https://zetcode.com/csharp/func/

**C# Func**

*last modified December 3, 2021*

C# Func tutorial shows how to use the Func delegate in C#.

C# does not have plain functions only member functions (aka methods). And the methods are not first-class citizens. First-class functions allow us to create beautiful and powerful code, as seen in F#. C# ameliorates this somewhat with the usage of delegates and lambda expressions. Func is a built-in delegate which brings some functional programming features and helps reduce code verbosity.

**C# Func**

*Func* is a built-in generic delegate type. Others include Predicate and Action. Func can be used with a method, an anonymous method or a lambda expression.

Func can contains 0 to 16 input parameters and must have one return type. (There are 16 overloads of the Func delegate.)

public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);

For instance, this delegate encapsulates a method that has two parameters and returns a value of the type specified by the TResult parameter.

**C# Func simple example**

The following example is a simple demonstration of a C# Func delegate.

**Program.cs**

string GetMessage()

{

return "Hello there!";

}

Func<string> sayHello = GetMessage;

Console.WriteLine(sayHello());

In the example, we use the Func delegate which has no parameters and returns a single value.

string GetMessage()

{

return "Hello there!";

}

This is the function to which we refer with the help of the Func delegate.

Func<string> sayHello = GetMessage;

We refer to the GetMessage function via the Func delegate. The Func helps us create concise code.

Console.WriteLine(sayHello());

We invoke the function through the delegate and print the output.

$ dotnet run

Hello there!

**C# Func examples**

The following example uses Func to add values.

**Program.cs**

int Sum(int x, int y)

{

return x + y;

}

Func<int, int, int> add = Sum;

int res = add(150, 10);

Console.WriteLine(res);

We have a Sum method which adds two values. We refer to the method via the Func delegate.

Func<int, int, int> Add = Sum;

This Func delegate takes two parameters and returns a single value.

$ dotnet run

160

In the following example, we use a delegate with three input parameters.

**Program.cs**

int Sum(int x, int y, int z)

{

return x + y + z;

}

Func<int, int, int, int> add = Sum;

int res = add(150, 20, 30);

Console.WriteLine(res);

This time we refer to a method which takes three parameters.

$ dotnet run

200

Without the built-in Func delegate, we need to declare our custom delegate.

**Program.cs**

int Sum(int x, int y)

{

return x + y;

}

Add AddTwo = Sum;

int res = AddTwo(150, 10);

Console.WriteLine(res);

delegate int Add(int x, int y);

In this example, we refer to Sum method via a custom delegate type.

**C# Func with lambda expression**

C# lambda expression simplifies the creation of C# Funcs. Lambda expressions are created with the => lambda declaration operator.

**Program.cs**

Func<int, int, int> randInt = (n1, n2) => new Random().Next(n1, n2);

Console.WriteLine(randInt(1, 100));

In the example, we create a function which returns a random integer. The delegate accepts two values for the lower and upper bounds of the random range.

**C# Func Linq Where**

Many Linq methods take Func delegates as parameters. For instance, the Where method filters a sequence of values based on a predicate.

**Program.cs**

Func<string, bool> HasThree = str => str.Length == 3;

string[] words =

{

"sky", "forest", "wood", "cloud", "falcon", "owl" , "ocean",

"water", "bow", "tiny", "arc"

};

IEnumerable<string> threeLetterWords = words.Where(HasThree);

foreach (var word in threeLetterWords)

{

Console.WriteLine(word);

}

In the example, we have an array of words. With the help of the Func delegate, we filter all words that have three letters.

Func<string, bool> HasThree = str => str.Length == 3;

We declare a Func variable and assign a lambda expression to the variable. The method checks the length of the string and returns a boolean value.

IEnumerable<string> threeLetterWords = words.Where(HasThree);

We query the array and select strings according to the HasThree method.

$ dotnet run

sky

owl

bow

arc

**C# list of Func delegates**

Func delegates can be placed into containers.

