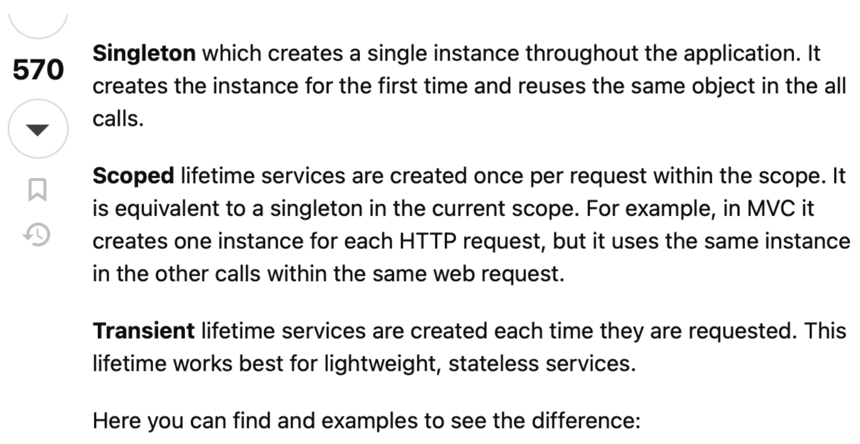
Dotnet Definations

Singleton , Transient and Scoped



570

Singleton which creates a single instance throughout the application. It creates the instance for the first time and reuses the same object in the all calls.

Scoped lifetime services are created once per request within the scope. It is equivalent to a singleton in the current scope. For example, in MVC it creates one instance for each HTTP request, but it uses the same instance in the other calls within the same web request.

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

Here you can find and examples to see the difference:

<u>Use Case -</u>

E-Commerce Website
Scoped - - Here, **ShoppingCartService** will be created once per user session/request, allowing items to be added and retrieved consistently within the same session.
E.g- Ideal for database contexts and services

Transient - (Doesn't Maintain the state)
Every time **IEmailService** is requested, a new instance of EmailService is created, which is efficient for stateless operations like sending an email.
Also - logging, email sending, etc.

Singleton - (It Maintain the state)
A single instance of **CacheService** is created and shared across the entire application, making it efficient for storing and retrieving cached data.
Also - configuration services, caching, and other shared resources.

Middleware -

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

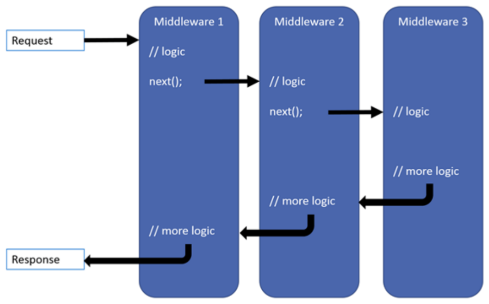
- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are middleware, also called middleware components. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a terminal middleware because it prevents further middleware from processing the request.

Dotnet Definations

Create a middleware pipeline with `WebApplication`

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Short-circuiting the request pipeline

When a delegate doesn't pass a request to the next delegate, it's called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a terminal middleware by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their next.Invoke statements. However, see the following warning about attempting to write to a response that has already been sent.

--

Content negotiation:

Content negotiation is the process of selecting one of multiple possible representations to return to a client, based on client or server preferences.
Json to xml

--

What is Kestrel and how does it differ from IIS?

Kestrel is a lightweight, cross-platform, and open-source web server for ASP.NET Core that runs on Linux, Windows, and Mac. It is designed to be fast and scalable, and it is the preferred web server for all new ASP.NET applications.

IIS (Internet Information Services), on the other hand, is a web server that is developed and maintained only by Microsoft. It is a Windows-specific web server that is not cross-platform.

Dotnet Definations

One of the main differences between Kestrel and IIS is that Kestrel is a cross-platform server that can run on Linux, Windows, and Mac, whereas IIS is Windows-specific. Another essential difference between the two is that Kestrel is fully open-source, whereas IIS is closed-source and developed and maintained only by Microsoft.

--

When I will go with Abstract Class and Interface ?

