.NET INTERVIEW QUESTIONS

ROHATASH KUMAR

PREFACE

ABOUT THE BOOK

This book contains important .NET interview questions that will help you prepare for technical interviews. The questions are selected to cover the topics that are most commonly asked in real interviews, so you can be well-prepared and confident.

ABOUT THE AUTHOR

Rohatash Kumar has over 15 years of experience in software development. He has helped many candidates succeed in technical known tech companies by sharing his knowledge and guidance.

Topic	(Number of Questions)
1. Partial Class in C#	1
2. Constructor in C#	1
3. Encapsulation in C#	1
4. Single Responsibility Principle	1
5. MVC- Action Method Return Types	1
6. MVC-ViewData vs ViewBag vs TempData	1
7. MVC- Action Method Return Types	1
8. Cors in .NetCore	1
9. Design Patterns Introduction	1
10. C#-Singleton Design Pattern	1
11. C#-Factory Design Pattern	1
12. Angular Custom Pipe	1
13. Angular Subject vs BehaviorSubject	1
TOTAL	12

Partial Class in C#

Partial Class Introduction

1. A partial class allows a single class to be divided into two separate physical files. During compile time these files get compiled into a single class.
2. A partial class is created by using a partial keyword.
3. The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace.
4. All the parts must use the partial keyword.
5. All the parts must be available at compile time to form the final type.
6. All the parts must have the same accessibility such as public, private and so on.

The following class definition defines a partial class in C#:

```
//partial class
Public partial class PartialClass
{
}
```

Suppose you have a Person class. That definition is divided into two physical source files, Person1.cs and Person2.cs. Then these two files have a class that is a partial class. You compile the source code and then create it in a single class.

Real World Example

Recently, In My Project, I am using Entity Framework Database First Approach. In that case, the Entity Framework will create the models i.e. the classes based on the database and it creates the classes as partial classes. Next, I want to do some modifications with the auto-generated partial classes like adding some additional properties or adding some attributes. But, if I do the modification with the auto-generated partial classes, then my changes will be lost when I update the EDMX file. So, what I generally do is, create a partial class, and in that partial class, I do all the customization.

In the following figure you can see how the auto-generated code has partial classes and partial method.

The partial methods later can be extended for including custom logic. For instance, you can see in the following code for the preceding auto-generated class "tblCustomer" we have used partial methods to override the "OnCustomerCodeChanged" event to ensure that the customer code is not more than 6 length.

```
public partial class tblCustomer
{
    partial void OnCustomerCodeChanged()
    {
        if (_CustomerCode.Length > 6)
        {
            throw new Exception("Customer code can not be greater than 6");
        }
    }
}
```

So, by using partial classes and partial methods, LINQ and EF framework keep auto-generating classes and by using partial methods we can customize the class with our own logic.

Why Partial Classes?

The following are some important points which explain why use partial classes in C#.

1. **Modularization -** Partial classes enable breaking a class's definition into multiple files.
2. **Organized Development -** Developers can work on different parts of a class simultaneously without conflicts.
3. **Code Generation -** Commonly used in tools or frameworks to generate code, while developers can add custom logic separately.
4. **Maintenance -** Enhances code readability and maintainability by segmenting class logic.
5. **Extensibility -** Additional functionality can be added to existing classes without altering the original source code.

Advantages of Partial Class

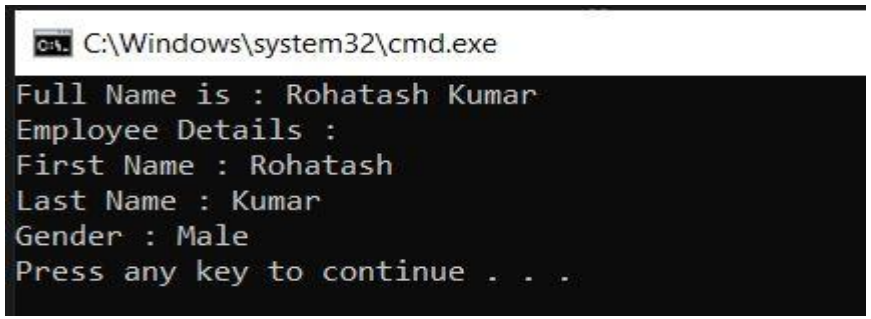
1. With the help of partial classes, multiple developers can work simultaneously in the same class in different files.
2. With the help of a partial class concept, you can split the UI of the design code and the business logic code to read and understand the code.
3. When you were working with automatically generated code, the code can be added to the class without having to recreate the source file like in Visual studio.
4. You can also maintain your application in an efficient manner by compressing large classes into small ones.

For Example

let us understand Partial Class with an example. First, create a console application and then add a class file with the name Student.cs. Once you add the class file, then copy and paste the following code into it. Here, please notices the class name is Student and in this class, we have declared 3 auto-implemented properties i.e. FirstName, LastName and Gender. Here, we have also declared 2 public methods i.e. DisplayName and DisplayStudentDetails.

```
using System;

namespace PartialClassDemo
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Gender { get; set; }

        public void DisplayFullName()
        {
            Console.WriteLine($"Full Name is : {FirstName}
{LastName}");
        }

        public void DisplayStudentDetails()
        {
            Console.WriteLine("Employee Details : ");
            Console.WriteLine($"First Name : {FirstName}");
            Console.WriteLine($"Last Name : {LastName}");
            Console.WriteLine($"Gender : {Gender}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Student emp = new Student
            {
                FirstName = "Rohatash",
                LastName = "Kumar",
                Gender = "Male"
            };
            emp.DisplayFullName();
            emp.DisplayStudentDetails();

        }
    }
}
```

Output



Splitting Above Class Definition into 2 Files using Partial Classes in C#

Now we split the above Student class definition into two different class files. One class file is going to contain all 3 public auto-implemented properties (FirstName, LastName and Gender) and the other class file is going to contain the two public methods i.e. DisplayName and DisplayStudentDetails that we have defined inside the Student class.

Now we need to add two class files with the name StudentOne.cs and StudentTwo.cs The StudentOne.cs class file going to contain all 3 public auto-implemented properties (FirstName, LastName, Gendar) and the StudentTwo.cs class file going to contain the two public DisplayName and DisplayStudentDetails methods. Even though the class file names are different, the class name is going to be the same, and in this case, we are providing the class name as Student in both the class file as well as we are making the class as partial by using the partial keyword.

StudentOne.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace partialExample
{
    public partial class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Gender { get; set; }
    }
}
```

StudentTwo.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace partialExample
{
    public partial class Student
    {
        public void DisplayFullName()
        {
            Console.WriteLine($"Full Name is : {FirstName} {LastName}");
        }

        public void DisplayStudentDetails()
        {
            Console.WriteLine("Employee Details : ");
            Console.WriteLine($"First Name : {FirstName}");
            Console.WriteLine($"Last Name : {LastName}");
            Console.WriteLine($"Gender : {Gender}");
        }
    }
}
```

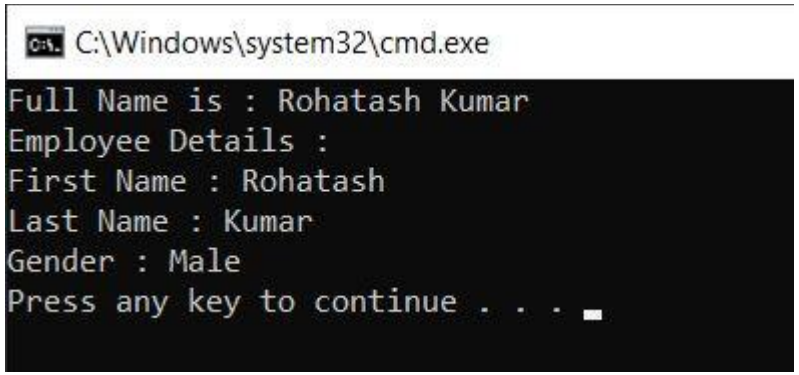
Using the Partial Class in C#

Now, we are going to use the Partial classes whose definition is split across two different class files. We need to use the Partial class just like a normal class. We can create an instance and we can also invoke the members using the instance.

```
using System;

namespace partialExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Student emp = new Student
            {
                FirstName = "Rohatash",
                LastName = "Kumar",
                Gender = "Male"
```

```
            };
            emp.DisplayFullName();
            emp.DisplayStudentDetails();

        }
    }
}
```

Output



When do we need to use Partial Class in C#?

There are several situations when splitting a class definition is desirable

1. **Organization -** For very large classes, breaking down the class into logical chunks (e.g., grouping methods by their functionality) can make the codebase easier to navigate and maintain.
2. When working on large projects, splitting a class over separate files allows multiple programmers to work on it simultaneously.
3. When working with automatically generated source code, the code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating windows form, Web service wrapper code, and so on.
4. **Code generation -** In scenarios where parts of the codebase are generated automatically (e.g. - designer files in Windows Forms, web service reference classes, or entities in Entity Framework), separating auto-generated code from custom code through partial classes helps in maintaining the codebase. It ensures that manually written code does not get overwritten by the auto-generation process.

