**Delegates in C#**

**Delegates in C# with Examples**

In this article, I am going to discuss the **Delegates in C#** with Examples. Please read our previous article where we discussed [**Exception Handling**](https://dotnettutorials.net/lesson/exception-handling-csharp/) in Details. As part of this article, we are going to discuss the following important pointers in detail.

1. **What are delegates in C#?**
2. **How many ways we can call a method in C#?**
3. **How to invoke methods using delegates in C#?**
4. **Examples of using Delegates.**
5. **Rules of using Delegates in C#.**
6. **What are the types of delegates?**

**What are delegates in C#?**

In simple words, we can say that the delegates in C# are the **Type-Safe Function Pointer.**It means they hold the reference of a method or function and then calls that method for execution.

**How many ways we can call a method in C#?**

In C#, we can call a method that is defined in a class in two ways. They are as follows:

1. We can call the method using the object of the class if it is a non-static method or we can call the method through class name if it is a static method.
2. We can also call a method in C# by using delegates. Calling a C# method using delegate will be faster in execution as compared to the first process i.e. either by using an object or by using the class name.

**Invoking Method using Object and Class Name:**

In the below example, we are invoking the method by using the object of the class for the non-static method and using the class name for the static method.

**namespace** *DelegateDemo*

**{**

**public** **class** Program

**{**

//NonStatic method

**public** **void** Add**(int** x, **int** y**)**

**{**

Console.WriteLine**(**@"The Sum of {0} and {1}, is {2} ", x, y, **(**x + y**))**;

**}**

//Static Method

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

//calling non static method through object

obj.Add**(**100, 100**)**;

//Calling static method with class name

string GreetingsMessage = Program.Greetings**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Delegates in C#

In the above example, we are calling the methods using the object and the class name. Now let’s see how to call a method using delegates in C#.

**How to invoke methods using delegates in C#?**

If you want to invoke or call a method using delegates then you need to follow three simple steps.  The steps are as follows.

1. **Defining a delegate**
2. **Instantiating a delegate**
3. **Invoking a delegate**

**Step1: Define a Delegate in C#**

The syntax to declare a delegate in C# is very much similar to the function declaration. In delegate, we need to use the keyword delegate. The syntax for defining a delegate:  
**<Access Modifier> delegate <return type> <delegate name> (arguments list);**  
Example: If you have a method like below.

**public** **void** Add**(int** x, **int** y**)**

**{**

Console.WriteLine**(**@"The Sum of {0} and {1}, is {2} ", x, y, **(**x + y**))**;

**}**

Then you have to define a delegate like below.  
**public delegate void AddDelegate(int a, int b);**

**Example:**If you have a method like below

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name;

**}**

Then you need to define a delegate like below.  
**public delegate string GreetingsDelegate(string name);**

**Note:** The point that you need to remember while working with C# Delegates is, the signature of the delegate and the method it points should be the same. So, when you create a delegate, then the **access modifier**, **return type**, **number of arguments,** and their **data types** of the delegates must and should be the same as the **access modifier, return type, number of arguments,** and **the data types** of the function that the delegate wants to refer. You can define the delegates either within a class or just like other types we defined under a namespace.

**Step2: Instantiating the Delegate in C#.**

Once we declare the delegate, then we need to consume the delegate. To consume the delegate, first, we need to create an object of the delegate and while creating the object the method you want to execute using the delegate should be passed as a parameter. The syntax to create an instance of a delegate is given below.  
**DelegateName ObjectName = new DelegateName (target function-name);**

**Example:**  
**AddDelegate ad = new AddDelegate(obj.Add);**  
**GreetingsDelegate gd = new GreetingsDelegate(Program.Greetings);**

While binding the method with delegate, if the method is non-static refer to it as the object of the class and if it is static refer to it with the name of the class.

**Step3: Invoking the Delegate in C#.**

Once we create an instance of a delegate, then we need to call the delegate by providing the required values to the parameters so that the methods get executed internally which is bound with the delegates. For example:  
**ad(100, 50);**  
**ad.Invoke(200, 300);**  
**string GreetingsMessage = gd(“Priyanka”);**  
**string GreetingsMessage = gd.Invoke(“Anurag”);**  
At this point, the function that is referred to by the delegate will be called for execution.

**Complete Example for a Better Understanding.**

Please have a look at the below code. In the below example, first, we declare two delegates. Then within the program class, we define two methods whose signature is the same as the delegate signature. Then we create instances of the delegates and while creating the instance we are providing the function name as a parameter to the delegate constructor. It is this function that will execute when we invoke the delegate. Once we create the instance then we call the delegate by providing the required parameter. We can also invoke a delegate by using the Invoke method. The following code is self-explained, please go through the comment lines.

**namespace** *DelegateDemo*

**{**

//Defining Delegates

//Note: The access specifeis, return type and the number, order and type of parameters of delegate

//should be same as the function it refers to.

**public** **delegate** **void** AddDelegate**(int** a, **int** b**)**;

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**public** **class** Program

**{**

//Defining Methods

//NonStatic method

**public** **void** Add**(int** x, **int** y**)**

**{**

Console.WriteLine**(**@"The Sum of {0} and {1}, is {2} ", x, y, **(**x + y**))**;

**}**

//Static Method

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

//Instantiating delegate by passing the target function Name

//The Add method is non static so here we are calling method using object

AddDelegate ad = new AddDelegate**(**obj.Add**)**;

//Greetings method is static so here we are callling the method by using the class name

GreetingsDelegate gd = new GreetingsDelegate**(**Program.Greetings**)**;

//Invoking The Delegates

ad**(**100, 50**)**;

string GreetingsMessage = gd**(**"Pranaya"**)**;

//We can also use Invoke method to execute delegates

// ad.Invoke(100, 50);

// string GreetingsMessage = gd.Invoke("Pranaya");

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

When you run the application, it will give you the following output.

Delegates in C#

**Rules of using Delegates in C#:**

1. A delegate in C# is a user-defined type and hence before invoking a method using a delegate, we must have to define that delegate first.
2. The **signature**of the delegate **must match** the signature of the **method**, the delegate points to otherwise we will get a compiler error. This is the reason why delegates are called type-safe function pointers.
3. A Delegate is similar to a class. This means we can create an instance of a delegate and when we do so, we need to pass the method name as a parameter to the delegate constructor, and it is the function the delegate will point to
4. **Tip to remember delegate syntax:**Delegates syntax looks very much similar to a method with a delegate keyword.

