using System;

namespace RefvsOutDemo

{

class Program

{

static void Main(string[] args)

{

//Calling the Method with the REF arguments

int AdditionRef = 0;

int SubtractionRef = 0;

MathRef(200, 100, ref AdditionRef, ref SubtractionRef);

Console.WriteLine($"AdditionRef: {AdditionRef}");

Console.WriteLine($"SubtractionRef: {SubtractionRef}");

//Call the Method with the OUT arguments

int AdditionOut = 0;

int SubtractionOut = 0;

MathOut(200, 100, out AdditionOut, out SubtractionOut);

Console.WriteLine($"AdditionOut: {AdditionOut}");

Console.WriteLine($"SubtractionOut: {SubtractionOut}");

Console.ReadKey();

}

//Creating Method with Ref Parameters

public static void MathRef(int number1, int number2, ref int Addition, ref int Subtraction)

{

Addition = number1 + number2; //This will Update the Addition variable inside the Main method

Subtraction = number1 - number2; //This will Update the Subtraction variable inside the Main method

}

//Creating Method with out Parameters

public static void MathOut(int number1, int number2, out int Addition, out int Subtraction)

{

Addition = number1 + number2; //This will Update the Addition variable inside the Main method

Subtraction = number1 - number2; //This will Update the Subtraction variable inside the Main method

}

}

}

**finally** block we need to write the resource releasing logic or clean up the code. Resource releasing logic means un-referencing objects that are created in the try block. Since the statements written in the try and catch block are not guaranteed to be executed, we must place them in finally block.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void SomeMethod()

{

try

{

Console.WriteLine("Inside SomeMethod");

int num1 = 10, num2 = 0;

int result = num1 / num2; //Exception will be thrown here

Console.WriteLine($"Result: {result}");

}

finally

{

Console.WriteLine("SomeMethod finally Block");

}

}

static void Main(string[] args)

{

try

{

SomeMethod();

}

catch (Exception)

{

Console.WriteLine("Exception Caught");

}

Console.ReadKey();

}

}

}

Output

A black and white screen with white text

Description automatically generated

Delegates

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 call that method for execution.

A close-up of a white background

Description automatically generated

Now, we have our data somewhere in our program and we want to route those data over to this Handler method. How we can route the data into the Handler Method? We are going to route the data which is stored somewhere in our program over to this Handler Method using the pipeline i.e. using a delegate. In delegate, we need to define the parameters which will route the data from point A to point B (i.e. Handler method).

A blue and green line with arrows

Description automatically generated

The point that you need to remember while working with C# Delegates is that 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**, and **Number, Type, and Order of Parameters** of the delegates must and should be the same as the **Access Modifier, Return Type, and Number, Type, and Order of Parameters** 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.

Hoe to Use Delegate:

How to use the delegate means how we are going to use the delegate to move the data

using System;

namespace DelegatesDemo

{

public delegate void WorkPerformedHandler(int hours, WorkType workType);

class Program

{

static void Main(string[] args)

{

WorkPerformedHandler del1 =

new WorkPerformedHandler(Manager\_WorkPerformed);

del1(10, WorkType.Golf);

//del1.Invoke(50, WorkType.GotoMeetings);

Console.ReadKey();

}

public static void Manager\_WorkPerformed(int workHours, WorkType wType)

{

Console.WriteLine("Work Performed by Event Handler");

Console.WriteLine($"Work Hours: {workHours}, Work Type: {wType}");

}

}

public enum WorkType

{

Golf,

GotoMeetings,

GenerateReports

}

}

using System;

namespace DelegatesDemo

{

public delegate void CallbackMethodHandler(string message);

class Program

{

static void Main(string[] args)

{

Program obj = new Program();

CallbackMethodHandler del1 = new CallbackMethodHandler(obj.CallbackMethod);

//Here, I am calling the DoSomework function and I want the

//DoSomework function to call the delegate at some point of time

//which will invoke the CallbackMethod method

DoSomework(del1);

Console.ReadKey();

}

public static void DoSomework(CallbackMethodHandler del)

{

Console.WriteLine("Processing some Task");

del("Pranaya");

}

public void CallbackMethod(string message)

{

Console.WriteLine("CallbackMethod Executed");

