anonymous method. Value: {0}", val);

};

print(100);

}

The concept of anonymous method was introduced in C# 2.0. An anonymous method is inline unnamed method in the code. It is created using the delegate keyword and doesn’t required name and return type. Hence we can say, an anonymous method has only body without name, optional parameters and return type. An anonymous method behaves like a regular method and allows us to write inline code in place of explicitly named methods.

## **A Simple Anonymous Method Example**

delegate int MathOp(int a, int b);

class Program

{

*//delegate for representing anonymous method*

delegate int del(int x, int y);

static void Main(string[] args)

{

*//anonymous method using delegate keyword*

del d1 = delegate(int x, int y) { return x \* y; };

int z1 = d1(2, 3);

Console.WriteLine(z1);

}

}

*//output:*

6

Anonymous methods can access variables defined in an outer function.

Example: Anonymous Method

public delegate void Print(int value);

static void Main(string[] args)

{

int i = 10;

Print prnt = delegate(int val) {

val += i;

Console.WriteLine("Anonymous method: {0}", val);

};

prnt(100);

}

The scope of the parameters of an anonymous method is the *anonymous-method-block*.

It is an error to have a jump statement, such as [goto](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto), [break](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/break), or [continue](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/continue), inside the anonymous method block if the target is outside the block. It is also an error to have a jump statement, such as goto, break, or continue, outside the anonymous method block if the target is inside the block.

The local variables and parameters whose scope contains an anonymous method declaration are called *outer*variables of the anonymous method. For example, in the following code segment, n is an outer variable:

int n = 0;

Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };

A reference to the outer variable n is said to be *captured* when the delegate is created. Unlike local variables, the lifetime of a captured variable extends until the delegates that reference the anonymous methods are eligible for garbage collection.

An anonymous method cannot access the [in](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in), [ref](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref) or [out](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier) parameters of an outer scope.

No unsafe code can be accessed within the *anonymous-method-block*.

Anonymous methods are not allowed on the left side of the [is](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/is) operator.

## Example

The following example demonstrates two ways of instantiating a delegate:

* Associating the delegate with an anonymous method.
* Associating the delegate with a named method (DoWork).

In each case, a message is displayed when the delegate is invoked.

// Declare a delegate.

delegate void Printer(string s);

class TestClass

{

static void Main()

{

// Instantiate the delegate type using an anonymous method.

Printer p = delegate(string j)

{

System.Console.WriteLine(j);

};

// Results from the anonymous delegate call.

p("The delegate using the anonymous method is called.");

// The delegate instantiation using a named method "DoWork".

p = DoWork;

// Results from the old style delegate call.

p("The delegate using the named method is called.");

}

// The method associated with the named delegate.

static void DoWork(string k)

{

System.Console.WriteLine(k);

}

}

/\* Output:

The delegate using the anonymous method is called.

The delegate using the named method is called.

\*/

Anonymous methods can also be passed to a method that accepts the delegate as a parameter.

In the following example, PrintHelperMethod() takes the first parameters of the Print delegate:

Example: Anonymous Method as Parameter

public delegate void Print(int value);

class Program

{

public static void PrintHelperMethod(Print printDel,int val)

{

val += 10;

printDel(val);

}

static void Main(string[] args)

{

PrintHelperMethod(delegate(int val) { Console.WriteLine("Anonymous method: {0}", val); }, 100);

}

}

The following example demonstrates the evolution of delegate creation from C# 1.0 to C# 3.0:

class Test

{

delegate void TestDelegate(string s);

static void M(string s)

{

Console.WriteLine(s);

}

static void Main(string[] args)

{

// Original delegate syntax required

// initialization with a named method.

TestDelegate testDelA = new TestDelegate(M);

// C# 2.0: A delegate can be initialized with

// inline code, called an "anonymous method." This

// method takes a string as an input parameter.

TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

// C# 3.0. A delegate can be initialized with

// a lambda expression. The lambda also takes a string

// as an input parameter (x). The type of x is inferred by the compiler.

TestDelegate testDelC = (x) => { Console.WriteLine(x); };

// Invoke the delegates.

testDelA("Hello. My name is M and I write lines.");

testDelB("That's nothing. I'm anonymous and ");

testDelC("I'm a famous author.");

// Keep console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

Hello. My name is M and I write lines.