**Program.cs**

var vals = new int[] { 1, 2, 3, 4, 5 };

Func<int, int> square = x => x \* x;

Func<int, int> cube = x => x \* x \* x;

Func<int, int> inc = x => x + 1;

var fns = new List<Func<int, int>>

{

inc, square, cube

};

foreach (var fn in fns)

{

var res = vals.Select(fn);

Console.WriteLine(string.Join(", ", res));

}

We put three Func delegates into a list. We go over the list and apply each delegate on the array.

$ dotnet run

2, 3, 4, 5, 6

1, 4, 9, 16, 25

1, 8, 27, 64, 125

**C# Func filter array**

In the example, we use Func to filter an array of users.

**Program.cs**

User[] users =

{

new (1, "John", "London", "2001-04-01"),

new (2, "Lenny", "New York", "1997-12-11"),

new (3, "Andrew", "Boston", "1987-02-22"),

new (4, "Peter", "Prague", "1936-03-24"),

new (5, "Anna", "Bratislava", "1973-11-18"),

new (6, "Albert", "Bratislava", "1940-12-11"),

new (7, "Adam", "Trnava", "1983-12-01"),

new (8, "Robert", "Bratislava", "1935-05-15"),

new (9, "Robert", "Prague", "1998-03-14"),

};

var city = "Bratislava";

Func<User, bool> livesIn = e => e.City == city;

var res = users.Where(livesIn);

foreach (var e in res)

{

Console.WriteLine(e);

}

record User(int id, string Name, string City, string DateOfBirth);

From the array of users, we get those that live in Bratislava.

var city = "Bratislava";

Func<User, bool> livesIn = e => e.City == city;

In the predicate, we test all user objects whose City attribute is equal to the city variable.

var res = users.Where(livesIn);

We pass the livesIn predicate to the Where method.

$ dotnet run

User { id = 5, Name = Anna, City = Bratislava, DateOfBirth = 1973-11-18 }

User { id = 6, Name = Albert, City = Bratislava, DateOfBirth = 1940-12-11 }

User { id = 8, Name = Robert, City = Bratislava, DateOfBirth = 1935-05-15 }

**C# Func filter by age**

We are going to filter a list by age.

**Program.cs**

var users = new List<User>

{

new (1, "John", "London", "2001-04-01"),

new (2, "Lenny", "New York", "1997-12-11"),

new (3, "Andrew", "Boston", "1987-02-22"),

new (4, "Peter", "Prague", "1936-03-24"),

new (5, "Anna", "Bratislava", "1973-11-18"),

new (6, "Albert", "Bratislava", "1940-12-11"),

new (7, "Adam", "Trnava", "1983-12-01"),

new (8, "Robert", "Bratislava", "1935-05-15"),

new (9, "Robert", "Prague", "1998-03-14"),

};

var age = 60;

Func<User, bool> olderThan = e => GetAge(e) > age;

var res = users.Where(olderThan);

foreach (var e in res)

{

Console.WriteLine(e);

}

int GetAge(User user)

{

var dob = DateTime.Parse(user.DateOfBirth);

return (int) Math.Floor((DateTime.Now - dob).TotalDays / 365.25D);

}

record User(int id, string Name, string City, string DateOfBirth);

We get all users older than sixty.

Func<User, bool> olderThan = e => GetAge(e) > age;

In the Func definition, we uset the GetAge method to determine the age of the user.

var res = users.Where(olderThan);

The olderThan function is applied with Where.

int GetAge(User user)

{

var dob = DateTime.Parse(user.DateOfBirth);

return (int) Math.Floor((DateTime.Now - dob).TotalDays / 365.25D);

}

The GetAge method parses the date of birth string and computes the current age.

$ dotnet run

User { id = 4, Name = Peter, City = Prague, DateOfBirth = 1936-03-24 }

User { id = 6, Name = Albert, City = Bratislava, DateOfBirth = 1940-12-11 }

User { id = 8, Name = Robert, City = Bratislava, DateOfBirth = 1935-05-15 }

**C# Predicate**

Predicate is a specialization of Func. Everything what can be done with Predicate can be done with a Func.

Predicates represent single argument functions that return a boolean value.

