.Net Core

1. [AddTransient Vs AddScoped Vs AddSingleton in ASP.NET Core](#_AddTransient_Vs_AddScoped)
2. [Lazy initialization](#_Lazy_initialization)
3. [Why strings are immutable in .Net](#_Why_strings_are)
4. [Semaphore](#_Semaphore)
5. [Delegates](#_Delegates)

# AddTransient Vs AddScoped Vs AddSingleton in [ASP.NET](http://asp.net/) Core

<https://www.c-sharpcorner.com/article/understanding-addtransient-vs-addscoped-vs-addsingleton-in-asp-net-core/>

The below three methods define the lifetime of the services,

*1.       AddTransient*Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

*2.       AddScoped*Scoped lifetime services are created once per request.

*3.       AddSingleton*Singleton lifetime services are created the first time they are requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance.

|  |  |  |
| --- | --- | --- |
| **Service Type** | **In the scope of a given http request** | **Across different http requests** |
| **Transient** | New Instance | New Instance |
| **Scoped** | Same Instance | New Instance |
| **Singleton** | Same Instance | Same Instance |

# Lazy initialization

[*https://docs.microsoft.com/en-us/dotnet/framework/performance/lazy-initialization*](https://docs.microsoft.com/en-us/dotnet/framework/performance/lazy-initialization)

### Implementing a Lazy-Initialized Property

To implement a public property by using lazy initialization, define the backing field of the property as a [Lazy<T>](https://docs.microsoft.com/en-us/dotnet/api/system.lazy-1), and return the [Value](https://docs.microsoft.com/en-us/dotnet/api/system.lazy-1.value) property from the get accessor of the property.

class Customer

{

    private Lazy<Orders> \_orders;

    public string CustomerID {get; private set;}

    public Customer(string id)

    {

        CustomerID = id;

        \_orders = new Lazy<Orders>(() =>

        {

            // You can specify any additional

            // initialization steps here.

            return new Orders(this.CustomerID);

       });

    }

    public Orders MyOrders

    {

        get

        {

            // Orders is created on first access here.

            return \_orders.Value;

        }

    }

}

## Why strings are immutable in .Net

In C#, the CLR (Common Language Runtime) is responsible for determining where to store strings. It’s noted that a string is an array of characters. The CLR implements an array to store strings. Arrays are a fixed size data structure, meaning that they cannot be dynamically increased or decreased in size. Once an array is assigned a size, the size cannot be changed. To make an array larger, the data must be copied and cloned into a new array, which is put into a new block of memory by the CLR. If you edit a string, you are really not modifying that string; rather, the CLR is creating a new memory reference for the modified string, and the original string will get removed from memory via garbage collection.

# Semaphore

The semaphore class lets you set a limit on the number of threads that have access to a critical section. The class is used to control access to a pool of resources. **System.Threading.Semaphore** is the namespace for Semaphore because it has all the methods and properties required to implement Semaphore.

**Semaphore (Int32, Int32)**  
Initializes a new instance of the Semaphore class, specifying the initial number of entries and the maximum number of concurrent entries.

using System;

using System.Threading;

namespace Program

{

class Demo

   {

      static Thread[] t = new Thread[5];

      static Semaphore semaphore = new Semaphore(2, 2);

      static void DoSomething()

      {

         Console.WriteLine("{0} = waiting", Thread.CurrentThread.Name);

         semaphore.WaitOne(); // Entry point

         Console.WriteLine("{0} begins!", Thread.CurrentThread.Name);

         Thread.Sleep(1000);

         Console.WriteLine("{0} releasing...", Thread.CurrentThread.Name);

         semaphore.Release(); // Exit point

      }

      static void Main(string[] args)

      {

         for (int j = 0; j < 5; j++)

         {

            t[j] = new Thread(DoSomething);

            t[j].Name = "thread number " + j;

            t[j].Start();

         }

         Console.Read();

      }

   }

}

## Output

The following is the output

thread number 2 = waiting

thread number 0 = waiting

thread number 3 = waiting

thread number 1 = waiting

thread number 4 = waiting

thread number 2 begins!

thread number 1 begins!

thread number 2 releasing...

thread number 1 releasing...

thread number 4 begins!

thread number 3 begins!

thread number 4 releasing...

thread number 0 begins!

thread number 3 releasing...

thread number 0 releasing...

# Delegates

* Provides a good way to encapsulate the methods.
* Delegates are the library class in System namespace.
* These are the type-safe pointer of any method.
* Delegates are mainly used in implementing the call-back methods and events.
* Delegates can be chained together as two or more methods can be called on a single event.

## // C# program to illustrate the use of Delegates

using System;

namespace GeeksForGeeks {

// declare class "Geeks"

class Geeks {

// Declaring the delegates

// Here return type and parameter type should

// be same as the return type and parameter type

// of the two methods

// "addnum" and "subnum" are two delegate names

public delegate void addnum(int a, int b);

public delegate void subnum(int a, int b);

// method "sum"

public void sum(int a, int b)

{

Console.WriteLine("(100 + 40) = {0}", a + b);

}

// method "subtract"

public void subtract(int a, int b)

{

Console.WriteLine("(100 - 60) = {0}", a - b);

}

// Main Method

public static void Main(String []args)

{

// creating object "obj" of class "Geeks"

Geeks obj = new Geeks();

// creating object of delegate, name as "del\_obj1"

// for method "sum" and "del\_obj2" for method "subtract" &

// pass the parameter as the two methods by class object "obj"

// instantiating the delegates

addnum del\_obj1 = new addnum(obj.sum);

subnum del\_obj2 = new subnum(obj.subtract);

// pass the values to the methods by delegate object

del\_obj1(100, 40);

del\_obj2(100, 60);

// These can be written as using

// "Invoke" method

// del\_obj1.Invoke(100, 40);

// del\_obj2.Invoke(100, 60);

}}}

Output:

(100 + 40) = 140

(100 - 60) = 40

## Multicasting of a Delegate

* Delegates are combined and when you call a delegate then a complete list of methods is called.
* All methods are called in First in First out (FIFO) order.
* ‘+’ or ‘+=’ Operator is used to add the methods to delegates.
* ‘–’ or ‘-=’ Operator is used to remove the methods from the delegates list.

// C# program to illustrate the Multicasting of Delegates

using System;

class rectangle {

// declaring delegate

public delegate void rectDelegate(double height, double width);

// "area" method

public void area(double height, double width)

{

Console.WriteLine("Area is: {0}", (width \* height));

}

// "perimeter" method

public void perimeter(double height, double width)

{

Console.WriteLine("Perimeter is: {0} ", 2 \* (width + height));

}

// Main Method

public static void Main(String []args)

{

// creating object of class

// "rectangle", named as "rect"

rectangle rect = new rectangle();

// these two lines are normal calling of that two methods

// rect.area(6.3, 4.2);

// rect.perimeter(6.3, 4.2);

// creating delegate object, name as "rectdele"

// and pass the method as parameter by

rectDelegate rectdele = new rectDelegate(rect.area);

// also can be written as

// rectDelegate rectdele = rect.area;

// call 2nd method "perimeter"

// Multicasting

rectdele += rect.perimeter;

// pass the values in two method

// by using "Invoke" method

rectdele.Invoke(6.3, 4.2);

Console.WriteLine();

// call the methods with

// different values

rectdele.Invoke(16.3, 10.3);

}}

Output:

Area is: 26.46

Perimeter is: 21

Area is: 167.89

Perimeter is: 53.2