Constructor in C#

Constructor Introduction

A constructor is a special method in a class that is automatically called when an object of that class is created. It is used to initialize the object. Constructor name will same as of Class name.

```
using System;

namespace Constructorproject
{
    public class Person
    {
        public string FirstName;
        public string LastName;

        // Constructor
        public Person(string firstName, string lastName)
        {
            Console.WriteLine(firstName + " " + lastName);
        }
    }
    class Program
    {
        static void Main()
        {
            // Usage
            Person person = new Person("Rohatash", "Kumar");

            Console.ReadLine();
        }
    }
}
```

When to use constructors in real applications?

1. **Initialization** - To set initial values for object attributes.
2. **Resource Allocation -** To allocate resources like memory or file handles.
3. **Dependency Injection -** To inject dependencies required by the object.

1. **Initialization**

To set initial values for object attributes.

```
using System;
namespace delegateproject
{
    public class Person
    {
        public string FirstName;
        public string LastName;

        // Initialization (Parameterized) Constructor
        public Person(string firstName, string lastName)
        {
            FirstName = firstName;
            LastName = lastName;
            //Console.WriteLine(firstName + " " + lastName);
        }
    }
    class Program
    {
        static void Main()
        {
            // Usage
            Person person = new Person("Rohatash", "Kumar");

            Console.ReadLine();
        }
    }
}
```

2. Resource Allocation

To allocate resources like memory or file handles. This example demonstrates how a constructor can be used to allocate resources (opening a file) and how the destructor can be used to clean up those resources (closing the file).

```
using System;
using System.IO;

public class FileReader
{
    private StreamReader _reader;
    public string FilePath { get; }

    // Constructor
    public FileReader(string filePath)
    {
        FilePath = filePath;

        try
        {
            // Allocate the resource (open the file)
            _reader = new StreamReader(FilePath);
            Console.WriteLine("File opened successfully.");
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine($"File not found:
{ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred:
{ex.Message}");
        }
    }

    // Method to read file content
    public string ReadFile()
    {
        if (_reader == null)
        {
            return "Reader is not initialized.";
        }

        try
        {
            return _reader.ReadToEnd();
        }
        catch (Exception ex)
        {
            return $"An error occurred while reading the
file: {ex.Message}";
        }
    }

    // Destructor to clean up resources
    ~FileReader()
    {
        if (_reader != null)
        {
            _reader.Close();
            Console.WriteLine("File reader closed.");
        }
    }
}

public class Program
```

```
{
    public static void Main(string[] args)
    {
        // Provide a valid file path
        string filePath = "example.txt";

        // Create an instance of FileReader
        FileReader fileReader = new FileReader(filePath);

        // Read the file content
        string content = fileReader.ReadFile();
        Console.WriteLine(content);
    }
}
```

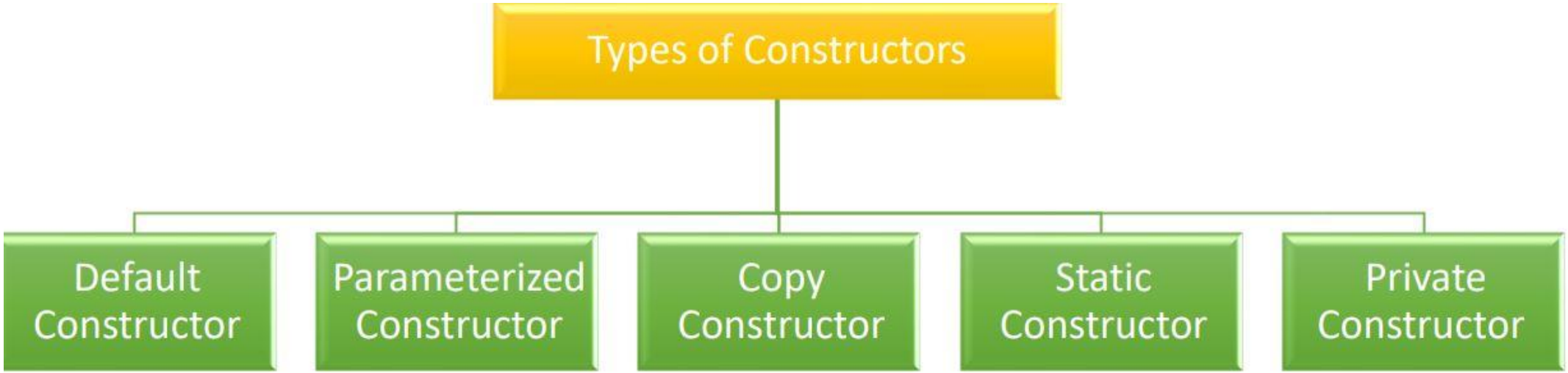
3. Dependency Injection

To inject dependencies required by the object. This example illustrates how dependency injection allows you to decouple the Service class from the specific implementation of the ILogger interface, promoting flexibility and testability.

```
using System;

public interface ILogger
{
    void Log(string message);
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine($"Log: {message}");
    }
}

public class Service
{
    private readonly ILogger _logger;

    // Constructor Injection
    public Service(ILogger logger)
    {
        _logger = logger;
    }

    public void DoWork()
    {
        _logger.Log("Work is being done.");
        // Perform work
        Console.WriteLine("Service is doing work.");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Manually create the dependency
        ILogger logger = new ConsoleLogger();

        // Inject the dependency into the service
        Service service = new Service(logger);

        // Use the service
        service.DoWork();
    }
}
```

Types of Constructors

There are several types of constructors, each serving a different purpose. Here are the main types of constructors.



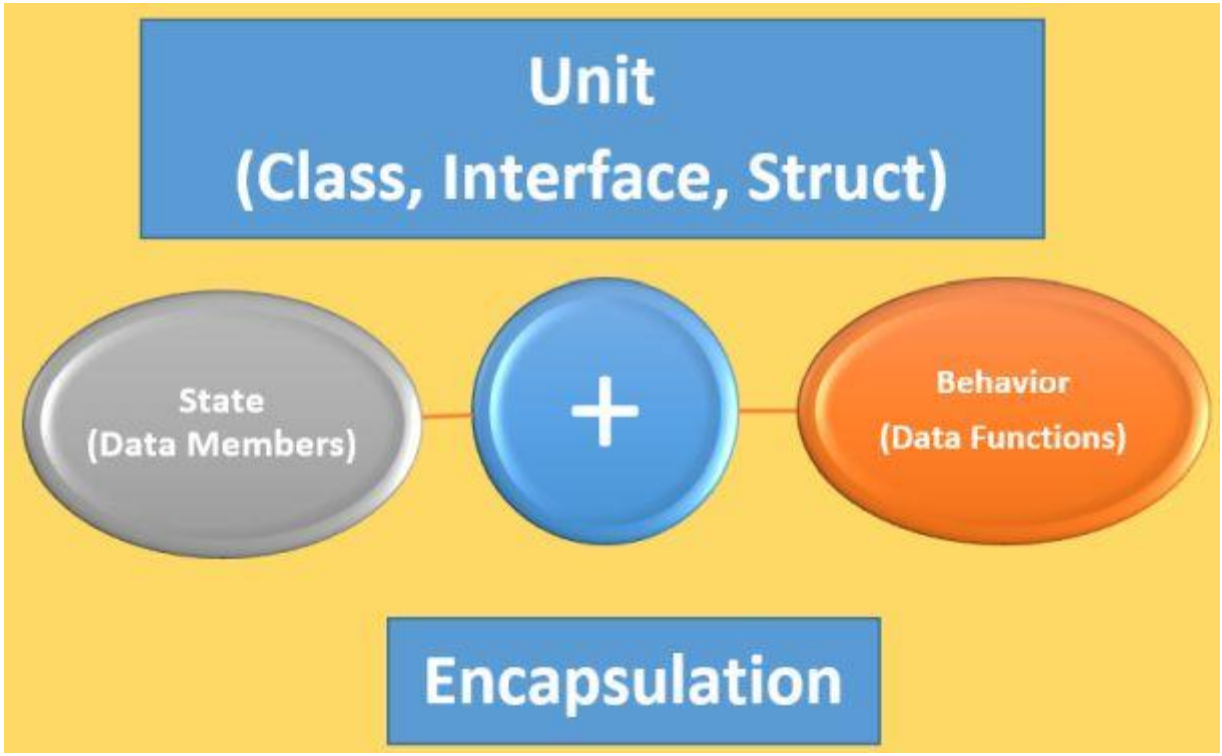
There are several types of constructors, each serving a different purpose. Here are the main types of constructors.

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Encapsulation in C#

Encapsulation Introduction

The process of binding or grouping the state(Data Members) and behaviour(Data Functions) together into a single unit(class, interface, struct) is called Encapsulation in C#.



We can define the above as following.