**What are the Types of Delegates in C#?**

The Delegates in C# are classified into two types such as

1. **Single Cast Delegate**
2. **Multicast Delegate**

If a delegate is used for invoking a single method then it is called a single cast delegate or unicast delegate. In other words, we can say that the delegates that represent only a single function are known as single cast delegates.

If a delegate is used for invoking multiple methods then it is known as the multicast delegate. Or the delegates that represent more than one function are called Multicast delegates.

The example that we discussed in this article is of type Single Cast Delegate because the delegate points to a single function. In the next article, I am going to discuss the [**Multicast Delegate in C#**](https://dotnettutorials.net/lesson/multicast-delegate-csharp/) with examples. Here, in this article, I try to explain **Delegates in C#**with examples. I hope you understood the need and use of delegates in C#.

**Multicast Delegates in C#**

**Multicast Delegates in C# with Examples**

In this article, I am going to discuss the **Multicast Delegates in C#** with Examples. Please read our previous article where we discussed [**Single Cast Delegates in C#**](https://dotnettutorials.net/lesson/delegates-csharp/)with examples. As part of this article, we are going to discuss the following pointers in detail.

1. **What is Multicast Delegate in C#?**
2. **How to create Multicast Delegates in C#?**
3. **Different ways to create Multicast Delegates.**
4. **Multicast Delegate with Return Type in C#.**
5. **Multicast Delegate with Output Parameter in C#.**

**What is Multicast Delegate in C#?**

A Multicast Delegate in C# is a delegate that holds the references of more than one function. When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked. If you want to call multiple methods using a delegate then all the method signatures should be the same.

**Example: Multicast Delegate in C#**

Let us see an example for understanding the Multicast Delegate in C#. Please have a look at the following example which is without using delegates. In the below example, we created two methods, and then from the main method, we are creating the instance of the class and then invoke the two methods.

**namespace** *MulticastDelegateDemo*

**{**

**public** **class** Rectangle

**{**

**public** **void** GetArea**(double** Width, **double** Height**)**

**{**

Console.WriteLine**(**@"Area is {0}", **(**Width \* Height**))**;

**}**

**public** **void** GetPerimeter**(double** Width, **double** Height**)**

**{**

Console.WriteLine**(**@"Perimeter is {0}", **(**2 \* **(**Width + Height**)))**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Rectangle rect = new Rectangle**()**;

rect.GetArea**(**23.45, 67.89**)**;

rect.GetPerimeter**(**23.45, 67.89**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

What is Multicast Delegate in C#?

In the above example, we created an instance of the **Rectangle** class and then called the two methods. Now I want to create a single delegate that is going to invoke the above two methods (i.e. **GetArea** and **GetPerimeter**). The two methods having the same signature with the different method names, so we can create a single delegate that holds the reference of the above two methods. And when we invoke the delegate, it is going to invoke the above two methods. And when we do so, then it is called a **Multicast Delegate**.

**Example: Multicast Delegate in C#.**

In the below example, we have created one delegate whose signature is the same as the two methods i.e. GetArea and GetPerimeter. Then we created the instance of delegate and bind the two methods using the += operator. Similarly, you can use the -= operator to remove a function from the delegate. Once we bind the two methods with the delegate instance and then when we call the delegate, both methods are going to be executed.

**namespace** *MulticastDelegateDemo*

**{**

**public** **delegate** **void** RectangleDelete**(double** Width, **double** Height**)**;

**public** **class** Rectangle

**{**

**public** **void** GetArea**(double** Width, **double** Height**)**

**{**

Console.WriteLine**(**@"Area is {0}", **(**Width \* Height**))**;

**}**

**public** **void** GetPerimeter**(double** Width, **double** Height**)**

**{**

Console.WriteLine**(**@"Perimeter is {0}", **(**2 \* **(**Width + Height**)))**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Rectangle rect = new Rectangle**()**;

RectangleDelete rectDelegate = new RectangleDelete**(**rect.GetArea**)**;

//RectangleDelete rectDelegate = rect.GetArea;

//binding a method with delegate object

// In this example rectDelegate is a multicast delegate. You use += operator

// to chain delegates together and -= operator to remove.

rectDelegate += rect.GetPerimeter;

rectDelegate**(**23.45, 67.89**)**;

Console.WriteLine**()**;

rectDelegate.Invoke**(**13.45, 76.89**)**;

Console.WriteLine**()**;

//Removing a method from delegate object

rectDelegate -= rect.GetPerimeter;

rectDelegate.Invoke**(**13.45, 76.89**)**;

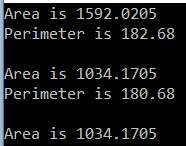
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Another approach to Create Multicast Delegates in C#.**

Here in the below example, I am going to show you the use of both static and non-static methods along with different ways to create and invoke multicast delegates. Please have a look at the below example. Here, we created one delegate which takes two integer parameters and returns nothing. Then within the program class, we define four methods and all these four methods take two integer parameters and return nothing i.e. void. Then we created four instances of the delegate and bind the four methods. Finally, we create the fifth delegate instance and to this instance, we are binding all the four delegate instances using the + operator. Now, the fifth delegate becomes a multicast delegate. And when we invoke the fifth delegate instance then all the four methods are going to be executed. If you want to remove one method binding, then simply you need to use the -= operator and specify the delegate instance which you want to remove.

**namespace** *MulticastDelegateDemo*

**{**

**public** **delegate** **void** MathDelegate**(int** No1, **int** No2**)**;

**public** **class** Program

**{**

**public** **static** **void** Add**(int** x, **int** y**)**

**{**

Console.WriteLine**(**"THE SUM IS : " + **(**x + y**))**;

**}**

**public** **static** **void** Sub**(int** x, **int** y**)**

**{**

Console.WriteLine**(**"THE SUB IS : " + **(**x - y**))**;

**}**

**public** **void** Mul**(int** x, **int** y**)**

**{**

Console.WriteLine**(**"THE MUL IS : " + **(**x \* y**))**;

**}**

**public** **void** Div**(int** x, **int** y**)**

**{**

Console.WriteLine**(**"THE DIV IS : " + **(**x / y**))**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Program p = new Program**()**;

MathDelegate del1 = new MathDelegate**(**Add**)**;

MathDelegate del2 = new MathDelegate**(**Program.Sub**)**;

MathDelegate del3 = p.Mul;

MathDelegate del4 = new MathDelegate**(**p.Div**)**; ;

//In this example del5 is a multicast delegate. We can use +(plus)

// operator to chain delegates together and -(minus) operator to remove.