- If we have some common info that have to share with derived class
  Like speed, fuelCapacity with Vehicle class
- We can share that by using the **Abstract class hierarchy** for derived

----
--

Object Creation Method Overloading, Overriding, Abstract Class and Interface Confusion -

- **Overloading (Same class passing diff. variable)**

```
Math  n =  new Math();
n.AddMath (1,2);
n.AddMath (1.2, 2.3)
```

- **Overriding (Different class by creating Independent Object)**

```
News n = new News();// obj 1
n.newsItem();
TechNews tn = new TechNews (); // obj 2
tn.newsItem();
```

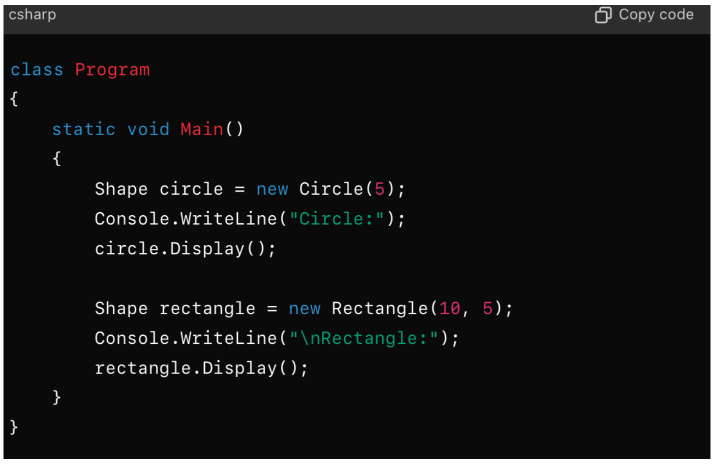
- **Abstract Class (Child class object and call both class methods)**

```
TechNews tn = new TechNews(); //Child Object
tn.GeneralNews(); //Calling Normal Method
tn.Technews(); //Calling Abstract Method
```

- **Interface (Abstract Class (Child class objects call methods))**

```
TechNews tn = new TechNews(); //Child Object
tn.Technews(); //Calling Main Interface Method
```

```
public abstract class Shape
{
    // Abstract property for the name of the shape
    public abstract string Name { get; }
    // Abstract method to calculate the area of the shape
    public abstract double CalculateArea();
    // Abstract method to calculate the perimeter of the shape
    public abstract double CalculatePerimeter();
    // A regular method to display information about the shape
    public void Display()
    {
        Console.WriteLine($"Shape: {Name}");
        Console.WriteLine($"Area: {CalculateArea()}");
        Console.WriteLine($"Perimeter: {CalculatePerimeter()}");
    }
}
```


csharp

```csharp
class Program
{
    static void Main()
    {
        Shape circle = new Circle(5);
        Console.WriteLine("Circle:");
        circle.Display();

        Shape rectangle = new Rectangle(10, 5);
        Console.WriteLine("\nRectangle:");
        rectangle.Display();
    }
}
```

--

How to call sealed class static Extension Methods -

Extension methods allow you to **add new functionality to existing types** without modifying them.

```
Singleton s1 = Singleton.GetInstance;

// singleton is sealed class and GetInstance is static Extension Method
Public static Singleton GetInstance { }
```

--

Advantages of Delegates

Encapsulation: They encapsulate method calls.
Flexibility: They make it easy to pass methods as parameters, enhancing flexibility.
Event Handling: They form the basis of the event handling model in .NET.
Callback Mechanism: They support implementing callback mechanisms.
Multicast: They can hold references to multiple methods.

--

Use Case of Custom Middleware/How you handle your Exception and Logging -
By creating custom middleware, you can encapsulate specific **logic**, such as **request logging** or **error handling**, and easily integrate it into your application's request pipeline.

Use Case1: Request Logging Middleware
Use Case2: Custom Error Handling Middleware

Dotnet Definations

a **tuple** is a **data structure** that allows you to **store a fixed-size collectio**n of items of potentially **different types.**

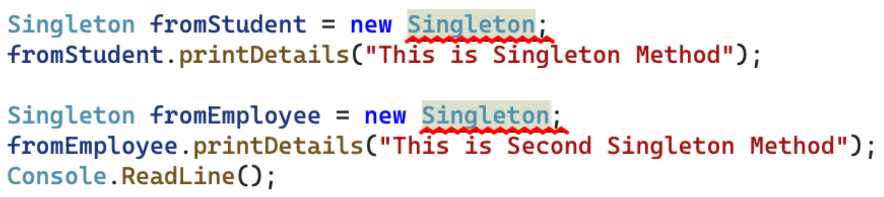
How to register Cross-Origin Resource Sharing (CORS) ?

```
public void ConfigureServices(IServiceCollection services)
    {
        // Define a CORS policy
        services.AddCors(options =>
        {
        }
        }
```

```
app.UseCors("AllowSpecificOrigin");
```

Why we use private constructor in Dotnet ?

