C# - Delegate

A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter? How does C# handle the callback functions or event handler? The answer is - **delegate**.

A delegate is like a pointer to a function. It is a reference type data type and it holds the reference of a method. All the delegates are implicitly derived from System.Delegate class.

A delegate can be declared using **delegate** keyword followed by a function signature as shown below.

Delegate Syntax:

<access modifier> delegate <return type> <delegate\_name>(<parameters>)

The following example declares a Print delegate.

public delegate void Print(int value);

The Print delegate shown above, can be used to point to any method that has same return type & parameters declared with Print. Consider the following example that declares and uses Print delegate.

class Program

{

// declare delegate

public delegate void Print(int value);

static void Main(string[] args)

{

// Print delegate points to PrintNumber

Print printDel = PrintNumber;

// or

// Print printDel = new Print(PrintNumber);

printDel(100000);

printDel(200);

// Print delegate points to PrintMoney

printDel = PrintMoney;

printDel(10000);

printDel(200);

}

public static void PrintNumber(int num)

{

Console.WriteLine("Number: {0,-12:N0}",num);

}

public static void PrintMoney(int money)

{

Console.WriteLine("Money: {0:C}", money);

}

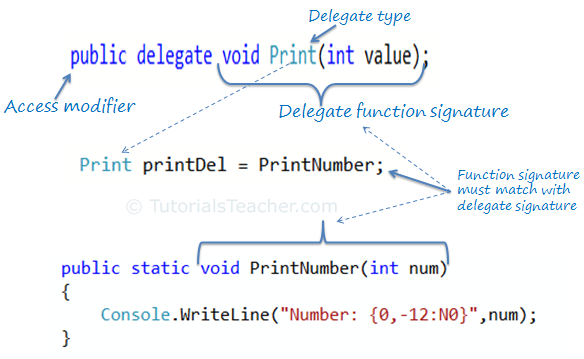
}

Output:

Number: 10,000   
Number: 200   
Money: $ 10,000.00   
Money: $ 200.00

In the above example, we have declared Print delegate that accepts *int* type parameter and returns void. In the Main() method, a variable of Print type is declared and assigned a PrintNumber method name. Now, invoking Print delegate will in-turn invoke PrintNumber method. In the same way, if the Print delegate variable is assigned to the PrintMoney method, then it will invoke the PrintMoney method.

The following image illustrates the delegate.

[](https://www.tutorialsteacher.com/Content/images/csharp/delegate.png)

Optionaly, a delegate object can be created using the new operator and specify a method name, as shown below:

Print printDel = new Print(PrintNumber);

Invoking Delegate

The delegate can be invoked like a method because it is a reference to a method. Invoking a delegate will in-turn invoke a method which id refered to. The delegate can be invoked by two ways: using () operator or using the Invoke() method of delegate as shown below.

Print printDel = PrintNumber;

printDel.Invoke(10000);

//or

printDel(10000);

Number: 10000   
Number: 10000

Pass Delegate as a Parameter

A method can have a parameter of a delegate type and can invoke the delegate parameter.

public static void PrintHelper(Print delegateFunc, int numToPrint)

{

delegateFunc(numToPrint);

}

In the above example, PrintHelper method has a delegate parameter of Print type and invokes it like a function:*delegateFunc(numToPrint)*.

The following example shows how to use PrintHelper method that includes delegate type parameter.

Example: Delegate parameter

class Program

{

public delegate void Print(int value);

static void Main(string[] args)

{

PrintHelper(PrintNumber, 10000);

PrintHelper(PrintMoney, 10000);

}

public static void PrintHelper(Print delegateFunc, int numToPrint)

{

delegateFunc(numToPrint);

}

public static void PrintNumber(int num)

{

Console.WriteLine("Number: {0,-12:N0}",num);

}

public static void PrintMoney(int money)

{

Console.WriteLine("Money: {0:C}", money);

}

}

Output:

Number: 10,000   
Money: $ 10,000.00

Multicast Delegate

The delegate can points to multiple methods. A delegate that points multiple methods is called a multicast delegate. The "+" operator adds a function to the delegate object and the "-" operator removes an existing function from a delegate object.

Example: Multicast delegate

public delegate void Print(int value);

static void Main(string[] args)

{

Print printDel = PrintNumber;

printDel += PrintHexadecimal;

printDel += PrintMoney;

printDel(1000);

printDel -=PrintHexadecimal;

printDel(2000);

}

public static void PrintNumber(int num)

{

Console.WriteLine("Number: {0,-12:N0}",num);

}

public static void PrintMoney(int money)

{

Console.WriteLine("Money: {0:C}", money);

}

public static void PrintHexadecimal(int dec)

{

Console.WriteLine("Hexadecimal: {0:X}", dec);

}

As you can see in the above example, Print delegates becomes a multicast delegate because it points to three methods - PrintNumber, PrintMoney & PrintHexadecimal. So invoking printDel will invoke all the methods sequentially.

Delegate is also used with [Event](https://www.tutorialsteacher.com/csharp/csharp-event), [Anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method), [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate), [Action delegate](https://www.tutorialsteacher.com/csharp/csharp-action-delegate).

 Points to Remember :

1. Delegate is a function pointer. It is reference type data type.
2. Syntax: *public delegate void <function name>(<parameters>)*
3. A method that is going to assign to delegate must have same signature as delegate.
4. Delegates can be invoke like a normal function or Invoke() method.
5. Multiple methods can be assigned to the delegate using "+" operator. It is called multicast delegate.