MathDelegate del5 = del1 + del2 + del3 + del4;

del5.Invoke**(**20, 5**)**;

Console.WriteLine**()**;

del5 -= del2;

del5**(**22, 7**)**;

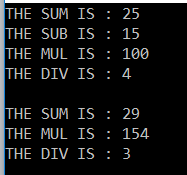
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Multicast Delegates with Return Type in C#:**

A multicast delegate invokes the methods in the same order in which they are added. If the delegate has a return type other than void and if the delegate is a multicast delegate, then only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last invoked method.

**Example: Multicast Delegates with Return Type in C#**

Let’s understand Multicast Delegates with Return Type in C# with an example. Please have a look at the below example. Here, we created one delegate which does not take any parameter but its return type is int. Then we created two static methods and the first static method returns 1 and the second static method returns 2. Then we created the delegate instance and first bind Method one and second bind method two. And when we invoke the delegate, the first MethodOne is executed and then MethodOne is executed and it returns 2 as the last invoked method is MethodTwo which returns 2.

**namespace** *MulticastDelegateDemo*

**{**

// Deletegate's return type is int

**public** **delegate** **int** SampleDelegate**()**;

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

SampleDelegate del = new SampleDelegate**(**MethodOne**)**;

del += MethodTwo;

// The ValueReturnedByDelegate will be 2, returned by the MethodTwo(),

// as it is the last method in the invocation list.

**int** ValueReturnedByDelegate = del**()**;

Console.WriteLine**(**"Returned Value = {0}", ValueReturnedByDelegate**)**;

Console.ReadKey**()**;

**}**

// This method returns one

**public** **static** **int** MethodOne**()**

**{**

**return** 1;

**}**

// This method returns two

**public** **static** **int** MethodTwo**()**

**{**

**return** 2;

**}**

**}**

**}**

**Output:**

Multicast Delegate with Return Type in C#.

**Multicast Delegates with out parameter in C#.**

Now we will see an example of Multicast delegate in C# with out parameter. Please have a look at the below example. Here, we created one delegate which takes one out parameter and returns nothing i.e. void. Then we created two static methods and both the static methods take one out parameter. The first static method assigning 1 value to the out parameter and the second static assigns value 2 to the out parameter. Then we created the delegate instance and first bind MethodOne and second bind MethodTwo. And when we invoke the delegate, first MethodOne is executed and then MethodOne is executed and it returns 2 as the last invoked method is MethodTwo which assigns value 2 to the out parameter.

**namespace** *MulticastDelegateDemo*

**{**

// Deletegate has an int output parameter

**public** **delegate** **void** SampleDelegate**(**out **int** Integer**)**;

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

SampleDelegate del = new SampleDelegate**(**MethodOne**)**;

del += MethodTwo;

// The ValueFromOutPutParameter will be 2, initialized by MethodTwo(),

// as it is the last method in the invocation list.

**int** ValueFromOutPutParameter = -1;

del**(**out ValueFromOutPutParameter**)**;

Console.WriteLine**(**"Returned Value = {0}", ValueFromOutPutParameter**)**;

Console.ReadKey**()**;

**}**

// This method sets ouput parameter Number to 1

**public** **static** **void** MethodOne**(**out **int** Number**)**

**{**

Number = 1;

**}**

// This method sets ouput parameter Number to 2

**public** **static** **void** MethodTwo**(**out **int** Number**)**

**{**

Number = 2;

**}**

**}**

**}**

**Output:**

Multicast Delegate with Output Parameter in C#.

**Delegates Real-time Example in C#**

**Delegates Real-time Example in C#**

In this article, I am going to discuss the **Delegate’s Real-time Example in C#**. The delegates are one of the most important concepts that you need to understand as a C# developer. In many interviews, most of the interviewers ask to explain the usage of delegates in the real-time project that you have worked on.  Please read the following two articles before proceeding to this article, where we discussed the basics of delegate and multicast delegates in C# with examples.

1. [**Single Cast Delegates in C#**](https://dotnettutorials.net/lesson/delegates-csharp/) – Here we discussed the basics of Delegates in C# with examples.
2. [**Multicast Delegate in C#**](https://dotnettutorials.net/lesson/multicast-delegate-csharp/)– Here we discussed the Multicast Delegates in C# with examples.

The Delegates in C# are extensively used by framework developers.  Let us understand delegates in C# with one real-time example. Let say we have a class called Employee as shown below.

**Employee.cs**

**namespace** *DelegateRealtimeExample*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Experience **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

The above **Employee** class has the following properties.

1. **Id**
2. **Name**
3. **Gender**
4. **Experience**
5. **Salary**

Now I want to write a method in the Employee class which can be used to promote the employees. The method that we are going to write will take a list of Employee objects as a parameter and then should print the names of all the employees who are eligible for a promotion.

But the logic based on which the employee gets promoted should not be hardcoded. At times we may promote employees based on their experience and at times we may promote them based on their salary or maybe some other condition. So, the logic to promote employees should not be hard-coded within the method.

**To achieve this we can make use of delegates. So now I would design my class as shown below.**

**namespace** *DelegateRealtimeExample*

**{**

**public** **delegate** **bool** EligibleToPromotion**(**Employee EmployeeToPromotion**)**;

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Experience **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** **static** **void** PromoteEmployee**(**List**<**Employee**>** lstEmployees, EligibleToPromotion IsEmployeeEligible**)**

**{**

**foreach** **(**Employee employee in lstEmployees**)**

**{**

**if** **(**IsEmployeeEligible**(**employee**))**

**{**

Console.WriteLine**(**"Employee {0} Promoted", employee.Name**)**;

**}**

**}**

**}**

**}**

**}**

In the above example, we created a delegate called **EligibleToPromotion**. This delegate takes the Employee object as a parameter and returns a boolean. In the Employee class, we have a **PromoteEmpoloyee** method. This method takes a list of Employees and a Delegate of the type EligibleToPromotion as parameters.