- (Use in Singleton Design Patterns) To Ensures a class has **only one instance and provides a global point of access to it.**
- The private constructor prevents the creation of instances from outside the class, while a static property or method provides the single instance.
- Example -
  with private constructor you wont able to create multiple instance like below
-



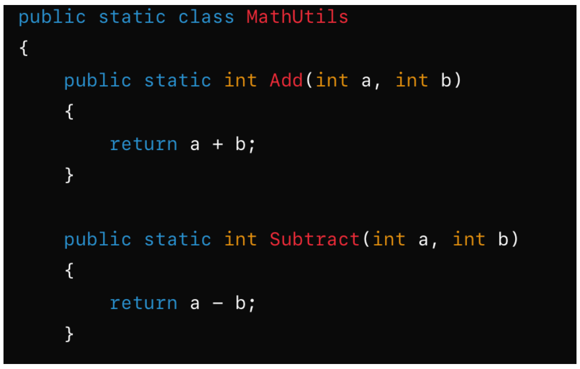
```
Singleton fromStudent = new Singleton;
fromStudent.printDetails("This is Singleton Method");

Singleton fromEmployee = new Singleton;
fromEmployee.printDetails("This is Second Singleton Method");
Console.ReadLine();
```

Static class , Can we create object of Static Class ?

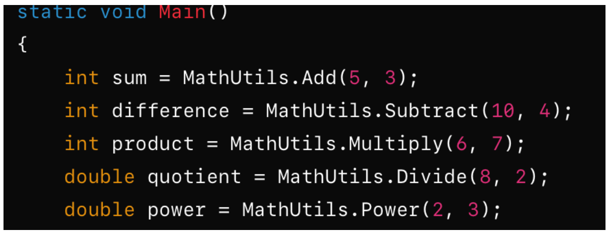
- Static class that **cannot be instantiated and can only contain static members.**
- Static classes are useful when you want to **create utility or helper classes**.

Dotnet Definations



```
public static class MathUtils
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static int Subtract(int a, int b)
    {
        return a - b;
    }
```

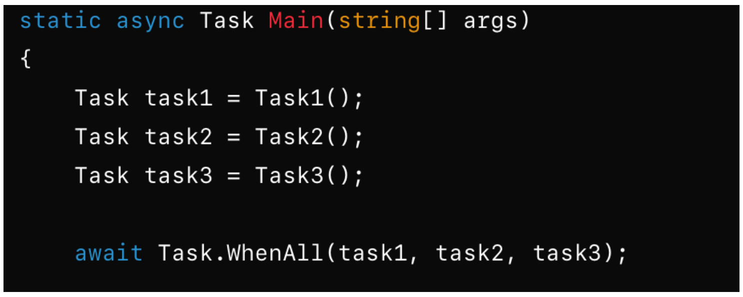


```
static void Main()
{
    int sum = MathUtils.Add(5, 3);
    int difference = MathUtils.Subtract(10, 4);
    int product = MathUtils.Multiply(6, 7);
    double quotient = MathUtils.Divide(8, 2);
    double power = MathUtils.Power(2, 3);
```

A **static constructor** in C# is a special type of constructor that is **used to initialize the static members of a class**.

Threading -
 allowing a program to perform **multiple operations simultaneously.**
We can achieve threading by Asynchronous programming by async and await keyword.

Example -

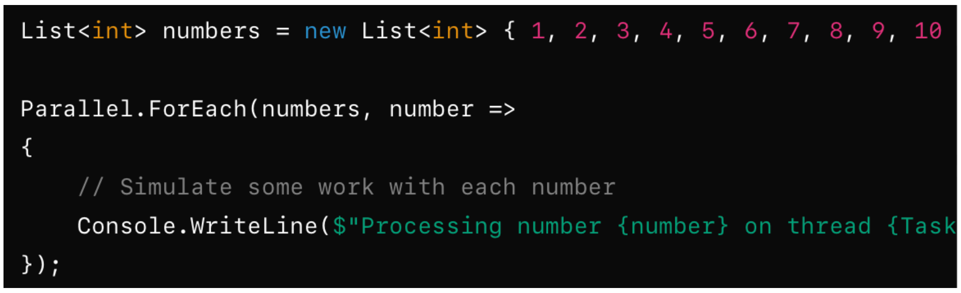


```
static async Task Main(string[] args)
{
    Task task1 = Task1();
    Task task2 = Task2();
    Task task3 = Task3();

    await Task.WhenAll(task1, task2, task3);
```

Parallel.ForEach is used to perform parallel iterations over a collection.