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.

Id

Name

Gender

Experience

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 should 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 EligibleToPromote. 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 type EligibleToPromote as parameters.

The method then loops thru each employee 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 hard coded 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. We then create an employeeList 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(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 EligibleToPromote 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 as 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 Sharp 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 Lambda expression is shown below. 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#

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

##### **C# Generic Delegates Types**

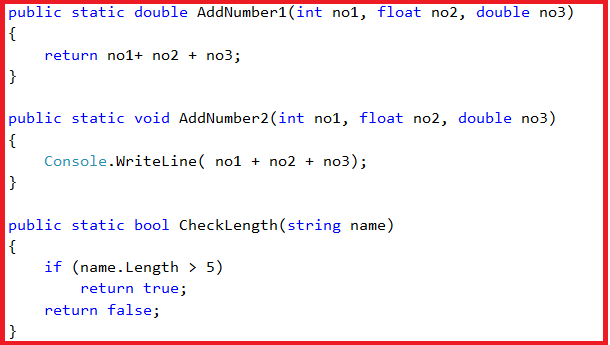
C# provides three built-in generic delegates, they are

1. Func
2. Action
3. Predicate

Before understanding the above three generic delegates, lets first understand how we use delegates to invoke methods.

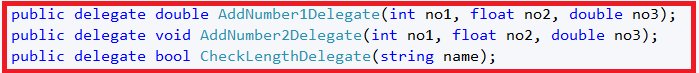
###### **Example:**

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

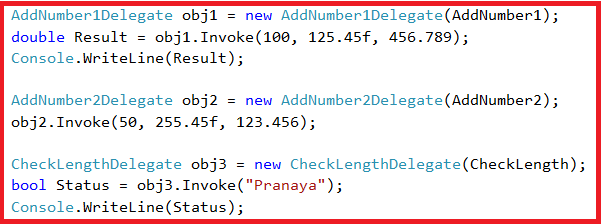


As you can see the **AddNumber1** method takes three parameters and returns a value of double type. Similarly, the**AddNumber2** method takes three parameters but it does not return any value. The third method i.e. the **CheckLength**method takes one string parameter and return a Boolean value.

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.



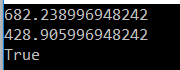
As you can see in the above image, we create three delegates. Let’s invoke the methods using the respective delegate instance as shown in the below image.



###### **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 which can do the job for us.

Let’s see what are the Generic Delegates provided by C#. C# provide three Generic Delegates, they are as follows

1. Func
2. Action
3. Predicate

##### **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#.

##### **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#.

##### **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 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#.

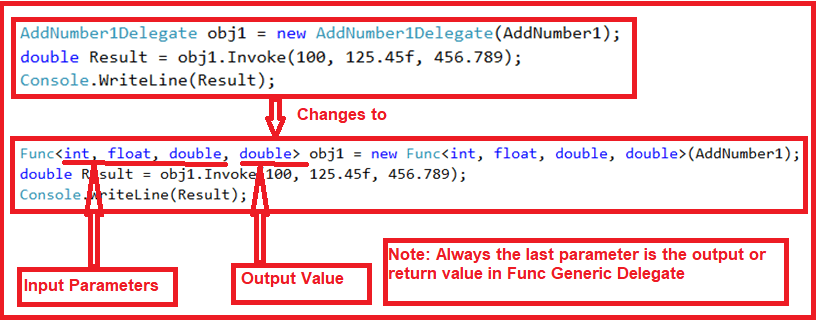
##### **Let’s understand the above three generic delegates in C# with an example.**

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.

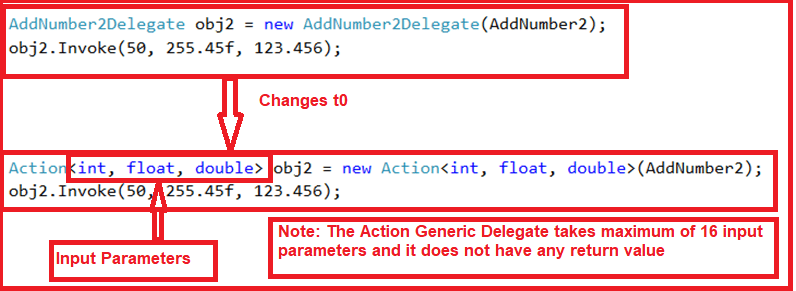
First, remove the three Delegates that we created.

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



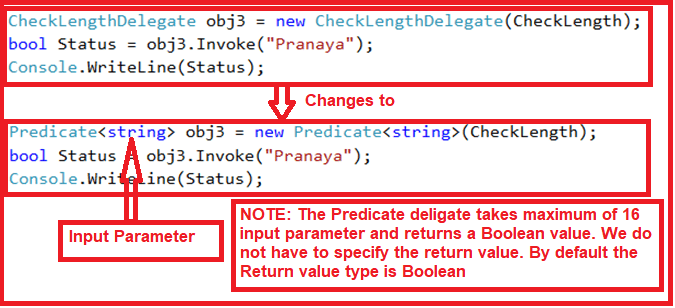
As shown in the above image, the **Func Generic Delegate** takes four parameters, the first three are input parameters and the last one is the return value. To the **Func Generic Delegate** constructor, we pass **the AddNumber1** method which is going to execute when we invoke the Func delegate.

###### **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 parameters. 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;

}

}

}