The method then loops thru each employee’s object and passes it to the delegate. If the delegate returns true, then the Employee is promoted, else not promoted. So within the method, we have not hardcoded any logic on how we want to promote employees.

Now the client who uses the Employee class has the flexibility of determining the logic on how they want to promote their employees. First create the employee objects – emp1, emp2, and emp3. Populate the properties for the respective objects. Then create an employee List to hold all the 3 employees as shown below.

**namespace** *DelegateRealtimeExample*

**{**

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Experience = 5,

Salary = 10000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Experience = 10,

Salary = 20000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Experience = 15,

Salary = 30000

**}**;

List**<**Employee**>** lstEmployess = new List**<**Employee**>()**;

lstEmployess.Add**(**emp1**)**;

lstEmployess.Add**(**emp2**)**;

lstEmployess.Add**(**emp3**)**;

EligibleToPromotion eligibleTopromote = new EligibleToPromotion**(**Program.Promote**)**;

Employee.PromoteEmployee**(**lstEmployess, eligibleTopromote**)**;

Console.ReadKey**()**;

**}**

**public** **static** **bool** Promote**(**Employee employee**)**

**{**

**if** **(**employee.Salary **>** 10000**)**

**{**

**return** **true**;

**}**

**else**

**{**

**return** **false**;

**}**

**}**

**}**

**}**

Notice the Promote method that we have created. This method has the logic of how we want to promote our employees. The method is then passed as a parameter to the delegate. Also, note this method has the same signature as that of the EligibleToPromotion delegate. This is very important because the Promote method cannot be passed as a parameter to the delegate if the signature differs. This is the reason why delegates are called type-safe function pointers.

**Output:**

Delegates Real-time example in C#

So if we did not have the concept of delegates it would not have been possible to pass a function as a parameter. As Promote method in the Employee class makes use of delegate, it is possible to dynamically decide the logic on how we want to promote employees.

In C# 3.0 Lambda expressions are introduced. So you can make use of lambda expressions instead of creating a function and then an instance of a delegate and then passing the function as a parameter to the delegate. The sample example rewritten using the Lambda expression is shown below. The Private Promote method is no longer required now.

**namespace** *DelegateRealtimeExample*

**{**

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Experience = 5,

Salary = 10000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Experience = 10,

Salary = 20000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Experience = 15,

Salary = 30000

**}**;

List**<**Employee**>** lstEmployess = new List**<**Employee**>()**;

lstEmployess.Add**(**emp1**)**;

lstEmployess.Add**(**emp2**)**;

lstEmployess.Add**(**emp3**)**;

Employee.PromoteEmployee**(**lstEmployess, x =**>** x.Experience **>** 5**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Delegates Real-time example in C#

**Anonymous Method in C#**

**Anonymous Method in C# with examples**

In this article, I am going to discuss the **Anonymous Method in C#** with Examples. Please read our previous article where we discussed one [**real-time example of the delegate**](https://dotnettutorials.net/lesson/delegates-real-time-example-csharp/). As part of this article, we are going to discuss the following concepts of Anonymous methods in detail.

1. **What is Anonymous Method in C#?**
2. **Why do we need Anonymous Methods in C#?**
3. **Examples of the Anonymous Method.**
4. **What are the Advantages of Using Anonymous Methods in C#?**
5. **Examples of Anonymous methods accessing variables defined in an outer function.**
6. **What are the Limitations of Anonymous Methods in C#?**
7. **Anonymous Methods Real-time Examples in C#**

**What is Anonymous Method in C#?**

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

**Why do we need Anonymous Methods in C#?**

In our [**Delegates in C#**](https://dotnettutorials.net/lesson/delegates-csharp/) article, we discussed how to bind a delegate with a method. To bind a delegate with a method, first, we need to create an instance of a delegate and when we create the instance of a delegate, we need to pass the method name as a parameter to the delegate constructor, and it is the function the delegate will point to.

**Below is an example to understand Delegate in C#.**

**namespace** *DelegateDemo*

**{**

**public** **class** AnonymousMethods

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name + " Welcome to Dotnet Tutorials";

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

GreetingsDelegate gd = new GreetingsDelegate**(**AnonymousMethods.Greetings**)**;

string GreetingsMessage = gd.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Anonymous Method in C#

In the above example,

1. We create one delegate (GreetingsDelegate)
2. Then we instantiate the delegate, while we are instantiating the delegate, we are passing the Method name as a parameter to the constructor of the delegate
3. Finally, we invoke the delegate

As of now, this is the approach we are following to bind a method to a delegate and execute. An anonymous method is also related to a delegate. Without binding a named block (function) to a delegate, we can also bind a code block to a delegate means an unnamed code blocked to a delegate which is nothing but an anonymous method in C#.

**Let’s see an example for a better understanding of the Anonymous Methods in C#.**

**namespace** *DelegateDemo*

**{**

**public** **class** AnonymousMethods

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

GreetingsDelegate gd = **delegate(**string name**)**

**{**

**return** "Hello @" + name + " Welcome to Dotnet Tutorials";

**}**;

string GreetingsMessage = gd.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

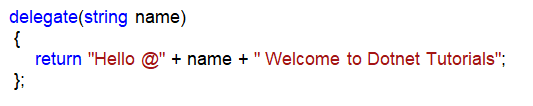
**}**

**}**

**Output:**

Anonymous Method in C#

In the above example, the following code is an example of an Anonymous method.



The above code is without a name and it contains only the body and the method is defined using the delegate keyword. We don’t require writing any access modifiers like public, private, and protected, etc. We also don’t require writing any return type like a void, int, double, etc.

**What are the Advantages of Using Anonymous Methods in C#?**

Lesser typing word. Generally, anonymous methods are suggested when the code volume is very less.

**Examples of Anonymous methods accessing variables defined in an outer function.**

Let’s see an example for a better understanding.