Dotnet Definations

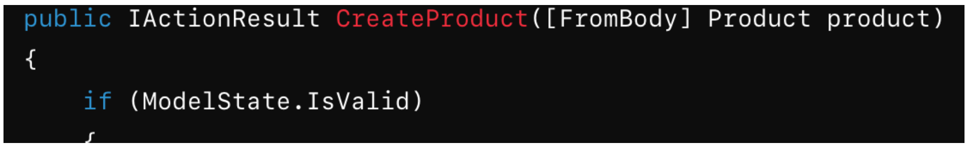


```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Parallel.ForEach(numbers, number =>
{
    // Simulate some work with each number
    Console.WriteLine($"Processing number {number} on thread {Task
});
```

Model Binding/Parameter Binding -
Model Binding in .NET Core API is a **mechanism** that allows the framework to automatically **map data from HTTP requests**
(such as **formBody, query strings, route data, and request bodies**) to parameters or model properties in your controller action methods.

This simplifies the process of handling incoming data and
ensures that it is correctly formatted and validated before it reaches your application logic.



```
public IActionResult CreateProduct([FromBody] Product product)
{
    if (ModelState.IsValid)
    {
```

- The Product class represents the data model.
- The CreateProduct action method expects a Product object in the request body.

Custom formatters in .NET Core are used to control how the data is serialized and deserialized when handling HTTP requests and responses. By default, .NET Core supports JSON and XML formatters, but custom formatters allow you to extend or replace these with your own implementations to support additional media types or custom serialization logic.

Example -
using Microsoft.AspNetCore.Mvc.Formatters;

```
public class CsvInputFormatter : TextInputFormatter
{
    public CsvInputFormatter()
    {
        SupportedMediaTypes.Add("text/csv");
        SupportedEncodings.Add(Encoding.UTF8);
    }
}
```

Common HTTP Headers

Here are some common HTTP headers used in .NET applications:

Dotnet Definations

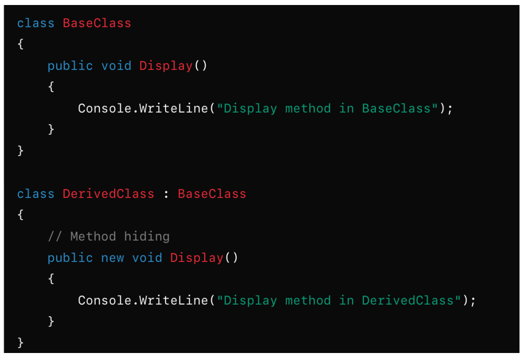
- **Request Headers**:
    - **Accept**: Specifies the media types that are acceptable for the response.
    - **Authorization**: Contains credentials for authenticating the client to the server.
    - **Content-Type**: Indicates the media type of the resource.
    - **User-Agent**: Contains information about the user agent originating the request.
    - **Cookie**: Sends stored cookies to the server.

IEnumerable is used to represent a collection that **can be enumerated**.
IEnumerator is used to **iterate over a collection.**

JWT Token/Authentication adding steps :

1. Install necessary **NuGet** packages.
2. Configure JWT settings in **appsettings.json**. ("SecretKey", "Issuer", "Audience","ExpirationMinutes")
3. Create a service to generate JWT tokens. (**Create Token Method**)
4. Configure JWT authentication in Startup.cs. (by **services.AddAuthentication** Mehod)
5. Use the **[Authorize]** attribute to protect your endpoints.
6. Implement a login endpoint to authenticate users and generate tokens.