Console.WriteLine($"Hello: {message}, Good Morning");

}

}

}

Callback using System;

namespace DelegatesDemo

{

public delegate void CallbackMethodHandler(string message);

class Program

{

static void Main(string[] args)

{

Program obj = new Program();

CallbackMethodHandler del1 = new CallbackMethodHandler(obj.CallbackMethod);

//Here, I am calling the DoSomework function and I want the

//DoSomework function to call the delegate at some point of time

//which will invoke the CallbackMethod method

DoSomework(del1);

Console.ReadKey();

}

public static void DoSomework(CallbackMethodHandler del)

{

Console.WriteLine("Processing some Task");

del("Pranaya");

}

public void CallbackMethod(string message)

{

Console.WriteLine("CallbackMethod Executed");

Console.WriteLine($"Hello: {message}, Good Morning");

}

}

}

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

The Delegates in C# are classified into two types as

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

**Where do we use Delegates in C#?**

Delegates are used in the following cases:

1. Event Handlers
2. Callbacks
3. Passing Methods as Method Parameters
4. LINQ
5. Multithreading

**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 the 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.

Func, Action And Predicate Delegates In C#

1. The Func delegate takes zero, one or more input parameters, and returns a value (with its out parameter).
2. The action takes zero, one or more input parameters, but does not return anything.
3. Predicate is a special kind of Func. It represents a method that contains a set of criteria mostly defined inside an if condition and checks whether the passed parameter meets those criteria or not.

The last parameter in the angle brackets <> is considered as the return type, and the remaining parameters are considered as input parameter types. It can have 0 - 16 input parameters.

class Program

{

static void Main(string[] args) public delegate TResult Func<in T1, in T2, out TResult>(T1 arg, T2 arg2)

{

Func<int,int,int> Addition = AddNumbers;

int result = Addition(10, 20);

Console.WriteLine($"Addition = {result}");

}

private static int AddNumbers(int param1, int param2 )

{

return param1 + param2;

}

}

Func with an Anonymous Method

Func<int,int,int> Addition = delegate (int param1, int param2)

{

return param1 + param2;

};

int result = Addition(10, 20);

Console.WriteLine($"Addition = {result}");

Func with Lambda Expression

Func<int, int, int> Addition = (param1, param2) => param1 + param2;

int result = Addition(10, 20);

Console.WriteLine($"Addition = {result}");

Action delegate

Here, the method AddNumbers takes 2 parameters but returns nothing. The results are assigned to an instance variable result.

Even an ion delegate can have 0 - 16 input parameters.

private static int result;

static void Main(string[] args)

{

Action<int, int> Addition = AddNumbers;

Addition(10, 20);

Console.WriteLine($"Addition = {result}");

}

private static void AddNumbers(int param1, int param2 )

{

result = param1 + param2;

}

Action with an Anonymous method

private static int result;

static void Main(string[] args)

{

Action<int, int> Addition = delegate (int param1, int param2)

{

result = param1 + param2;

};

Addition(10, 20);

Console.WriteLine($"Addition = {result}");

}

Action with a Lambda expression

private static int result;

static void Main(string[] args)

{

Action<int, int> Addition = (param1, param2) => result = param1 + param2; ;

Addition(10, 20);

Console.WriteLine($"Addition = {result}");

}

Predicate delegate

Syntax difference between predicate & func is that here in predicate, you don't specify a return type because it is always a bool.

.Let's check if the phone is an iPhone or not.

class Program

{

static void Main(string[] args)

{

Predicate<string> CheckIfApple = IsApple;

bool result = IsApple("I Phone X");

if(result)

Console.WriteLine("It's an IPhone");

}

private static bool IsApple(string modelName)

{

if (modelName == "I Phone X")

return true;

else

return false;

}

}

A predicate with the Anonymous method

Predicate < string > CheckIfApple = delegate(string modelName) {

if (modelName == "I Phone X") return true;

else return false;

};

bool result = CheckIfApple("I Phone X");

if (result) Console.WriteLine("It's an IPhone");

A predicate with Lambda expressions

Predicate < string > CheckIfApple = modelName => {

if (modelName == "I Phone X") return true;

else return false;

};

bool result = CheckIfApple("I Phone X");

if (result) Console.WriteLine("It's an IPhone");

Wonderful!