**namespace** *DelegateDemo*

**{**

**public** **class** AnonymousMethods

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

string Message = "Welcome to Dotnet Tutorials";

GreetingsDelegate gd = **delegate(**string name**)**

**{**

**return** "Hello @" + name + " " + Message;

**}**;

string GreetingsMessage = gd.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Anonymous Method in C#

**What are the Limitations of Anonymous Methods in C#?**

1. An anonymous method in C# cannot contain any jump statement like goto, break or continue.
2. It cannot access the ref or out parameter of an outer method.
3. The Anonymous methods cannot have or access the unsafe code.

**Points to Remember while working with the Anonymous Methods in C#:**

1. The anonymous methods can be defined using the delegate keyword
2. An anonymous method must be assigned to a delegate type.
3. This method can access outer variables or functions.
4. An anonymous method can be passed as a parameter.
5. This method can be used as event handler.

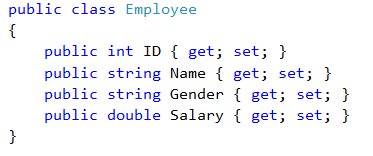
**Anonymous Method Real-Time Example in C#**

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

**Let’s understand how a method can exist without a name in C# with one example.**

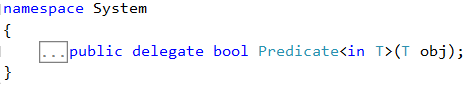
**Step1:**

Create a class Employee and copy and paste the below code

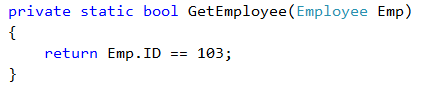


**Step2:**

In our [**Generic Delegates in C#**](https://dotnettutorials.net/lesson/generic-delegates-csharp/) article, we already discussed that Predicate is a generic delegate that accepts a single input parameter of any type and returns a Boolean value that is mandatory and fixed. Following is the signature of the Predicate Generic Delegate.



As shown in the image above, the Predicate Generic delegate takes an input parameter of type T (in T) and returns a Boolean value. Now, we need to create a method whose signature must match with the signature of Predicate<Employee> delegate



As shown in the above image, the GetEmployee method takes one input parameter of type Employee and returns a Boolean value. So the above method signature matches the signature of the Predicate generic delegate.

The method logic is very simple. It checks the ID value of the employee which is passed as a parameter to this function, if the ID value is 103, then it returns true else it returns false.

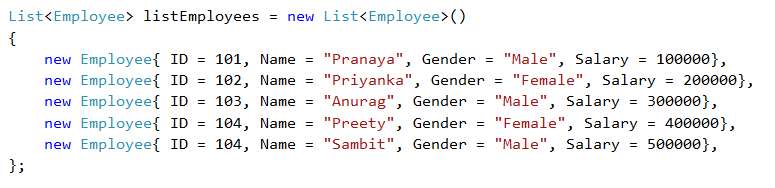
**Step3:**

In this step, we are going to create an instance of the Predicate Generic Delegate. While we are creating the instance, we need to pass the GetEmployee method as a parameter to the constructor of Predicate as shown in the image below.

Anonymous Method Real-Time Example in C#

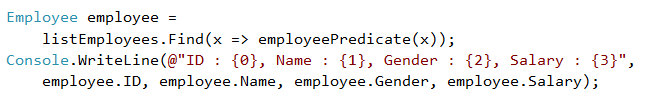
**Step4:**

Now we need to create a collection of Type Employee to hold a list of Employees as shown in the below image.



**Step5:**

In this step, we need to pass the delegate instance to the Find method of the List collection class as shown in the image below



**We are done. The complete code is shown below**

**namespace** *AnonymousMethodRealTimeExample*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

//Step4

//Create a collection of List of Employees

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

**{**

new Employee**{** ID = 101, Name = "Pranaya", Gender = "Male", Salary = 100000**}**,

new Employee**{** ID = 102, Name = "Priyanka", Gender = "Female", Salary = 200000**}**,

new Employee**{** ID = 103, Name = "Anurag", Gender = "Male", Salary = 300000**}**,

new Employee**{** ID = 104, Name = "Preety", Gender = "Female", Salary = 400000**}**,

new Employee**{** ID = 104, Name = "Sambit", Gender = "Male", Salary = 500000**}**,

**}**;

// Step 3:

// Create an instance of Predicate<Employee> delegate and pass

// the method name as an argument to the delegate constructor

Predicate**<**Employee**>** employeePredicate = new Predicate**<**Employee**>(**GetEmployee**)**;

// Step 5:

// Now pass the delegate instance as the

// argument to the Find() method of List collection

Employee employee =

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

Console.WriteLine**(**@"ID : {0}, Name : {1}, Gender : {2}, Salary : {3}",

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

Console.ReadKey**()**;

**}**

// Step 2:

// Create a method whose signature matches with the

// signature of Predicate<Employee> generic delegate

**private** **static** **bool** GetEmployee**(**Employee Emp**)**

**{**

**return** Emp.ID == 103;

**}**

**}**

// Step1

// Create a class called Employee with

// ID, Name, Gender and Salary Properties

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **double** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**

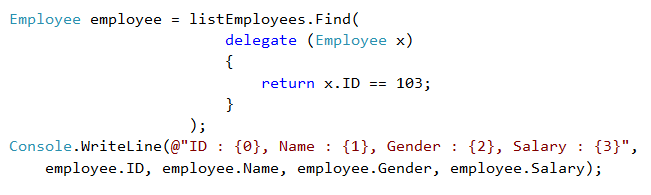
Anonymous Method Real-Time Example in C#

**Use the anonymous method.**

As of now what we did?

1. We create a method whose signature matches with the Predicate Generic Delegate
2. Then we create an instance of the Predicate Generic Delegate
3. Then we pass that Predicate Instance as an argument to the Find method of the List collection class

Using an anonymous method, we can safely avoid the above three steps. We can pass an anonymous method as an argument to the Find() method as shown in the image below.



**The complete code is given below.**

**namespace** *AnonymousMethodRealTimeExample*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

//Step2

//Create a collection of List of Employees

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

**{**

new Employee**{** ID = 101, Name = "Pranaya", Gender = "Male", Salary = 100000**}**,

new Employee**{** ID = 102, Name = "Priyanka", Gender = "Female", Salary = 200000**}**,

new Employee**{** ID = 103, Name = "Anurag", Gender = "Male", Salary = 300000**}**,

new Employee**{** ID = 104, Name = "Preety", Gender = "Female", Salary = 400000**}**,

new Employee**{** ID = 104, Name = "Sambit", Gender = "Male", Salary = 500000**}**,

**}**;

//Step3

// An anonymous method is being passed as an argument to

// the Find() method of List Collection.