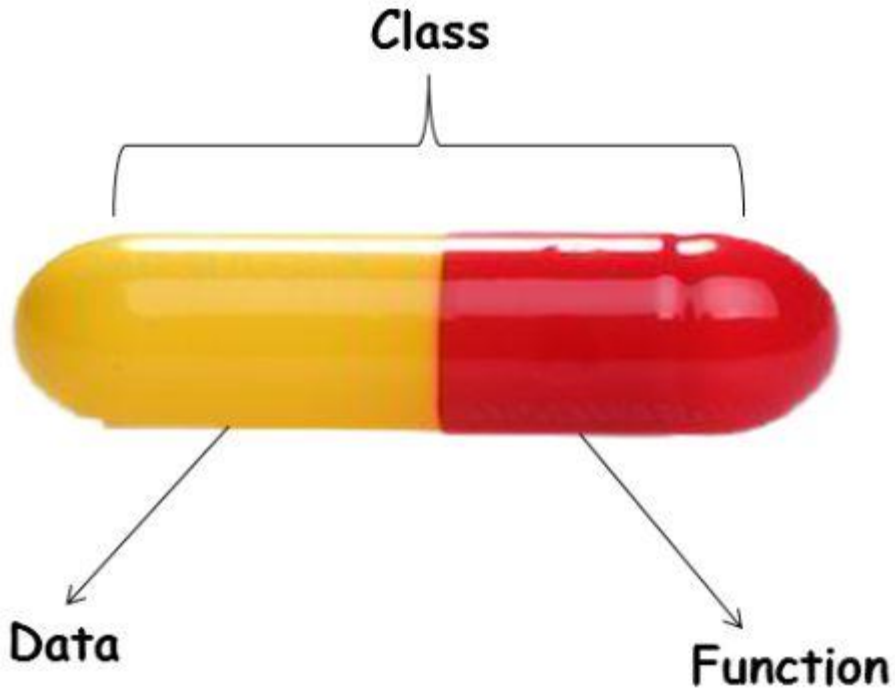
- **State** - Data Members
- **Behavior** - Data Functions
- **Unit** - class, interface, struct

The Encapsulation Principle ensures that the state and behavior of a unit (i.e., class, interface, struct, etc.) cannot be accessed directly from other units (i.e., class, interface, struct, etc.).

In C#, this is typically achieved through the use of classes. The idea behind encapsulation is to keep the implementation details of a class hidden from the outside world, and to only expose a public interface that allows users to interact with the class in a controlled and safe manner.

Real-World Example of Encapsulation

As we already discussed, one of the real-world examples of encapsulation is the Capsule, as the capsule binds all its medicinal materials within it. In the same way, C# Encapsulation, i.e., units (class, interface, enums, structs, etc) encloses all its data member and member functions within it.



Example

We want to create a **BankAccount** class with encapsulated attributes such as balance, and methods like **Deposit, Withdraw, and GetBalance**. We'll encapsulate these attributes by making them **private** and provide public methods to interact with them.

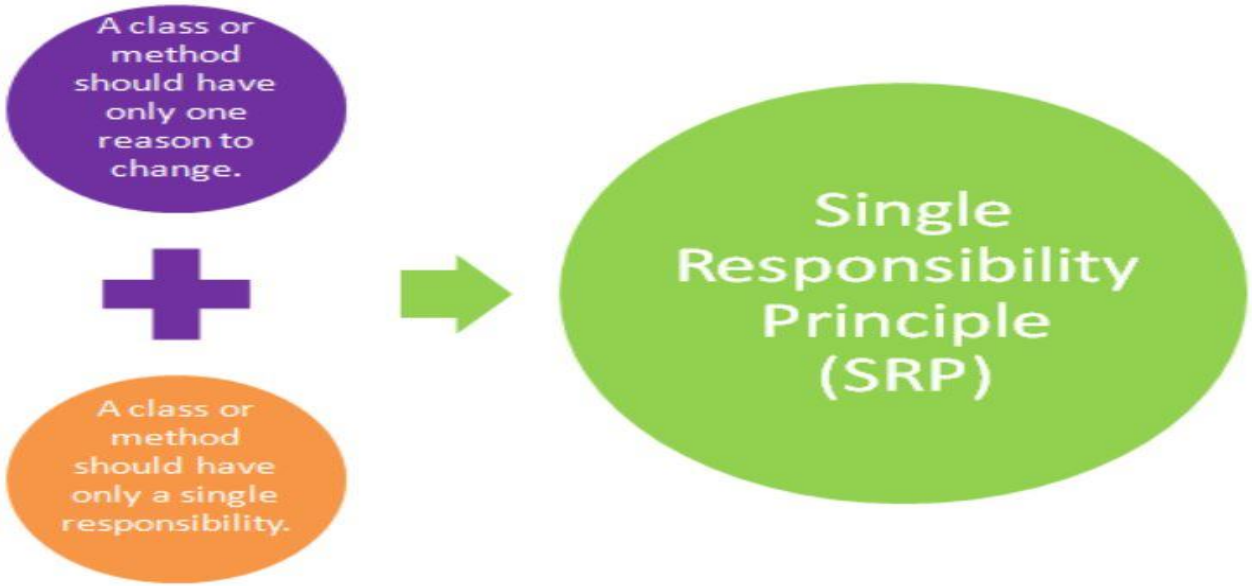
```
public class BankAccount
{
    private decimal balance;

    public BankAccount(decimal initialBalance)
    {
        balance = initialBalance;
    }

    public void Deposit(decimal amount)
    {
        balance = balance + amount;
    }

    public void Withdraw(decimal amount)
    {
        if (balance >= amount)
        {
            balance =balance- amount;
        }
        else
        {
            Console.WriteLine("Invalid withdrawal amount or
insufficient balance.");
        }
    }
    public decimal GetBalance()
    {
        return balance;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BankAccount myAccount = new BankAccount(1000);

        myAccount.Deposit(200);
        Console.WriteLine("Balance: " +
myAccount.GetBalance());

        myAccount.Withdraw(800);
        Console.WriteLine("Balance: " +
myAccount.GetBalance());
    }
}
```

Output

Single Responsibility Principle

Single Responsibility Principle

The Single Responsibility Principle (SRP) is one of the five SOLID principles of object-oriented programming and design. It states that a class should have only one reason to change, meaning that a class should only have one job or responsibility.



Violating the Single Responsibility Principle

Let's look at an example of code that violates the Single Responsibility Principle.

Example of a Class Violating SRP

```
public class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        // Validate order
        if (order.IsValid)
        {
            // Save order to database
            SaveOrderToDatabase(order);

            // Send confirmation email
            SendConfirmationEmail(order);
        }
    }

    private void SaveOrderToDatabase(Order order)
    {
        // Code to save order to the database
        Console.WriteLine("Order saved to database.");
    }

    private void SendConfirmationEmail(Order order)
    {
        // Code to send confirmation email
        Console.WriteLine("Confirmation email sent.");
    }
}
```

In this example, the OrderProcessor class has multiple responsibilities:

- validating the order(validation logic).
- Saving the order to the database.
- Sending a confirmation email.

Problems with Violating SRP

1. **Harder to Maintain -** Changes in the email sending process or database saving process will require changes to the OrderProcessor class.
2. **Difficult to Test -** Testing the OrderProcessor class will be more complex because it involves testing multiple responsibilities at once.
3. **Reduced Reusability -** The code for sending emails and saving to the database cannot be reused easily in other parts of the application.

Refactoring to Adhere to SRP

To adhere to the Single Responsibility Principle, we should refactor the code so that each class has only one responsibility. The below code define the refactor of above violating code.

Example

```
public interface IOrderRepository
{
    void SaveOrder(Order order);
}

public class OrderRepository : IOrderRepository
{
    public void SaveOrder(Order order)
    {
        // Code to save order to the database
        Console.WriteLine("Order saved to database.");
    }
}
public interface IEmailService
{
    void SendEmail(Order order);
}
public class EmailService : IEmailService
{
    public void SendEmail(Order order)
    {
        // Code to send confirmation email
        Console.WriteLine("Confirmation email sent.");
    }
```

```
}
public class OrderProcessor
{
    private readonly IOrderRepository _orderRepository;
    private readonly IEmailService _emailService;
    public OrderProcessor(IOrderRepository orderRepository, IEmailService emailService)
    {
        _orderRepository = orderRepository;
        _emailService = emailService;
    }
    public void ProcessOrder(Order order)
    {
        // Validate order
        if (order.IsValid)
        {
            // Save order to database
            _orderRepository.SaveOrder(order);
            // Send confirmation email
            _emailService.SendEmail(order);
        }
    }
}
```

Single Responsibility Principle Benefits

The SRP has many benefits to improve code complexity and maintenance. Some benefits of SRP are following,

1. **Reduction in complexity of a code** - A code is based on its functionality. A method holds logic for a single functionality or task. So, it reduces the code complexity.
2. **Increased readability, extensibility, and maintenance -** As each method has a single functionality so it is easy to read and maintain.
3. **Reusability and Reduced Error -** As code separates based functionality so if the same functionality uses somewhere else in an application then don't write it again.
4. **Better Testability-** In the maintenance, when a functionality changes then we don't need to test the entire model.
5. **Reduced Coupling -** It reduced the dependency code. A method's code doesn't depend on other methods.