That's nothing. I'm anonymous and

I'm a famous author.

Press any key to exit.

\*/

Anonymous methods can be used as event handlers:

Example: Anonymous Method as Event Handler

saveButton.Click += delegate(Object o, EventArgs e)

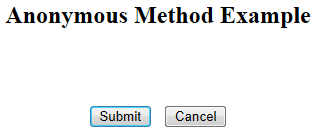
{

System.Windows.Forms.MessageBox.Show("Save Successfully!");

};

## **Anonymous Method as an Event Handler**

1. <form id="form1" runat="server">
2. <div align="center">
3. <h2>Anonymous Method Example</h2>
4. <br />
5. <asp:Label ID="lblmsg" runat="server" ForeColor="Green" Font-Bold="true"></asp:Label>
6. <br /><br />
7. <asp:Button ID="btnSubmit" runat="server" Text="Submit" />
8. <asp:Button ID="btnCancel" runat="server" Text="Cancel" />
9. </div>
10. </form>



protected void Page\_Load(object sender, EventArgs e)

{

*// Click Event handler using Regular method*

btnCancel.Click += new EventHandler(ClickEvent);

*// Click Event handler using Anonymous method*

btnSubmit.Click += delegate { lblmsg.Text="Submit Button clicked using Anonymous method"; };

}

protected void ClickEvent(object sender, EventArgs e)

{

lblmsg.Text="Cancel Button clicked using Regular method";

}

## **Anonymous Method Limitations**

* It cannot contain jump statement like goto, break or continue.
* It cannot access ref or out parameter of an outer method.
* It cannot have or access unsafe code.
* It cannot be used on the left side of the is operator.

 Points to Remember :

1. Anonymous method can be defined using the delegate keyword
2. Anonymous method must be assigned to a delegate.
3. Anonymous method can access outer variables or functions.
4. Anonymous method can be passed as a parameter.
5. Anonymous method can be used as event handlers.

**What is an anonymous method?**  
In simple terms, anonymous method is a method without a name.  
  
  
  
**Let's understand how a method can exist without a name**

usingSystem**;**

usingSystem.Collections.Generic**;**

classProgram

**{**

publicstaticvoidMain**()**

**{**

List<Employee>listEmployees=newList<Employee>**()**

**{**

newEmployee**{**ID=101**,**Name="Mark"**},**

newEmployee**{**ID=102**,**Name="John"**},**

newEmployee**{**ID=103**,**Name="Mary"**},**

**};**

// Step 2: Create an instance of Predicate<Employee>

// delegate and pass the method name as an argument

// to the delegate constructor

Predicate<Employee>predicateEmployee=

newPredicate<Employee>**(**FindEmployee**);**

// Step 3: Now pass the delegate instance as

// the argument for Find() method

Employeeemployee=

listEmployees.Find**(**x=>predicateEmployee**(**x**));**

Console.WriteLine**(**"ID = {0}, Name {1}"**,**

employee.ID**,**employee.Name**);**

// Anonymous method is being passed as an argument to

// the Find() method. This anonymous method replaces

// the need for Step 1, 2 and 3

employee=listEmployees.Find**(**delegate**(**Employeex**)**

**{**returnx.ID==102**; });**

Console.WriteLine**(**"ID = {0}, Name {1}"**,**

employee.ID**,**employee.Name**);**

**}**

// Step 1: Create a method whose signature matches

// with the signature of Predicate<Employee> delegate

privatestaticboolFindEmployee**(**EmployeeEmp**)**

**{**

returnEmp.ID==102**;**

**}**

publicclassEmployee

**{**

publicintID**{**get**;**set**; }**

publicstringName**{**get**;**set**; }**

**}**

**}**

In this example, **Find**() method expects a delegate to be passed as the argument. If you want to look at the signature of the delegate, right click on **Find()**method and select **"Go To Definition"**from the context menu. At this point you should see the following method.  
publicTFind**(**Predicate<T>match**);**  
  
Right click on Predicate<T> and select  **"Go To Definition"**  
  
Now you should see the signature of the Predicate delegate.  
publicdelegateboolPredicate<in**T**>**(T**obj**);**  
  
Notice that the delegate returns bool and expects an object of Type <T>. In our case T is Employee.   
  