Employee employee = listEmployees.Find**(**

**delegate** **(**Employee x**)**

**{**

**return** x.ID == 103;

**}**

**)**;

Console.WriteLine**(**@"ID : {0}, Name : {1}, Gender : {2}, Salary : {3}",

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

Console.ReadKey**()**;

**}**

**}**

// Step1

// Create a class called Employee with

// ID, Name, Gender and Salary Properties

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **double** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**

Anonymous Method Real-Time Example in C#

**Find Method:**

In the above two examples, the **Find**() method of the generic List collection class expects a delegate to be passed as an argument. If you want to look at the signature of the Find method, then right-click on the **Find()**method and select **“Go To Definition”**from the context menu. Then you will see the following method.

**public T Find(Predicate<T> match);**

**Lambda Expressions in C#**

**Lambda Expressions in C# with real-time Examples**

In this article, I am going to discuss the **Lambda Expressions in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Anonymous Methods in C#**](https://dotnettutorials.net/lesson/anonymous-method-c-sharp/) with examples. As part of this article, we are going to discuss the following pointers in detail.

1. **What are Lambda Expressions in C#?**
2. **Why do we need Lambda Expressions?**
3. **How to create Lambda Expression in C#?**
4. **Examples of using Lambda Expression.**

**What are Lambda Expressions in C#?**

The **Lambda Expression in C#** is the shorthand for writing the anonymous function. So we can say that the Lambda Expression in C# is nothing but to simplify the anonymous function in C#. Let’s understand this with an example.

**namespace** *LambdaExpressionDemo*

**{**

**public** **class** LambdaExpression

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

GreetingsDelegate obj = new GreetingsDelegate**(**LambdaExpression.Greetings**)**;

string GreetingsMessage = obj.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name + " welcome to Dotnet Tutorials";

**}**

**}**

**}**

**OUTPUT:**

What are Lambda Expressions in C#?

In the above example, we use a [**delegate**](https://dotnettutorials.net/lesson/delegates-csharp/)to bind a function. Let’s convert the above example to use an [**anonymous**](https://dotnettutorials.net/lesson/anonymous-method-c-sharp/)function.

**namespace** *LambdaExpressionDemo*

**{**

**public** **class** LambdaExpression

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

GreetingsDelegate obj = **delegate** **(**string name**)**

**{**

**return** "Hello @" + name + " welcome to Dotnet Tutorials";

**}**;

string GreetingsMessage = obj.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**}**

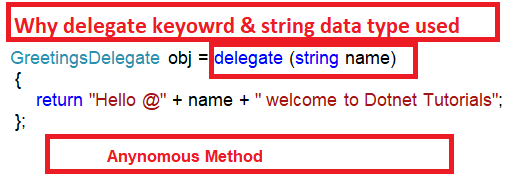
**}**

**OUTPUT:**

Why do we need Lambda Expressions?

**Questions: Two things come to our minds.**

As the Anonymous method is used for lesser writing, then why we used the delegate keyword. As the delegate knows about the return type and parameter type it accepts, then why we explicitly specifying the parameter type the delegate accepts.



We can overcome this by using Lambda Expressions which is introduced in C#3.

**How to create Lambda Expression in C#?**

To create a lambda expression in C#, we need to specify the input parameters (if any) on the left side of the lambda operator **=>**, and we need to put the expression or statement block on the other side.

**Let us rewrite the same example by using Lambda Expression in C#.**

**namespace** *LambdaExpressionDemo*

**{**

**public** **class** LambdaExpression

**{**

**public** **delegate** string GreetingsDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

GreetingsDelegate obj = **(**name**)** =**>**

**{**

**return** "Hello @" + name + " welcome to Dotnet Tutorials";

**}**;

string GreetingsMessage = obj.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**GreetingsMessage**)**;

Console.ReadKey**()**;

**}**

**public** **static** string Greetings**(**string name**)**

**{**

**return** "Hello @" + name + " welcome to Dotnet Tutorials";

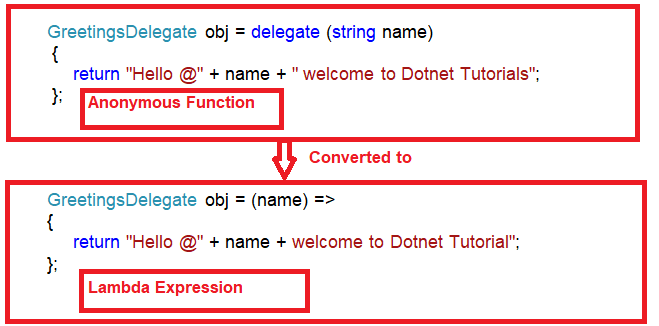
**}**

**}**

**}**

**OUTPUT:**

Examples of using Lambda Expression.



**Generic Delegates in C#**

**Generic Delegates in C# with Real-Time Examples**

In this article, I am going to discuss the **Generic Delegates in C#** with examples. Please read the [**Delegates in C#**](https://dotnettutorials.net/lesson/delegates-csharp/) article before proceeding to this article. As part of this article, we are going to discuss the following pointers in detail.

1. **What are Generic Delegates in C#?**
2. **Types of Generic Delegates in C#.**
3. **Why do we need the Generic Delegates?**
4. **Func Generic Delegate in C#?**
5. **Action Generic Delegate in C#?**
6. **Predicate Generic Delegate in C#?**
7. **Examples to understand Generic Delegates in C#.**

**What are Generic Delegates in C#?**

The Generic Delegates in C# were introduced as part of **.NET Framework 3.5** which doesn’t require defining the delegate instance in order to invoke the methods. To understand the Generic Delegates in C# you should have the basic knowledge of [**Delegates**](https://dotnettutorials.net/lesson/delegates-csharp/).