SRP with Multiple Methods

a class adhering to the Single Responsibility Principle (SRP) which have multiple methods. The key is that all the methods should contribute to fulfilling the single responsibility or purpose of that class. Each method within the class should perform a task that supports the class's main responsibility.

Example of SRP with Multiple Methods

Let's consider an example where we have a ReportGenerator class responsible for generating reports. This class can have multiple methods, each performing a specific part of the report generation process, but all contributing to the single responsibility of generating a report.

```
public class ReportGenerator
{
    public void GenerateReport()
    {
        string data = FetchData();
        string report = FormatReport(data);
        SaveReport(report);
    }
    private string FetchData()
    {
        Console.WriteLine("Data fetched.");
        return "Sample Data";
    }
    private string FormatReport(string data)
    {
        // Code to format the report
        Console.WriteLine("Report formatted.");
        return $"Formatted Report: {data}";
    }
    private void SaveReport(string report)
    {
        // Code to save the report
        Console.WriteLine("Report saved.");
    }
}
class Program
{
    static void Main()
    {
        ReportGenerator reportGenerator = new
ReportGenerator();
        reportGenerator.GenerateReport();
    }
}
```

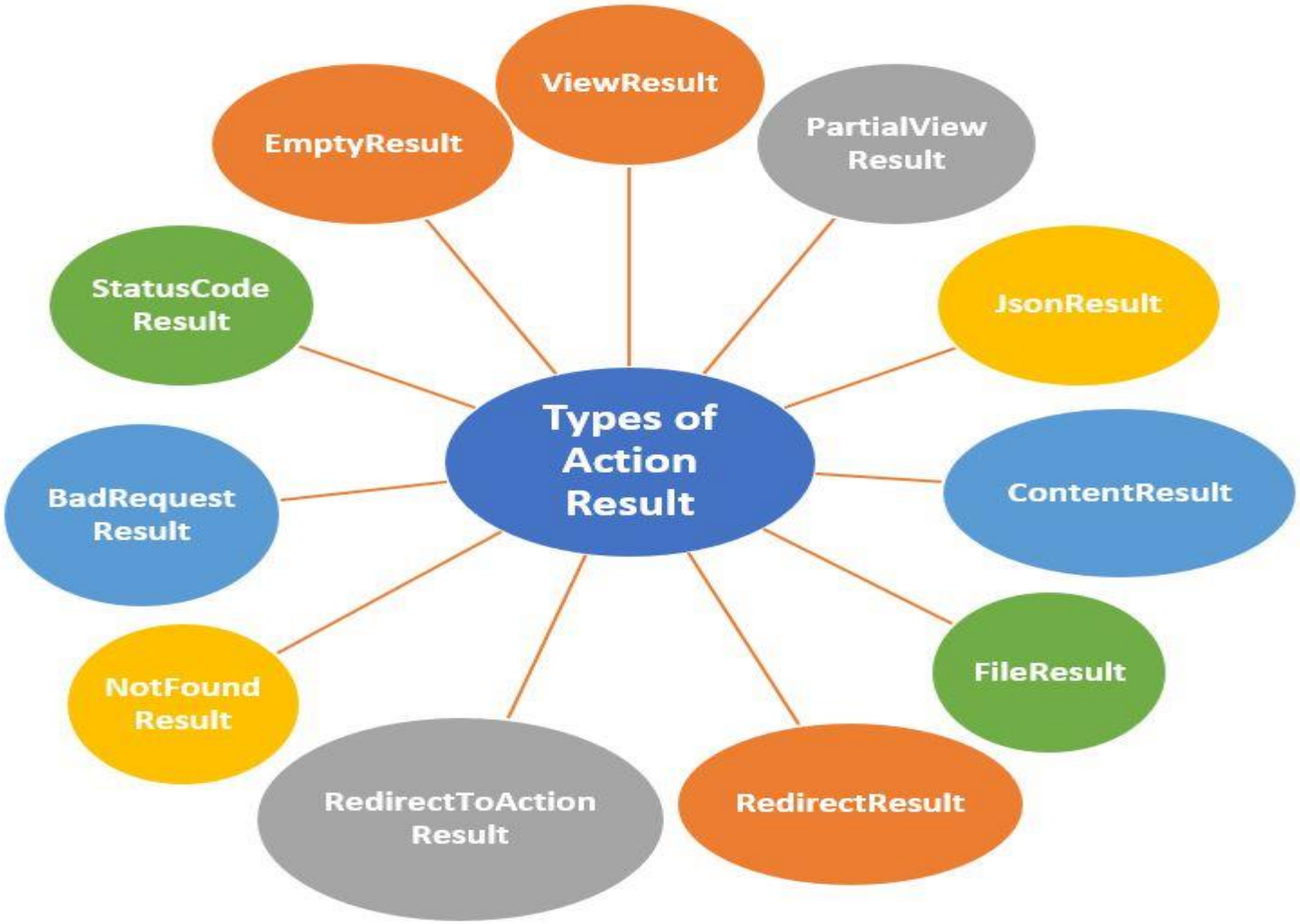
Explanation

GenerateReport Method - This is the main method responsible for generating the report. It coordinates the overall process by calling other private methods.
FetchData Method - Responsible for fetching data needed for the report.
FormatReport Method - Responsible for formatting the fetched data into a report.
SaveReport Method - Responsible for saving the formatted report.

Each method contributes to the single responsibility of the ReportGenerator class, which is to generate a report.

MVC- Action Method Return Types

A controller action method in ASP.NET MVC or ASP.NET Core can return various types of results based on the requirement. Below are the common return types with examples and full code.



Here are the names of the return types for a controller action method in ASP.NET MVC or Core.

1. ViewResult
2. PartialViewResult
3. JsonResult
4. ContentResult
5. FileResult
6. RedirectResult
7. RedirectToActionResult
8. NotFoundResult
9. BadRequestResult
10. StatusCodeResult
11. EmptyResult

1. ViewResult

This is used to return a view to the user.

```
public IActionResult Index()
{
    return View(); // Returns the default view associated
with this action.
}
```

2. PartialViewResult

Returns a partial view, often used for rendering parts of a web page.

```
public IActionResult LoadPartial()
{
    return PartialView("_PartialView"); // Returns a partial
view named "_PartialView".
}
```

3. JsonResult

Returns JSON data, typically used in APIs or AJAX requests.

```
public IActionResult GetJsonData()
{
    var data = new { Name = "John Doe", Age = 30 };
    return Json(data); // Returns JSON-encoded data.
}
```

4. ContentResult

Returns plain text or any string content.

```
public IActionResult GetText()
{
    return Content("This is plain text."); // Returns plain
text content.
}
```

5. FileResult

Returns a file for download or display.

```
public IActionResult DownloadFile()
{
    var fileBytes =
System.IO.File.ReadAllBytes("example.pdf");
    return File(fileBytes, "application/pdf",
"example.pdf"); // Returns a file for download.
```

```
}
```

6. RedirectResult

Redirects to a specific URL.

```
public IActionResult RedirectToGoogle()
{
    return Redirect("https://www.google.com"); // Redirects
to Google.
}
```

7. RedirectToActionResult

Redirects to another action method.

```
public IActionResult RedirectToAbout()
{
    return RedirectToAction("About", "Home"); // Redirects
to the About action of the Home controller.
}
```

8. NotFoundResult

Returns a 404 Not Found status.

```
public IActionResult PageNotFound()
{
    return NotFound(); // Returns a 404 status.
}
```

9. BadRequestResult

Returns a 400 Bad Request status.

```
public IActionResult InvalidRequest()
{
    return BadRequest("Invalid request."); // Returns a 400
status with a message.
}
```

10. StatusCodeResult

Returns a specific HTTP status code.

```
public IActionResult CustomStatus()
{
    return StatusCode(500); // Returns a 500 Internal Server
Error status.
}
```