Method Hiding -
- Same method in deffrent classes
- Like we use override keyword on Overriding purpose



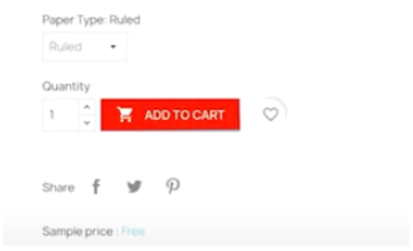
```
class BaseClass
{
    public void Display()
    {
        Console.WriteLine("Display method in BaseClass");
    }
}

class DerivedClass : BaseClass
{
    // Method hiding
    public new void Display()
    {
        Console.WriteLine("Display method in DerivedClass");
    }
}
```

How you handled your Production Issues ?
>> Use case , I was having "Color Grid" issue in our MI Ship Application

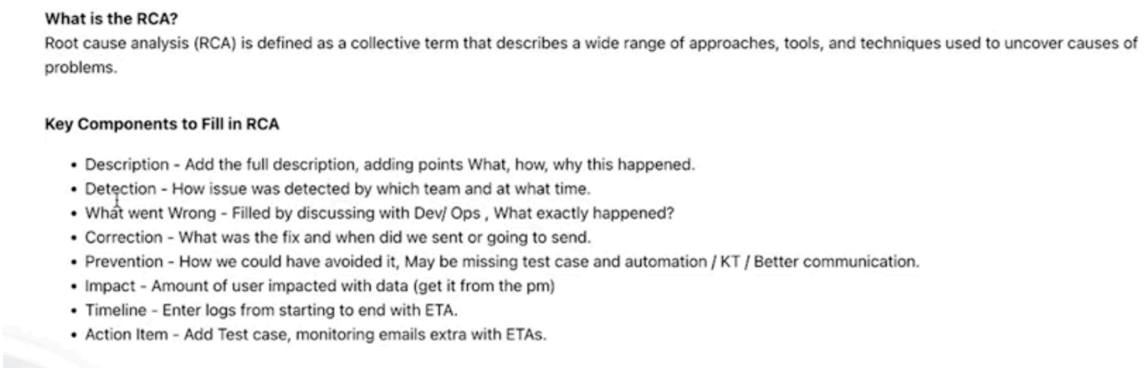
Dotnet Definations

>> Issue was it was "Red" instead of "Green"



Handle >>

1. Acknowledge the issue -
   Sometimes the issue with the Lower environment like "QA", "Staging"
   it **might issue with the configuration** when you're pushing your changes on Production
   So Acknowledge this issue first
2. Fill RCA- (Root Cause Analysis)



What is the RCA?

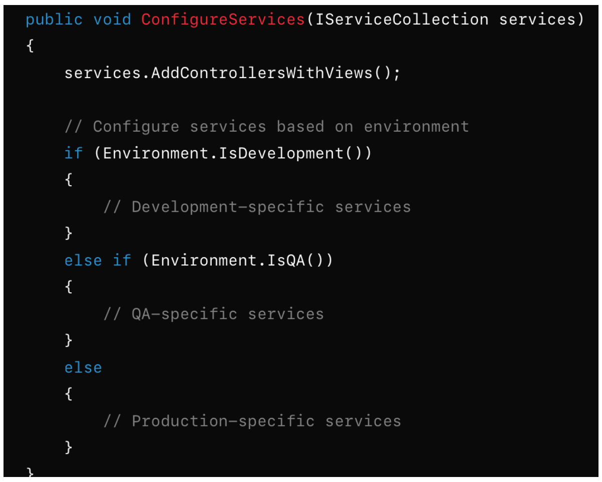
Root cause analysis (RCA) is defined as a collective term that describes a wide range of approaches, tools, and techniques used to uncover causes of problems.

Key Components to Fill in RCA

- Description - Add the full description, adding points What, how, why this happened.
- Detection - How issue was detected by which team and at what time.
- What went Wrong - Filled by discussing with Dev/ Ops , What exactly happened?
- Correction - What was the fix and when did we sent or going to send.
- Prevention - How we could have avoided it, May be missing test case and automation / KT / Better communication.
- Impact - Amount of user impacted with data (get it from the pm)
- Timeline - Enter logs from starting to end with ETA.
- Action Item - Add Test case, monitoring emails extra with ETAs.

How you handle your environments in your .Net core Application ?
>> create Setting file appsettings.Development.json, appsettings.QA.json:, appsettings.Production.json:
>> Use launchSettings.json to define profiles for different environments.
   Example - "ASPNETCORE_ENVIRONMENT": "Development"
>> Configure services and middleware in Startup.cs based on the environment.