That's how you can directly use ready-made delegates without having them declared.

I hope this blog has given you the basic idea of delegate types. Be sure to implement them in your project.

IEnumerable vs IQueryable

using (SchoolContext db = new SchoolContext())

{

IEnumerable<Person> P = db.People;

IEnumerable<Person> P2 = P.Where(x => x.PersonId == 1).ToList<Person>();

}

In the block, x => x.PersonId == 1 is an anonymous method that can be executed like any other method. And the “Where” will execute the method once for each person, yielding values for which the method returns true. So you can say all the processing (anonymous function) calls happened on the client side.

IQueryable<Person> P = db.People;

var P3 = P.Where(x => x.PersonId == 1).ToList();

In the second block, x => x.PersonId == 1 is an expression tree(), that can be thought of as "is the 'PersonId' property == 1".

You can see the query in the following window. The filter criteria (where clause) is applied in the DB Query. So, performance-wise the Iquerable is much more important.

* If you create IQueryable, then the query may be converted to SQL and will run on the database server.
* If you create IEnumerable, then all rows will be pulled into memory as objects before running the query.

So in short we can say that IEnumerable is great for working with in-memory collections, but IQueryable allows for a remote data source such as a database or web service.

Tuples

Following are the different mechanisms available in C# to return multiple values from a method:

1. **Using Custom DataType:** You can return multiple values from a method by using a **custom data type (i.e. class)** as the return type of the method. But sometimes we don’t need or don’t want to use classes and objects because that’s just too much for the given purpose.
2. **Using Ref and Out variable:** You can also return more than one value from the method either by using the **“out”** or **“ref”** parameters. Using **“out”** or **“ref”** parameters is quite difficult to understand and moreover, the **“out” and “ref”** parameters will not work with the async methods.
3. **Using dynamic keyword:** You can also return multiple values from a method by using the dynamic keyword as the return type. The **dynamic** keyword was introduced in C# 4. But from a performance point of view, we probably don’t want to use dynamic.

As I already told tuples are not new to C# 7. They come with much more improvements in C# 7. So, let us first understand the Tuples which are there before C# 7, and then we will see what improvements they have done with types in C# 7.

using System;

using System.Collections.Generic;

namespace TulesDemo

{

class Program

{

static void Main()

{

var values = new List<double>() { 10, 20, 30, 40, 50 };

//Store the Result of Calulate Method in a variable of Tuple type

Tuple<int, double> t = Calulate(values);

//Access the First value using Item1 and second value using Item2 properties

Console.WriteLine($"There are {t.Item1} values and their sum is {t.Item2}");

Console.ReadKey();

}

//Declaring the return type as Tuple<int, double>

private static Tuple<int, double> Calulate(IEnumerable<double> values)

{

int count = 0;

double sum = 0.0;

foreach (var value in values)

{

count++;

sum += value;

}

//Creating an object of Tuple class by calling the static Create method

Tuple<int, double> t = Tuple.Create(count, sum);

//Returning the tuple instance

return t;

}

}

}

There are 3 major problems in the above code with Tuple:

1. The **First Problem** is that the Tuples in C# are classes, i.e. reference types. As reference types, the memory is allocated on the Heap Area and Garbage is collected only when they are no longer used. For applications where performance is a major concern, it can be an issue.
2. The **Second Problem** is that the elements in the tuple don’t have any names and you can only access them by using the build-in names such as Item1, Item2, Item3, etc. which are not meaningful at all. The **Tuple<T1, T2>** type does not provide any information about what the tuple actually represents which makes it a poor choice in public APIs.
3. The **Third Problem** is that you can use a maximum of 8 properties in a Tuple in C#. If you want to return more than 8 values from a method, then again the last argument of the Tuple must be another Tuple i.e. Tuple within another Tuple. This makes the syntax more difficult to understand.
4. using System;
5. using System.Collections.Generic;
6. namespace TulesDemo
7. {
8. class Program
9. {
10. static void Main()
11. {
12. var values = new List<double>() { 10, 20, 30, 40, 50 };
13. var result = Calulate(values);
14. Console.WriteLine($"There are {result.Item1} values and their sum is {result.Item2}");
15. Console.ReadKey();
16. }
18. private static (int, double) Calulate(IEnumerable<double> values)
19. {
20. int count = 0;
21. double sum = 0.0;
22. foreach (var value in values)
23. {
24. count++;
25. sum += value;
26. }
27. return (count, sum);
28. }
29. }
30. }
31. using System;
32. using System.Collections.Generic;
33. namespace TulesDemo
34. {
35. class Program
36. {
37. static void Main()
38. {
39. var values = new List<double>() { 10, 20, 30, 40, 50 };
40. var result = Calulate(values);
41. Console.WriteLine($"There are {result.count} values and their sum is {result.sum}");
42. Console.ReadKey();
43. }
45. private static (int count, double sum) Calulate(IEnumerable<double> values)
46. {
47. int count = 0;
48. double sum = 0.0;
49. foreach (var value in values)
50. {
51. count++;
52. sum += value;
53. }
54. return (count, sum);
55. }
56. }
57. }