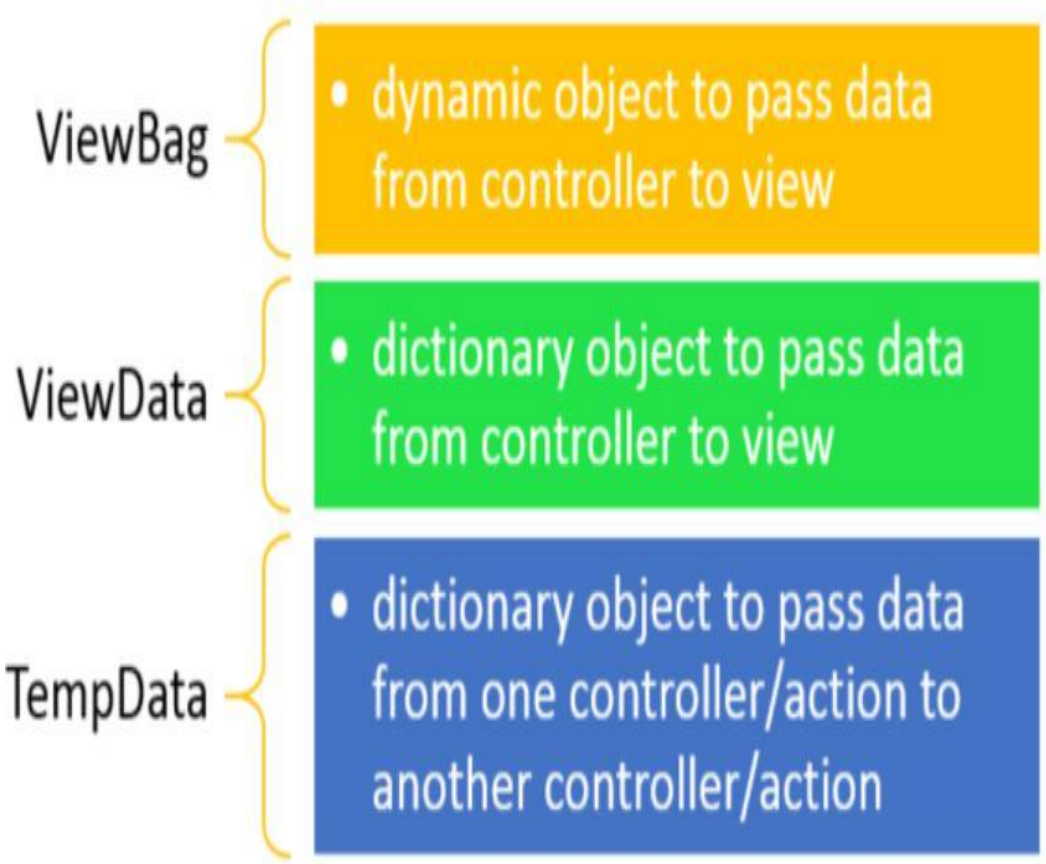
11. EmptyResult

Represents no result (does nothing).

```
public IActionResult DoNothing()
{
    return new EmptyResult(); // Does nothing.
}
```

MVC-ViewData vs ViewBag vs TempData

In ASP.NET MVC, ViewData, ViewBag, and TempData are used to pass data between controllers and views, but they each have their unique characteristics.



2. Data is accessed using string keys.
3. It's useful for passing data from the controller to the view.
4. Data is available only during the current request.

```
public ActionResult Index()
{
    ViewData["Message"] = "Hello from ViewData!";
    return View();
}

//View

<p>@ViewData["Message"]</p>
```

ViewBag

1. It's a dynamic object that uses the ExpandoObject under the hood.
2. Easier syntax compared to ViewData as it doesn't require casting.
3. Also used for passing data from the controller to the view.
4. Data is available only during the current request.

```
public ActionResult Index()
{
    ViewBag.Message = "Hello from ViewBag!";
    return View();
}

//View
<p>@ViewBag.Message</p>
```

TempData

1. It's a dictionary object derived from TempDataDictionary.
2. Stores data that is needed for more than a single request, useful for redirection.
3. Data persists until it is read, or the session expires.
4. Data is accessed using string keys.

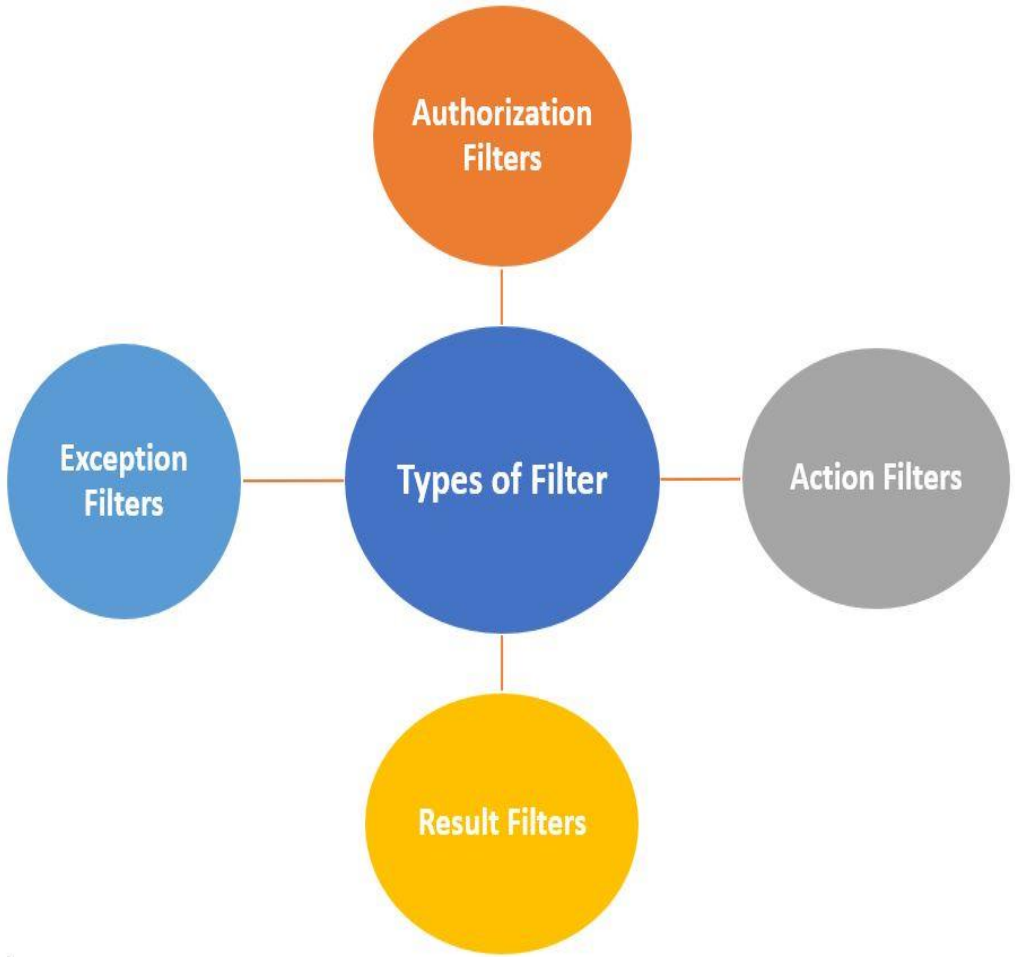
```
public ActionResult Index()
{
    TempData["Message"] = "Hello from TempData!";
    return RedirectToAction("NextAction");
}

public ActionResult NextAction()
{
    string message = TempData["Message"] as string;
    ViewBag.Message = message;
    return View();
}

//View

<p>@ViewBag.Message</p>
```

Here's a comparison of ViewData, ViewBag, and TempData in tabular form.

Feature	ViewData	ViewBag	TempData
Type	Dictionary (ViewDataDictionary)	Dynamic (ExpandoObject)	Dictionary (TempDataDictionary)
Syntax	ViewData["Key"]	ViewBag.Key	TempData["Key"]
Casting	Requires casting	No casting required	Requires casting
Scope	Current request	Current request	Subsequent requests (short-lived)
Usage	Pass data from controller to view	Pass data from controller to view	Pass data between controller actions
Persistence	Data is lost after the request is complete	Data is lost after the request is complete	Data persists until read or session expires

MVC- Action Method Return Types

In the context of the ASP.NET MVC framework, filters are used to execute code before or after specific stages in the request processing pipeline. Filters can handle cross-cutting concerns, such as authentication, authorization, logging, and error handling, across multiple actions or controllers.



There are several types of filters in MVC, each serving a different purpose.

1. Authorization Filter

Used to perform authentication and authorization before an action method is executed. It ensures that the user is authorized to access a particular action or controller.

Executed At - Before the Action Filter and Action Method.

Example - AuthorizeAttribute

You can use this filter to restrict access to certain users or roles.

Authorization Filter Example

Let's say you have an admin panel that only authorized users should access.

```
[Authorize(Roles = "Admin")]
public class AdminController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

Output

- If a user with the **Admin** role accesses the Index action, they see the admin panel view.
- If a user without the **Admin** role tries to access it, they are redirected to the login page or shown an unauthorized message.

2. Action Filter

Used to perform logic before and after an action method executes. Commonly used for logging, validation, or modifying data.

Executed At - Before and after the execution of an action method.

Example - ActionFilterAttribute

Action Filter Example

Suppose you want to log the execution time of each action method.

```
public class LogExecutionTimeAttribute : ActionFilterAttribute
{
    private Stopwatch stopwatch;
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        stopwatch = Stopwatch.StartNew();
    }
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        stopwatch.Stop();
        var executionTime = stopwatch.ElapsedMilliseconds;
        filterContext.HttpContext.Response.Headers.Add("X-Execution-Time", executionTime.ToString());
    }
}
[LogExecutionTime]
public class HomeController : Controller
{
    public ActionResult Index()
```

```
    {
        return View();
    }
}
```

Output

- The response header will include the execution time of the Index action.
- **Example -** X-Execution-Time: 15

3. Result Filter

Used to perform logic before and after a result (like a ViewResult or JsonResult) is executed. Useful for operations like modifying the response or adding headers.

Executed At - Before and after the ActionResult executes.

Example - OutputCacheAttribute (for caching responses)

Result Filter Example

Let's say you want to add a custom header to the response of an action.