Dotnet Definations



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    // Configure services based on environment
    if (Environment.IsDevelopment())
    {
        // Development-specific services
    }
    else if (Environment.IsQA())
    {
        // QA-specific services
    }
    else
    {
        // Production-specific services
    }
}
```

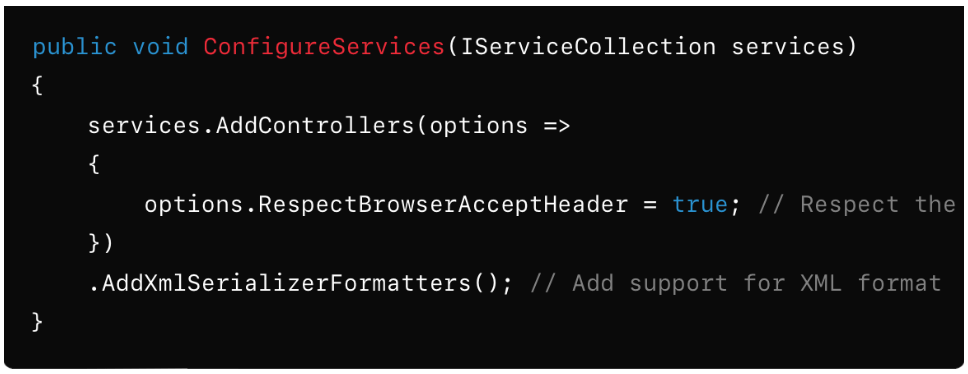
Handling Other Response Formats .Net Core Web API :

By default, .NET Core Web API controllers return data in JSON format.

While JSON is the default format,
ASP.NET Core **supports content negotiation**, which means it can return data in different formats (such as XML) based on the client's request.
However, you need to configure the supported formats in the Startup.cs file.

Here's an example of how to add support for XML format:

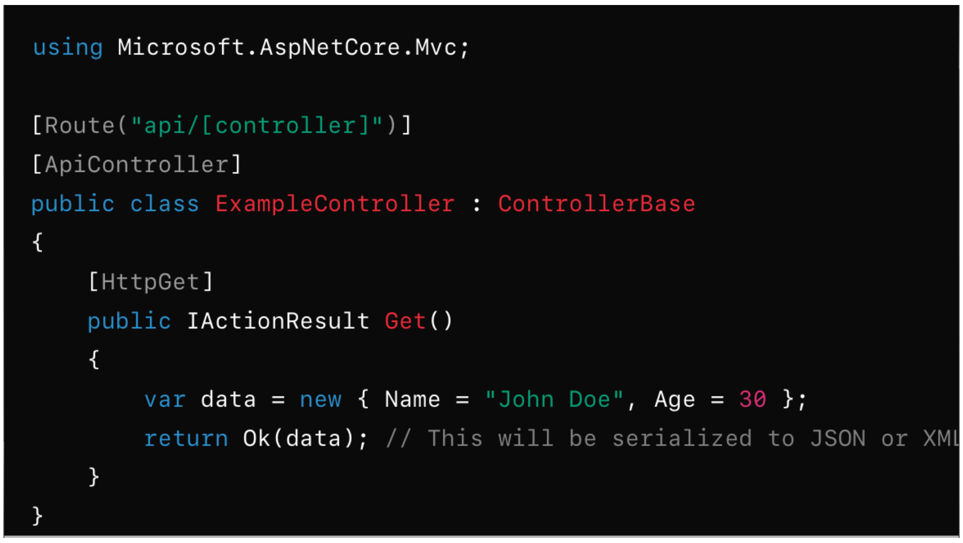
1. First, add the Microsoft.AspNetCore.Mvc.Formatters.Xml NuGet package to your project.
2. Then, configure the supported formats in the ConfigureServices method in Startup.cs:



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        options.RespectBrowserAcceptHeader = true; // Respect the
    })
    .AddXmlSerializerFormatters(); // Add support for XML format
}
```

With this configuration, your API can return XML if the client specifies **application/xml in the Accept header** of the request.

Dotnet Definations



```
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
[ApiController]
public class ExampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        var data = new { Name = "John Doe", Age = 30 };
        return Ok(data); // This will be serialized to JSON or XML
    }
}
```

In this example, if a client requests the data with **Accept: application/json,** the response will be in JSON format. If the client requests the data with **Accept: application/xml,** the response will be in XML format (assuming XML formatters are configured as shown above).

Destructor -
In C#, a destructor is a special method used to clean up resources before an object is reclaimed by the garbage collector. It is defined using a tilde (~) followed by the class name. Destructors are used to release unmanaged resources such as file handles, database connections, or unmanaged memory that the garbage collector does not handle automatically.