So, to the **Find**() method we need to pass an instance of Predicate<Employee> delegate as an argument. Delegates are function pointers. This means when we create an instance of a delegate, we pass the name of the method as an argument to the delegate constructor.   
  
**Step 1:**Create a method whose signature matches with the signature of Predicate<Employee> delegate

privatestaticboolFindEmployee**(**EmployeeEmp**)**

**{**

returnEmp.ID==102**;**

**}**

**Step 2:** Create an instance of Predicate<Employee> delegate and pass the method name as an argument to the delegate constructor

Predicate<Employee>predicateEmployee=

newPredicate<Employee>**(**FindEmployee**);**

**Step 3:** Now pass the delegate instance as the argument for Find() method

Employeeemployee=

listEmployees.Find**(**x=>predicateEmployee**(**x**));**

**Anonymous methods were introduced in C# 2** and they eliminate the need for Step 1, 2 & 3, that is they provide us a way of creating delegate instances without having to write a separate method.  
  
**Now let us see, how to pass anonymous method as an argument to Find() method.**  
employee=listEmployees.Find**(**delegate**(**Employeex**) {**returnx.ID==102**; });**  
  
**Subscribing for an event handler is another example**

privatevoidForm1\_Load**(**objectsender**,**EventArgse**)**

**{**

ButtonButton1=newButton**();**

Button1.Text="Click Me"**;**

Button1.Click+=newEventHandler**(**Button1\_Click**);**

this.Controls.Add**(**Button1**);**

**}**

voidButton1\_Click**(**objectsender**,**EventArgse**)**

**{**

MessageBox.Show**(**"Button Clicked"**);**

**}**

**The code above can be replaced with the following code**

privatevoidForm1\_Load**(**objectsender**,**EventArgse**)**

**{**

ButtonButton1=newButton**();**

Button1.Text="Click Me"**;**

Button1.Click+=delegate**(**objectobj**,**EventArgseventArgs**)**

**{**

MessageBox.Show**(**"Button Clicked"**);**

**};**

this.Controls.Add**(**Button1**);**

**}**

**With anonymous Methods delegate parameters are optional.** This means the below code

Button1.Click+=delegate**(**objectobj**,**EventArgseventArgs**)**

**{**

MessageBox.Show**(**"Button Clicked"**);**

**};**

**can be rewritten as shown below**

Button1.Click+=delegate

**{**

MessageBox.Show**(**"Button Clicked"**);**

**};**

### Lambda expression in c#

**Anonymous methods and Lambda expressions are very similar.** Anonymous methods were introduced in C# 2 and Lambda expressions in C# 3.   
  
**To find an employee with Id = 102, using anonymous method**

Employeeemployee=listEmployees.Find

**(**delegate**(**EmployeeEmp**) {**returnEmp.ID==102**; });**

**To find an employee with Id = 102, using lambda expression**  
Employeeemployee=listEmployees.Find**(**Emp=>Emp.ID==102**);**  
  
**You can also explicitly specify the Input type but not required**  
employee=listEmployees.Find**((**EmployeeEmp**)**=>Emp.ID==102**);**  
  
**Notice that with a Lambda expression you don't have to use the delegate keyword explicitly and you don't have to specify the input parameter type explicitly.** The parameter type is inferred. Lambda expressions are more convenient to use than anonymous methods. Lambda expressions are particularly helpful for writing LINQ query expressions.  
  
**=>** is called lambda operator and read as GOES TO.  
  
**In most of the cases Lambda expressions supersedes anonymous methods.** To my knowledge, the only time I prefer to use anonymous methods over lambdas is, when we have to omit the parameter list when it's not used within the body.   
  
Anonymous methods allow the parameter list to be omitted entirely when it's not used within the body, where as with lambda expressions this is not the case.  
  
For example, with anonymous method notice that we have omitted the parameter list as we are not using them within the body

Button1.Click+=delegate

**{**

MessageBox.Show**(**"Button Clicked"**);**

**};**

The above code can be **rewritten using lambda expression**as shown below. Notice that with lambda we cannot omit the parameter list.

Button1.Click+=**(**eventSender**,**eventAgrs**)**=>

**{**

MessageBox.Show**(**"Button Clicked"**);**

**};**