```
public class AddCustomHeaderAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext filterContext)
    {
        filterContext.HttpContext.Response.Headers.Add("X-Custom-Header", "This is a custom header");
    }
}

[AddCustomHeader]
public class HomeController : Controller
{
    public ActionResult Index()
```

```
    {
        return View();
    }
}
```

Output

- The response header will include the custom header.
- **Example -** X-Custom-Header: This is a custom header

4. Exception Filter

Used to handle unhandled exceptions that occur in controllers or actions. It is commonly used to log errors or show custom error pages.

Executed At - Only when an exception occurs during the execution of an action method or result.

Example - HandleErrorAttribute

Exception Filter Example - Suppose you want to log exceptions globally in your application.

```
public class GlobalExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        var exception = filterContext.Exception;
        filterContext.Result = new RedirectToRouteResult(
            new System.Web.Routing.RouteValueDictionary
            {
                { "controller", "Error" },
                { "action", "Index" }
            });
        filterContext.ExceptionHandled = true;
    }
}
```

```
public static void
RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new GlobalExceptionFilter());
}
```

Output

- If an exception occurs, the user is redirected to the Error controller's Index action.
- The exception details are logged.

your application to specify which domains are permitted to access its resources. By default, web browsers enforce the **same-origin policy**, which restricts web pages from making requests to a domain different from the one that served the web page. CORS is used to relax this restriction and allow cross-origin requests in a controlled and secure manner.

Understanding CORS

When a web page on Domain A tries to access resources (like APIs) from Domain B, the browser checks the CORS policy set by Domain B. CORS is implemented by adding specific HTTP headers to the server response.

Why CORS

Here is why CORS is important.

1. To Enable Cross-Origin Requests

- Browsers enforce a security policy called **Same-Origin Policy**, which restricts web pages from making requests to a domain different from the one that served the web page.
- CORS provides a way to bypass this restriction by explicitly allowing certain cross-origin requests.

2. To Improve API Usability

- If your .NET application exposes a Web API, it is likely that external clients (like front-end applications hosted on different domains) will need to consume the API.
- CORS enables these external applications to interact with your API securely.

3. To Enhance Security

- By configuring CORS, you can control which domains, HTTP methods, and headers are permitted to interact with your resources.
- This reduces the risk of unauthorized access and ensures only trusted clients are allowed.

4. To Support Modern Web Applications

- Single Page Applications (SPAs), mobile apps, and third-party integrations often require access to APIs hosted on different domains.
- CORS makes it possible for these applications to function seamlessly.

5. To Prevent Errors in Development

- Without CORS, requests to your API from a different origin would be blocked by the browser, resulting in errors like.

Access to XMLHttpRequest at 'http://example.com/api' from origin 'http://another-origin.com' has been blocked by CORS policy.

Same Origin

Two URLs have the same origin if they have identical schemes, hosts, and ports. These two URLs have the same origin:

https://example.com/foo.html
https://example.com/bar.html

Different Origins

These URLs have different origins than the previous two URLs.

https://example.net: **Different domain**

https://contoso.example.com/foo.html**: Different subdomain**

http://example.com/foo.html: **Different scheme**

https://example.com:9000/foo.html: **Different port**

What Happens Without CORS?

- Browsers block cross-origin requests, resulting in errors like.

  Access to XMLHttpRequest at '...' from origin '...' has been blocked by CORS policy.

- Your application will fail to function if it relies on APIs hosted on a different origin.

When Should You Use CORS?

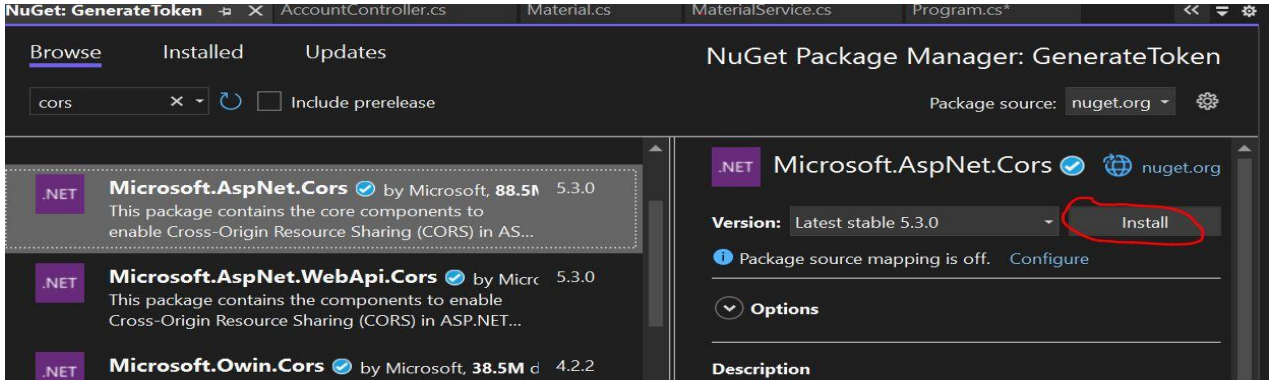
1. **Cross-Origin Frontend-Backend Communication -** Many applications have separate frontend and backend services running on different domains

   (e.g., frontend on https://app.example.com and backend API on https://api.example.com).

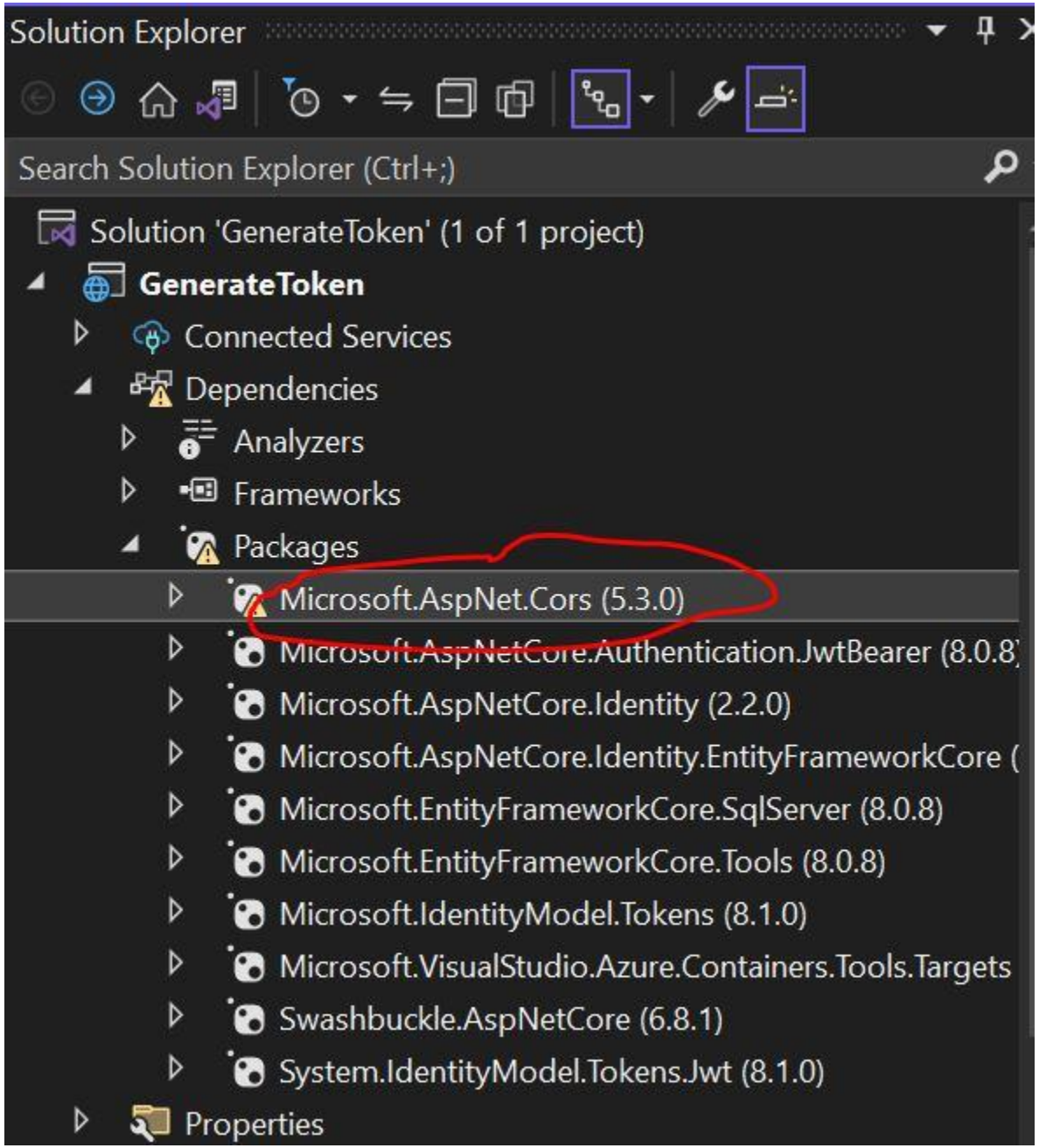
   Without CORS, the browser blocks requests between these domains. CORS enables these applications to work seamlessly.
2. **Third-Party API Consumers -** When your API is used by external clients.
3. **Microservices Architecture -** When services in a microservices setup communicate across origins.
4. **Content Delivery Networks (CDN) -** When using CDNs to serve resources like images or scripts from different origins.

Steps to Enable CORS in .NET Core

Here's how you can enable and configure CORS in a .NET Core application.



After installed Cors. We can check.



1. Install Required NuGet Package

If you are using .NET Core 2.x, ensure you have the required package:

2. Register CORS in Program.cs File in .NetCore 7

You need to add and configure CORS middleware in the service container.