**Types of Generic Delegates in C#**

C# provides three built-in generic delegates, they are

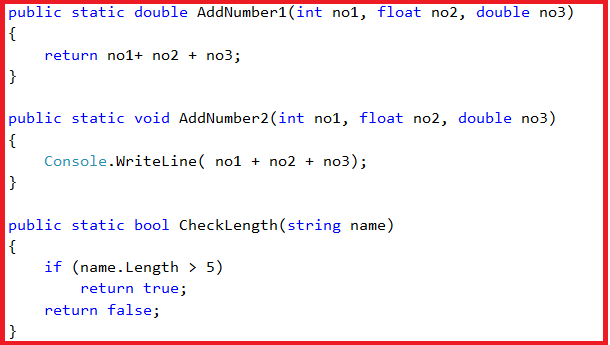
1. **Func**
2. **Action**
3. **Predicate**

**Why do we need the Generic Delegates in C#?**

Let us understand the need for Generic Delegates with an example. In order to understand this, let us first understand how we use delegates to invoke methods.

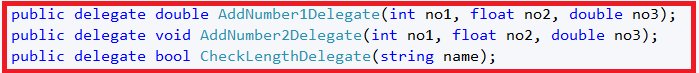
**Example:**

Let us say we have the following three methods and we want to invoke these methods using delegates.

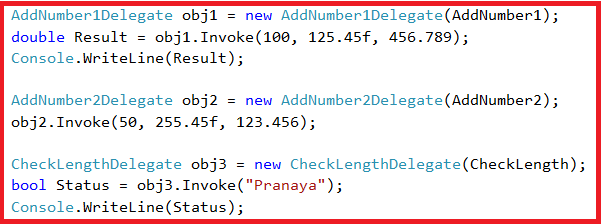


As you can see the **AddNumber1** method takes three parameters and returns a value of double type. Similarly, the **AddNumber2** method takes three parameters but it does not return any value. The third method i.e. the **CheckLength** method takes one string parameter and returns a Boolean value.

If we want to invoke the above three methods using delegates then we need to create three delegates whose signature should match with the above three methods as shown in the below image.



As you can see in the above image, we create three delegates. Let’s invoke the methods using the respective delegate instance as shown in the below image.



**Following is the complete example code.**

**namespace** *GenericDelegateDemo*

**{**

**public** **class** GenericDelegates

**{**

**public** **delegate** **double** AddNumber1Delegate**(int** no1, **float** no2, **double** no3**)**;

**public** **delegate** **void** AddNumber2Delegate**(int** no1, **float** no2, **double** no3**)**;

**public** **delegate** **bool** CheckLengthDelegate**(**string name**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

AddNumber1Delegate obj1 = new AddNumber1Delegate**(**AddNumber1**)**;

**double** Result = obj1.Invoke**(**100, 125.45f, 456.789**)**;

Console.WriteLine**(**Result**)**;

AddNumber2Delegate obj2 = new AddNumber2Delegate**(**AddNumber2**)**;

obj2.Invoke**(**50, 255.45f, 123.456**)**;

CheckLengthDelegate obj3 = new CheckLengthDelegate**(**CheckLength**)**;

**bool** Status = obj3.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**Status**)**;

Console.ReadKey**()**;

**}**

**public** **static** **double** AddNumber1**(int** no1, **float** no2, **double** no3**)**

**{**

**return** no1+ no2 + no3;

**}**

**public** **static** **void** AddNumber2**(int** no1, **float** no2, **double** no3**)**

**{**

Console.WriteLine**(** no1 + no2 + no3**)**;

**}**

**public** **static** **bool** CheckLength**(**string name**)**

**{**

**if** **(**name.Length **>** 5**)**

**return** **true**;

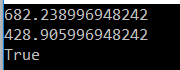
**return** **false**;

**}**

**}**

**}**

**Output:**



As of now, this is the way, we use delegates to invoke methods. The question that comes to our mind is

**Do we really need to create the Delegates?**

The answer is no. C# provides some generic delegates who can do the job for us. C# provides three Generic Delegates, they are as follows

1. **Func**
2. **Action**
3. **Predicate**

**What is Func Generic Delegate in C#?**

The **Func Generic Delegate in C#** is present in the **System** namespace. This delegate takes one or more input parameters and returns one out parameter. The last parameter is considered as the return value. The Func Generic Delegate in C# can take up to 16 input parameters of different types. It must have one return type. The return type is mandatory but the input parameter is not.

**Note:** Whenever your delegate returns some value, whether by taking any input parameter or not, you need to use the Func Generic delegate in C#.

**What is Action Generic Delegate in C#?**

The **Action Generic Delegate in C#** is also present in the **System** namespace. It takes one or more input parameters and returns nothing. This delegate can take a maximum of **16 input parameters** of the different or same type

**Note:** Whenever your delegate does not return any value, whether by taking any input parameter or not, then you need to use the Action  Generic delegate in C#.

**What is Predicate Generic Delegate in C#?**

The **Predicate Generic Delegate in C#** is also present in the **System** namespace. This delegate is used to verify certain criteria of the method and returns the output as Boolean, either True or False. It takes one input parameter and always returns a Boolean value which is mandatory. This delegate can take a maximum of **1 input parameter** and always return the value of the Boolean type.

**Note:** Whenever your delegate returns a Boolean value, by taking one input parameter, then you need to use the Predicate Generic delegate in C#.

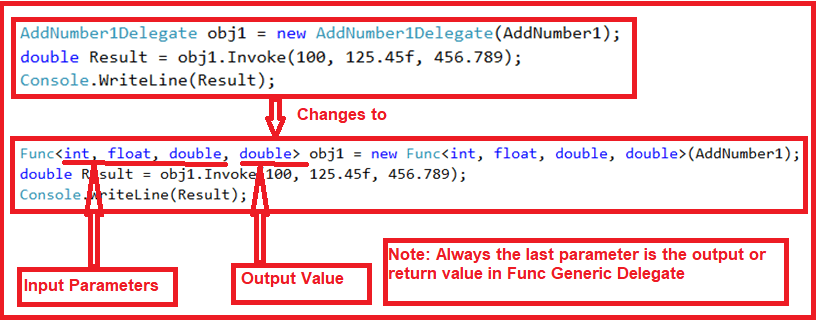
**Examples to understand Generic Delegates in C#.**

Let’s understand the above three generic delegates in C# with an example. In our first example, we created three methods,

1. The **addnumber1** method takes three parameters and returns a **double** value. Here we will use the **Func Generic Delegate** to achieve the same thing as we achieve in the first example.
2. Similarly, the**addnumber2** method takes three parameters but does not return any value. Here we will use the **Action Generic Delegate** to achieve the same thing as we achieve in the first example.
3. The **checklength** method takes one string parameter and returns a boolean value. Here we will use the **Predicate Generic Delegate** to achieve the same thing as we achieve in the first example.