API versioning -
API versioning in .NET refers to the practice of managing **changes to an API over time without breaking existing client applications** that depend on it. It allows developers to introduce new features, improvements, and fixes while maintaining compatibility with older versions. Here's an overview of how API versioning can be implemented in .NET:

Using Block - To disposed unmanaged resources in that using block

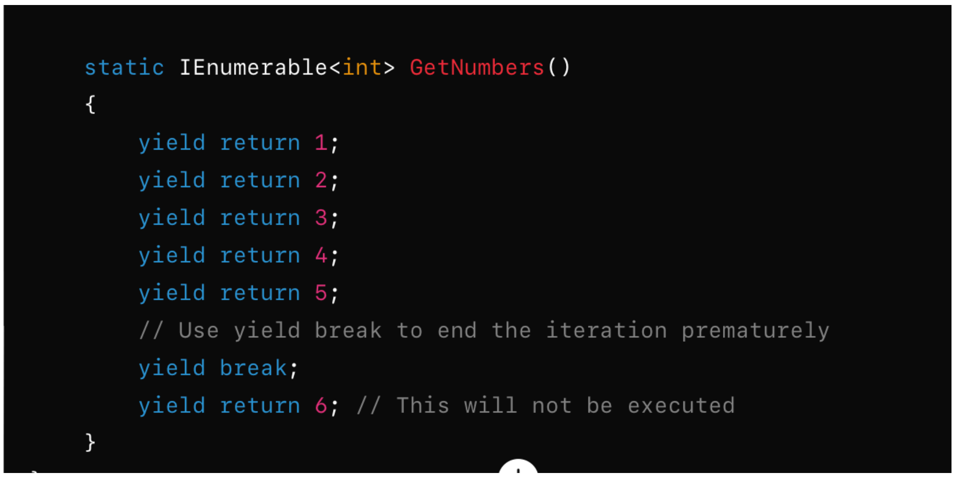
In .NET, the using block is used to ensure that resources are properly disposed of when they are no longer needed. This is particularly important for unmanaged resources, such as file handles, database connections, and network streams, which need to be released explicitly to avoid resource leaks and other issues.

Dotnet Definations

Yield : Iterator the Enumerator

The yield keyword in .NET is used in an iterator to provide a value to the enumerator object and to maintain the current position in the code. It simplifies the implementation of iterators by allowing you to write stateful iteration logic without explicitly managing the enumerator's state.
There are two primary uses of the yield keyword:

1. **yield return**: This is used to return each element one at a time. When the yield return statement is reached, the current location in the code is remembered, and execution is restarted from this location the next time the iterator is called.
2. **yield break**: This is used to end the iteration. It stops the iterator and signals the end of the collection.

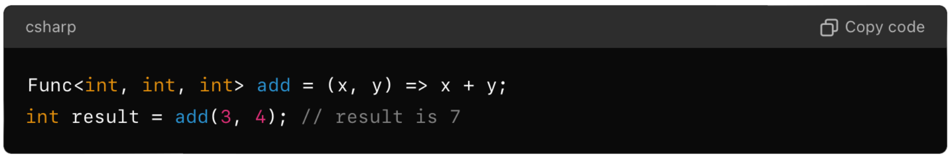


```
static IEnumerable<int> GetNumbers()
{
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    // Use yield break to end the iteration prematurely
    yield break;
    yield return 6; // This will not be executed
}
```

In .NET, both **Func and Predicate** are delegates used to represent methods. However, they have different purposes and usage patterns:

Func Delegate

- **Purpose**: Represents a method that can take one or more input parameters and returns a value.

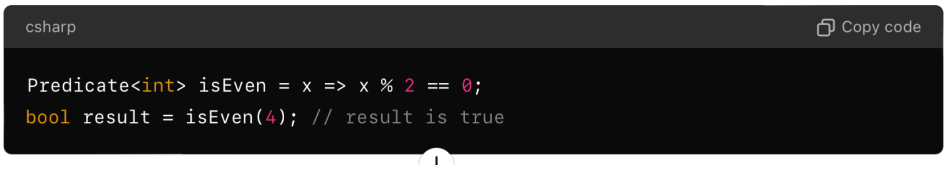


```csharp
Func<int, int, int> add = (x, y) => x + y;
int result = add(3, 4); // result is 7
```

Predicate Delegate

- **Purpose**: Represents a method that takes a single input parameter and returns a boolean value.