```
builder.Services.AddAuthorization();

builder.Services.AddScoped<IMaterialService, MaterialService>();
builder.Services.AddSwaggerGen();

// Adding CORS services with a policy
builder.Services.AddCors(options =>
{
   options.AddPolicy("AllowSpecificOrigins", builder =>
   {
      builder.WithOrigins("https://example.com", "https://anotherdomain.com") // Allow specific domains
           .AllowAnyHeader() // Allow any header
           .AllowAnyMethod(); // Allow any HTTP method (GET, POST, etc.)
   });

   options.AddPolicy("AllowAll", builder =>
   {
      builder.AllowAnyOrigin() // Allow all domains
           .AllowAnyHeader()
           .AllowAnyMethod();
   });
});
builder.Services.AddControllersWithViews();
```

Use the CORS middleware in Program.cs file

```
app.UseRouting();
// Enable CORS globally or for specific endpoints
app.UseCors("AllowSpecificOrigins");
app.UseAuthentication();
```

3. Apply CORS to Specific Controllers or Endpoints

You can apply the CORS policy to specific controllers or actions using the **[EnableCors]** attribute.

a. For a specific controller

```
[EnableCors("AllowSpecificOrigins")]
[ApiController]
[Route("api/[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("CORS is enabled for specific origins.");
    }
}
```

b. Disable CORS for a specific action

```
[DisableCors]
[HttpGet("no-cors")]
public IActionResult NoCors()
{
    return Ok("CORS is disabled for this action.");
}
```

Testing CORS

Once you've set up CORS, you can test it by making requests to your API from a different domain and verifying that the responses are successfully received.

CORS Options

The CorsPolicyBuilder provides several methods to configure CORS behavior:

1. **Allow Specific Origins**

   builder.WithOrigins("https://example.com");

2. **Allow All Origins**

   builder.AllowAnyOrigin();

3. **Allow Specific HTTP Methods**

   builder.WithMethods("GET", "POST");

4. **Allow Specific Headers**

   builder.WithHeaders("Content-Type", "Authorization");

5. **Allow All Headers**

   builder.AllowAnyHeader();

6. **Support Credentials -** If your API requires cookies or other credentials to be included in cross-origin requests.

   builder.AllowCredentials();

Debugging CORS Issues

1. **Check Response Headers -** Ensure the Access-Control-Allow-Origin header is returned by the server.
2. **Handle Credentials Properly -** Use AllowCredentials if necessary.
3. **Configure Allowed Methods and Headers -** Ensure your API explicitly allows the methods and headers being used by the client.
4. **Browser Console -** Look for CORS errors in the browser developer tools.

Design Patterns Introduction

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

Design Patterns Introduction

Design Pattern are well-established solutions to common software design problems that developers encounter while building applications. They are not specific pieces of code but rather general reusable templates that can be adapted to solve specific problems in various contexts. Design Patterns help in designing flexible, scalable, and maintainable software.

Key Benefits of Design Patterns

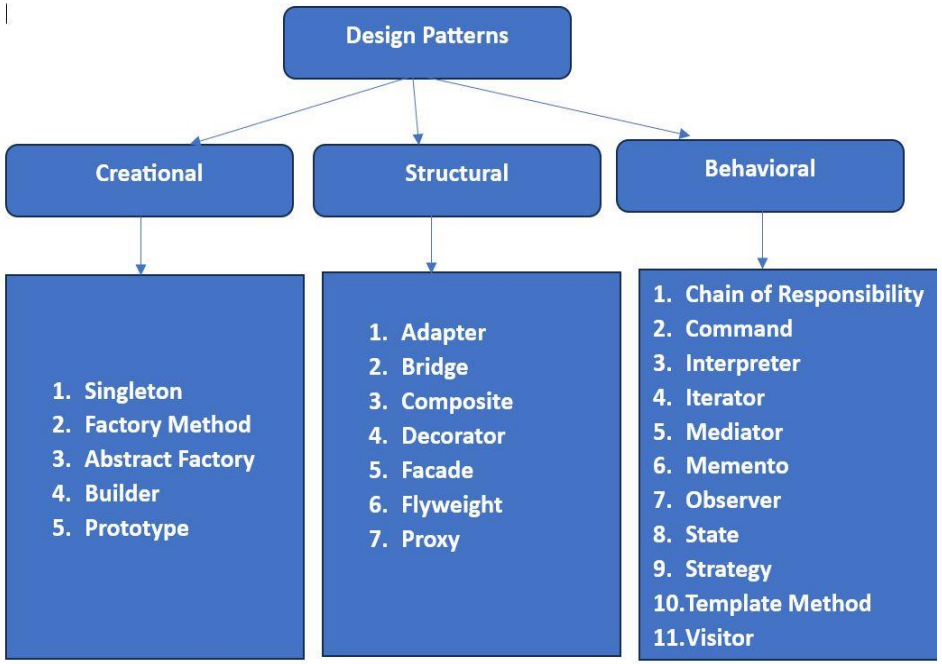
1. **Reusability -** Patterns provide proven solutions that can be reused in different projects, reducing the need to reinvent the wheel.
2. **Best Practices -** They encapsulate best practices and expertise, guiding developers toward effective design choices.
3. **Communication -** Patterns offer a shared vocabulary among developers, making it easier to communicate design ideas and solutions.
4. **Maintainability -** By promoting a structured approach to problem-solving, patterns help create systems that are easier to maintain and extend.
5. **Scalability -** They aid in designing systems that can grow and evolve without significant refactoring.
6. **Flexibility -** Patterns encourage the design of systems that are adaptable to changes in requirements or technology.

Problems Solved by Design Patterns

1. **Duplication of Code -** By using patterns like Singleton or Factory, you can avoid repetitive code and centralize logic that might otherwise be scattered.
2. **Complexity Management -** Patterns like Facade and Mediator simplify complex systems by providing a unified interface or managing communication between components.
3. **Decoupling Components -** Patterns like Dependency Injection, Adapter, and Strategy help in reducing the dependency between classes, making the system more modular.
4. **Flexibility and Extensibility -** Patterns such as Decorator and Observer allow extending the functionality of classes dynamically without modifying existing code.
5. **Maintainability -** By following patterns, developers create a more understandable and consistent codebase, which is easier to maintain and evolve.

Common Types of Design Patterns

Design Patterns are broadly categorized into three types.



1. **Creational Patterns** - These patterns deal with object creation mechanisms, trying to create objects in a manner suitable for the situation. They help manage the creation process, making it more adaptable and efficient.

   Examples - Singleton, Factory Method, Abstract Factory, Builder, Prototype

2. **Structural Patterns** - These patterns deal with the composition of classes or objects. They help in forming large structures by composing objects and classes in a way that results in more flexible and efficient design.

   Examples - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

3. **Behavioral Patterns** - These patterns deal with communication between objects and how responsibilities are distributed among them. They focus on improving communication and assigning clear responsibilities to objects.

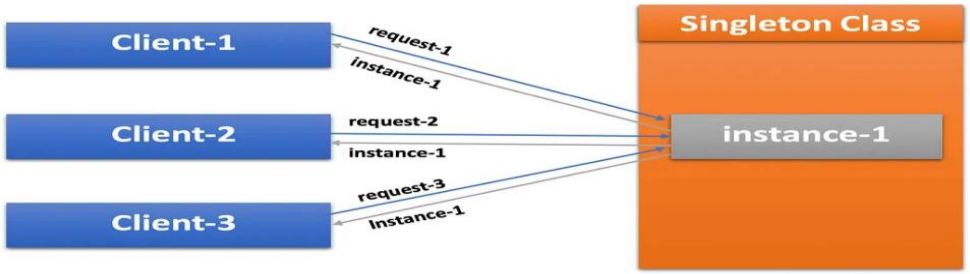
   Examples -Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

C# - Singleton Design Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. Let's explore the potential problems that arise if you don't use these design patterns in real-world scenarios.

1. Singleton Pattern

The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global point of access to that instance. If we do multiple requests to the class, will get a single copy of an instance. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.



Advantages of the Singleton Pattern

1. **Controlled Access -** Singleton ensures that there is controlled access to the instance, preventing accidental multiple instances.
2. **Reduced Memory Overhead -** Since only one instance is created, it saves memory.
3. **Consistency Across the System -** Having a single instance ensures consistent behavior across different parts of the application.

When to Use the Singleton Pattern

- **Resource Management -** When managing resources like file handles, database connections, or logging, where only one instance should be active.
- **Configuration Settings -** When an application has global configuration settings, it’s useful to have a single instance that manages these settings.
- **Cross-Cutting Concerns -** For concerns like logging or caching that are used across various parts of an application, Singleton ensures consistent behavior and state.

Problems - A logging system without Singleton

Multiple Instances - Without a Singleton, multiple instances of the logging class could be created.

This could lead to following points.

- **Inconsistent Logging -** Different parts of the application might log to different files or outputs, making it difficult to track issues.
- **Resource Contention -** Multiple instances could try to write to the same log file simultaneously, causing file access conflicts or corrupted log files.
- **Memory Waste -** Creating multiple logger instances would consume unnecessary memory.

A logging system with Singleton

You want to ensure that there's only one instance of the logger class that writes logs to a file. Multiple instances could lead to file access conflicts or inconsistent logging. Ensures that only one instance of the Logger class exists, preventing issues related to file access or inconsistent logging.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2
{

   public class Singleton
   {
      private static Singleton _instance;

      // Private constructor ensures that the class cannot be instantiated from outside
      private Singleton() { }

      public static Singleton Instance
      {
```

```
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }

    public void DoSomething()
    {
        Console.WriteLine("Singleton instance method called.");
    }
}
internal class Program
{
    static void Main(string[] args)
    {
        // Accessing the Logger instance
        Singleton logger = Singleton.Instance;

        // Usage
        logger.DoSomething();
    }
}
}
```