**Program.cs**

User[] users =

{

new (1, "John", "London", "2001-04-01"),

new (2, "Lenny", "New York", "1997-12-11"),

new (3, "Andrew", "Boston", "1987-02-22"),

new (4, "Peter", "Prague", "1936-03-24"),

new (5, "Anna", "Bratislava", "1973-11-18"),

new (6, "Albert", "Bratislava", "1940-12-11"),

new (7, "Adam", "Trnava", "1983-12-01"),

new (8, "Robert", "Bratislava", "1935-05-15"),

new (9, "Robert", "Prague", "1998-03-14"),

};

var age = 60;

Predicate<User> olderThan = e => GetAge(e) > age;

var res = Array.FindAll(users, olderThan);

foreach (var e in res)

{

Console.WriteLine(e);

}

int GetAge(User user)

{

var dob = DateTime.Parse(user.DateOfBirth);

return (int) Math.Floor((DateTime.Now - dob).TotalDays / 365.25D);

}

record User(int id, string Name, string City, string DateOfBirth);

We get all users older than 60.

Predicate<User> olderThan = e => GetAge(e) > age;

In a Predicate, we skip the return value, which is always a bool.

var res = Array.FindAll(users, olderThan);

The Array.FindAll method retrieves all the elements that match the conditions defined by the specified predicate.

int GetAge(User user)

{

var dob = DateTime.Parse(user.DateOfBirth);

return (int) Math.Floor((DateTime.Now - dob).TotalDays / 365.25D);

}

The GetAge method parses the date of birth string and computes the current age.

$ dotnet run

User { id = 4, Name = Peter, City = Prague, DateOfBirth = 1936-03-24 }

User { id = 6, Name = Albert, City = Bratislava, DateOfBirth = 1940-12-11 }

User { id = 8, Name = Robert, City = Bratislava, DateOfBirth = 1935-05-15 }

**C# pass Func as parameter**

In the next example, we pass Func delegate to a method.

**Program.cs**

var data = new List<Person>

{

new ("John Doe", "gardener"),

new ("Robert Brown", "programmer"),

new ("Lucia Smith", "teacher"),

new ("Thomas Neuwirth", "teacher")

};

ShowOutput(data, r => r.Occupation == "teacher");

void ShowOutput(List<Person> list, Func<Person, bool> condition)

{

var data = list.Where(condition);

foreach (var person in data)

{

Console.WriteLine("{0}, {1}", person.Name, person.Occupation);

}

}

record Person(string Name, string Occupation);

The example creates a list of persons. The ShowOutput method takes a Func as the second parameter. It returns all persons who are teachers.

void ShowOutput(List<Person> list, Func<Person, bool> condition)

We are passing a Func to the ShowOutput method. Methods cannot be passed as function arguments, only delegates.

$ dotnet run

Lucia Smith, teacher

Thomas Neuwirth, teacher

**C# Func compose**

We can compose Funcs by chaining.

**Program.cs**

var vals = new int[] { 1, 2, 3, 4, 5 };

Func<int, int> inc = e => e + 1;

Func<int, int> cube = e => e \* e \* e;

var res = vals.Select(inc).Select(cube);

foreach (var e in res)

{

Console.WriteLine(e);

}

We have an array of integers. We apply two functions on the array.

Func<int, int> inc = e => e + 1;

Func<int, int> cube = e => e \* e \* e;

One function increments the element and the other one cubes it.

var res = vals.Select(inc).Select(cube);

We apply the two functions on the array by chaining Select methods.

$ dotnet run

8

27

64

125

216

In this tutorial, we have worked with C# Func delegate.

Visit [C# tutorial](https://zetcode.com/lang/csharp/) or list [all C# tutorials](https://zetcode.com/all/#csharp).

[Home](https://zetcode.com/) [Facebook](https://www.facebook.com/zetcode7/) [Twitter](https://twitter.com/janbodnar) [Github](https://github.com/janbodnar" \o "Visit Github) [Subscribe](http://zetcode.us13.list-manage.com/subscribe?u=9def9ccd4c70dbbaf691f90fc&id=6556210f80) [Privacy](https://zetcode.com/privacy)

© 2007 - 2022 Jan Bodnar admin(at)zetcode.com