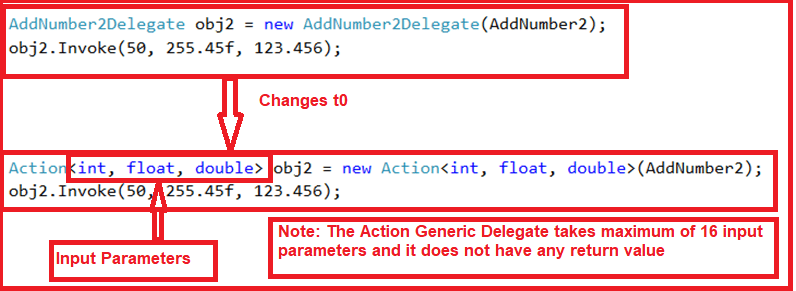
First, remove the three Delegates that we created.

**The following image shows how to use the Func Generic Delegate in C#.**



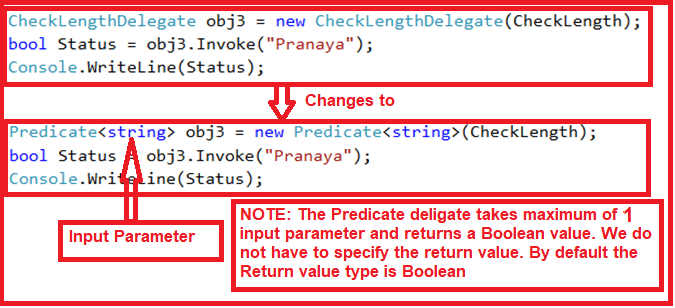
As shown in the above image, the **Func Generic Delegate** takes four parameters, the first three are input parameters and the last one is the return value. To the **Func Generic Delegate** constructor, we pass **the AddNumber1** method which is going to execute when we invoke the Func delegate.

**Let’s see how to use the Action Generic Delegate in C#. Have a look at the below image.**



As shown in the above image, the **Action Generic Delegate** takes three input parameters. To the **Action Generic Delegate** constructor, we pass the **AddNumber2** method which is going to execute when we invoke the Action delegate.

**Let’s see how to use the Predicate Generic Delegate in C#. Have a look at the below image.**



As shown in the above image, the **Predicate Generic Delegate** takes one string input parameter. To the **Predicate Generic Delegate** constructor, we pass the **CheckLength** method which is going to execute when we invoke the Predicate Generic delegate. This delegate can take a maximum of 1 input parameter and 0 return values. By default, it returns a Boolean value.

**Below is the complete code.**

**namespace** *GenericDelegateDemo*

**{**

**public** **class** GenericDelegates

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Func**<int**, **float**, **double**, **double>** obj1 = new Func**<int**, **float**, **double**, **double>(**AddNumber1**)**;

**double** Result = obj1.Invoke**(**100, 125.45f, 456.789**)**;

Console.WriteLine**(**Result**)**;

Action**<int**, **float**, **double>** obj2 = new Action**<int**, **float**, **double>(**AddNumber2**)**;

obj2.Invoke**(**50, 255.45f, 123.456**)**;

Predicate**<**string**>** obj3 = new Predicate**<**string**>(**CheckLength**)**;

**bool** Status = obj3.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**Status**)**;

Console.ReadKey**()**;

**}**

**public** **static** **double** AddNumber1**(int** no1, **float** no2, **double** no3**)**

**{**

**return** no1+ no2 + no3;

**}**

**public** **static** **void** AddNumber2**(int** no1, **float** no2, **double** no3**)**

**{**

Console.WriteLine**(** no1 + no2 + no3**)**;

**}**

**public** **static** **bool** CheckLength**(**string name**)**

**{**

**if** **(**name.Length **>** 5**)**

**return** **true**;

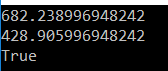
**return** **false**;

**}**

**}**

**}**

**Output:**



**Let’s see how to use Lambda Expression along with Generic Delegates in C#.**

**namespace** *GenericDelegateDemo*

**{**

**public** **class** GenericDelegates

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Func**<int**, **float**, **double**, **double>** obj1 = **(**x, y, z**)** =**>**

**{**

**return** x + y + z;

**}**;

**double** Result = obj1.Invoke**(**100, 125.45f, 456.789**)**;

Console.WriteLine**(**Result**)**;

Action**<int**, **float**, **double>** obj2 = **(**x, y, z**)** =**>**

**{**

Console.WriteLine**(**x + y + z**)**;

**}**;

obj2.Invoke**(**50, 255.45f, 123.456**)**;

Predicate**<**string**>** obj3 = new Predicate**<**string**>(**CheckLength**)**;

**bool** Status = obj3.Invoke**(**"Pranaya"**)**;

Console.WriteLine**(**Status**)**;

Console.ReadKey**()**;

**}**

**public** **static** **bool** CheckLength**(**string name**)**

**{**

**if** **(**name.Length **>** 5**)**

**return** **true**;

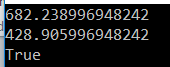
**return** **false**;

**}**

**}**

**}**

**Output:**



**Points to remember while working with C# Generic Delegates:**

1. Func, Action, and Predicate are generic inbuilt delegates that are present in the System namespace which is introduced in C# 3.
2. All these three delegates can be used with the method, [**anonymous method**](https://dotnettutorials.net/lesson/anonymous-method-c-sharp/), and l[**ambda expressions**](https://dotnettutorials.net/lesson/lambda-expression-csharp/).
3. The Func delegates can contain a maximum of 16 input parameters and must have one return type.
4. Action delegate can contain a maximum of 16 input parameters and does not have any return type.
5. The Predicate delegate should satisfy some criteria of a method and must have one input parameter and one Boolean return type either true or false which is by default. This means we should not have to pass that to the Predicate