**What are the Similarities Between Singleton and Static Class in C#?**

Before discussing the differences between them, let’s first discuss the similarities between Singleton vs Static Class in C#

1. Static Class and Singleton Class can have only one instance available in memory throughout the application.
2. They both hold the global state of an application that will be common for all clients.
3. Both Static Classes and Singleton Classes can be implemented as Thread-Safe.

**What are the Differences Between Singleton and Static Class in C#?**

The most important point you need to remember is that Static is a language feature, whereas Singleton is a Design Pattern. So, both belong to two different areas. With this in mind, let’s proceed and discuss the differences between Singleton vs Static class in C#.

Advertisements

1. We cannot create an instance of a static class in C#. Yes, one copy of the static class is available in memory, but we cannot create an instance of the static class as a developer. But, as a developer, we can create a single instance of a singleton class, and then we can reuse that singleton instance at many different places in the application.
2. When the compiler compiles the static class internally, it treats it as an Abstract and Sealed Class in C#. This is why we neither create an instance nor use the static class as the child class in inheritance. On the other hand, we can create a single instance of the Singleton class, as we can also use the Singleton class as a child class in C#.
3. The Singleton Class Constructor is always marked as private. This is why we cannot create an instance outside the singleton class. It provides either a public static property or a public static method whose job is to create the singleton instance only once and then return that singleton instance each and every time we call that public static property/method from outside of the singleton class.
4. A Singleton class can be initialized lazily or loaded automatically by CLR (Common Language Runtime) when the program or namespace containing the Singleton class is loaded, i.e., Eager Loading. A static class is generally initialized when it is loaded for the first time, and this may lead to potential classloader issues.
5. It is impossible to pass the static class as a method parameter in C #, whereas we can pass the singleton instance as a method parameter in C#.
6. In C#, it allows inheritance with the Singleton class. The Singleton class can be created as a Child class only. You cannot create child classes from the Singleton class. These are not possible with a static class. So, the Singleton class is more flexible than the Static Classes in C#.

**Memory Management of Static Class vs Singleton Class in C#:**

When the class is loaded, memory for static classes is allocated once in the high-frequency heap (a special heap for static data). This memory allocation is fixed and exists for the lifetime of the application. Since static classes are not instantiated and their resources are allocated directly in the memory, they live throughout the application life cycle and are only cleaned up when the application domain unloads, or the application exits.

Memory for a singleton instance is allocated on the heap when the instance is created, usually the first time it is requested. This memory remains allocated as long as there is a reference to the instance. The memory for the singleton instance is allocated when the instance is created and normally lasts for the duration of the application. However, unlike static classes, it’s possible to implement patterns like using a weak reference for the singleton instance, which allows for the instance to be garbage collected if there are no more references to it.

public class Singleton

{

private static Singleton \_instance;

private Singleton() { }

public static Singleton Instance

{

get

{

if (\_instance == null)

{

\_instance = new Singleton();

}

return \_instance;

}

}

public void DoSomething()

{

// Method implementation

}

}

The choice between using a Singleton or a static class in C# depends on the specific requirements of your application.

A Singleton is more appropriate if you need object-oriented features such as inheritance, interface implementation, and state management.

A static class is simpler and more efficient if you need a container for stateless functions or static data that is accessed globally.