Dotnet Definations



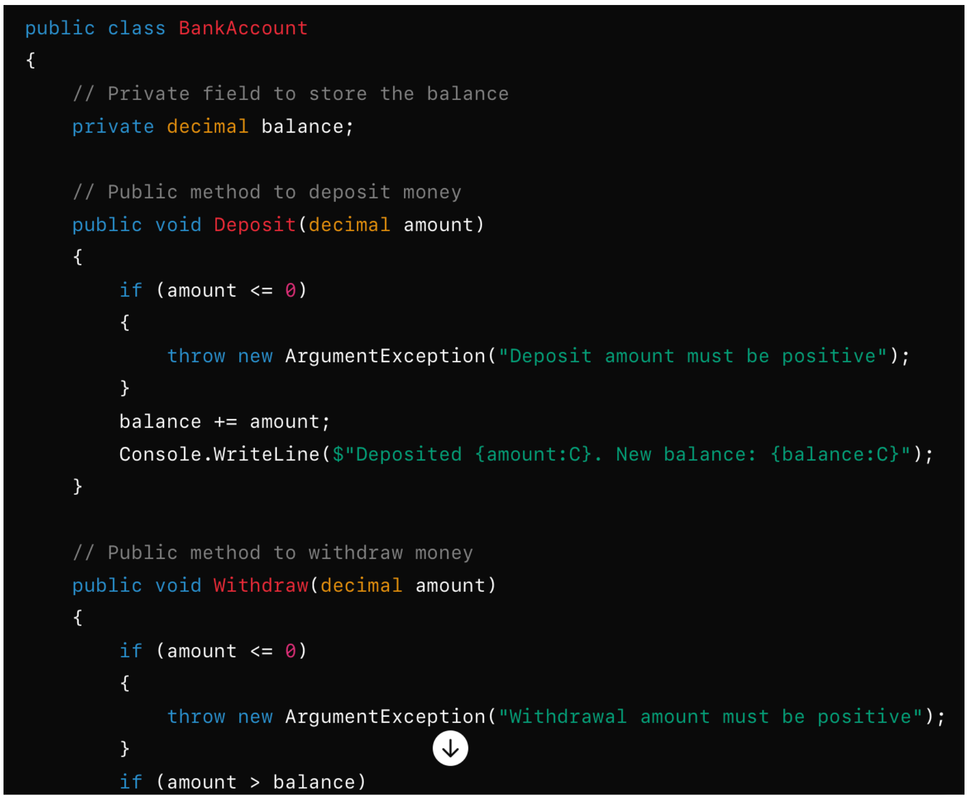
```csharp
Predicate<int> isEven = x => x % 2 == 0;
bool result = isEven(4); // result is true
```

Encapsulation Use Case -

Explanation:

1. **Private Field**: The balance field is private, meaning it cannot be accessed directly from outside the BankAccountclass.
2. **Public Methods**: The class provides public methods Deposit, Withdraw, and GetBalance to interact with the balance. These methods enforce rules and conditions for modifying the balance.

In this use case, encapsulation helps ensure that the balance can only be modified through controlled and validated operations, thereby protecting the integrity of the bank account's state.



```csharp
public class BankAccount
{
    // Private field to store the balance
    private decimal balance;

    // Public method to deposit money
    public void Deposit(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Deposit amount must be positive");
        }
        balance += amount;
        Console.WriteLine($"Deposited {amount:C}. New balance: {balance:C}");
    }

    // Public method to withdraw money
    public void Withdraw(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Withdrawal amount must be positive");
        }
        if (amount > balance)
```

Dotnet Definations

Execution Flow Diagram

1. **Program.cs**
   - **CreateHostBuilder -> Host.CreateDefaultBuilder -> ConfigureWebHostDefaults**
2. **Startup.cs**
   - **ConfigureServices -> Service registration**
   - **Configure -> Middleware pipeline configuration**
3. **Middleware Pipeline**
   - **Request -> [Middleware1] -> [Middleware2] -> ... -> [Routing Middleware]**
4. **Routing**
   - **Match URL to Route -> [Controller] -> [Action Method]**
5. **Action Execution**
   - **Perform logic -> Interact with Services -> Return Result**
6. **Response**
   - **Result -> [Middleware2] -> [Middleware1] -> Response to Client**

Diamond -
Multiple Inheritance wont support by OOPS(It cause diamond Problem)

To resolve that we use Interfaces to achieve Multiple Inheritance

```
class D : IB, IC
{
    void IB.Method()
    {
        // Implementation for IB
    }
    void IC.Method()
    {
        // Implementation for IC
    }
}
```