Example Explanation

Consider the Logger class from your previous code.

- **Private Constructor -**The constructor is private, so no one can create an instance of Logger directly using new Logger().
- **Static Instance -** A static field _instance holds the single instance of the Logger class.

- **Thread-Safe Access -** The Instance property uses a lock mechanism to ensure that only one thread can create the instance in a multithreaded environment.
- **Global Access Point -** Logger.Instance provides a global access point to the single Logger instance.

Singleton Instance Creation

```
Singleton logger = Singleton.Instance;
Singleton logger1 = Singleton.Instance;
```

1. Here, Singleton.Instance is used to access the single, globally shared instance of the Singleton class.
2. Both logger and logger1 are references to the same instance of the Singleton class.
3. The Instance property typically includes logic to check if an instance already exists. If it doesn't, it creates one; if it does, it returns the existing instance.

Singleton Instance Creation

```
logger.DoSomething();
logger1.DoSomething();
```

1. The DoSomething() method is called on both logger and logger1.
2. Since logger and logger1 refer to the same instance, calling DoSomething() on either will affect the same instance.

Lock Mechanism

Ensures thread safety, making sure that only one instance is created, even in multithreaded scenarios.

```
using System;

public sealed class Singleton
{
   private static Singleton _instance = null;
   private static readonly object _lock = new object();

   // Private constructor prevents instantiation from outside
   private Singleton()
   {
      Console.WriteLine("Singleton instance created");
   }
```

```
    // Public static method to provide access to the single instance
    public static Singleton Instance
    {
        get
        {
            lock (_lock)  // Ensures thread safety
            {
                if (_instance == null)
                {
                    _instance = new Singleton();
                }
            }
            return _instance;
        }
    }

    // Example method for the Singleton class
    public void DoSomething()
    {
        Console.WriteLine("Singleton is working!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Try to create instances
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both instances should be the same
        if (s1 == s2)
        {
            Console.WriteLine("Both instances are the same.");
```

```
    }

    s1.DoSomething();
  }
}
```

Above Code Explanation

- **Private Constructor -** Prevents external instantiation of the class.
- **Static Instance Field -** Holds the single instance of the class.
- **Lock Mechanism -** Ensures thread safety, making sure that only one instance is created, even in multithreaded scenarios.
- **Instance Property -** Provides global access to the instance

Real-World Example - Singleton Pattern as a Government

The government of a country perfectly fits the Singleton pattern, as there can only be one official government. It is a global point of access that identifies the group of people in charge.

Government Class (Singleton)

- The Government class is a singleton. It contains a private static field _instance and a static method Instance to provide access to this single instance.
- The constructor is private to ensure that no other part of the program can create additional government instances.

Thread Safety

- A lock object ensures that only one thread can create the government instance at a time, preventing multiple instances from being created in a multi-threaded environment.

Requests from Citizens and Agencies

- Citizen and Agency classes simulate entities that interact with the government by calling the ManageCountry() method.
- Both send different requests, and those requests are processed by the same government instance.

Global Access

- The main program demonstrates that both Citizen A and Agency B are interacting with the same government instance, fulfilling the Singleton pattern.

```
using System;

public sealed class Government
{
   // Static variable to hold the single instance of the Government class
   private static Government _instance = null;

   // Lock object for thread safety
   private static readonly object _lock = new object();

   // Private constructor ensures no other class can instantiate it
   private Government()
   {
      Console.WriteLine("Government has been created.");
   }

   // Public static method to provide global access to the instance
   public static Government Instance
   {
      get
      {
         lock (_lock)
         {
            // Create a new instance only if it doesn't exist
            if (_instance == null)
            {
               _instance = new Government();
            }
         }
         return _instance;
      }
   }

   // Method to simulate managing country policies
```

```
    public void ManageCountry(string request)
    {
        Console.WriteLine($"Processing request: {request}");
    }
}

// Simulating citizens and agencies sending requests to the Government
public class Citizen
{
    public void RequestService(string request)
    {
        Console.WriteLine("Citizen is sending request...");
        Government.Instance.ManageCountry(request);
    }
}

public class Agency
{
    public void RequestPolicyChange(string request)
    {
        Console.WriteLine("Agency is sending request...");
        Government.Instance.ManageCountry(request);
    }
}

// Main program to demonstrate the Singleton pattern
class Program
{
    static void Main(string[] args)
    {
        // Simulating citizens and agencies making requests
        Citizen citizenA = new Citizen();
        citizenA.RequestService("Request for better healthcare");

        Agency agencyB = new Agency();
```

```
    agencyB.RequestPolicyChange("Request for tax reform");

    // Showing that both Citizen A and Agency B are using the same Government instance
    if (Government.Instance == Government.Instance)
    {
        Console.WriteLine("Both Citizen A and Agency B are interacting with the same government instance.");
    }
  }
}
```

Output

```
Citizen is sending request...
Government has been created.
Processing request: Request for better healthcare
Agency is sending request...
Processing request: Request for tax reform
Both Citizen A and Agency B are interacting with the same government instance.
```

Application of Singleton Design Pattern

Some real-time techniques in which the Singleton Design Pattern can be used.

1. **Proxies for Services** - Invoking a Service API is a complex operation in an application, as we all know. The most extended process is creating the Service client to invoke the service API. If you make the Service proxy as a Singleton, your application's performance will improve.
2. **Facades** - Database connections can also be created as Singletons, which improves application performance.
3. **Logs** - Performing an I/O operation on a file in an application is an expensive operation. If you create your Logger as a Singleton, the I/O operation will perform better. A public and static method to get the reference to the new entity created by instance.
4. **Data sharing** - If you have any constant or configuration values, you can store them in Singleton so other application components can read them.
5. **Caching** - As we all know, retrieving data from a database takes time. You can avoid DB calls by caching your application's master and configuration in memory. In such cases, the Singleton class can handle caching efficiently with thread synchronization, significantly improving application performance.

Factory Design Pattern

Factory Design Pattern Introduction

The Factory Pattern is a creational design pattern that provides an interface for creating objects without specifying their exact class. The factory method encapsulates object creation, making it easier to manage and modify. This pattern allows for decoupling the code that uses the objects from the code that creates them.

Advantages of the Factory Pattern

1. **Decoupling -** It separates the object creation logic from the main business logic, reducing dependency and making the system more modular.
2. **Flexibility -** You can introduce new classes without modifying the client code. This improves scalability and maintainability.
3. **Simplifies Code -** Instead of spreading object creation logic throughout the system, it centralizes it in a factory, which makes the code more readable and manageable.
4. **Object Reusability -** It allows for reusing existing objects when applicable, reducing memory consumption.
5. **Promotes Interface-Based Programming -** Encourages the use of interfaces or abstract classes, improving code extensibility.

When to Use the Factory Pattern

1. **Complex Object Creation -** When creating objects is complex, involves multiple steps, or requires specific configuration.
2. **Switching Between Related Objects -** When a system needs to switch between different related classes dynamically.
3. **Code that Depends on Interfaces -** When working with abstractions (interfaces or abstract classes) and the specific implementation should be hidden.
4. **Needing Centralized Object Management -** When object creation needs to be managed and controlled from a central point, like in dependency injection scenarios.

Problems - A document system without FactoryScenario: A common real-world example of the Factory Design Pattern is the creation of different types of documents in a document processing application. Consider a scenario where you have a document editor application that can create different types of documents, such as Resume, Report, and Letter.

If a factory pattern is not used in a document generation system, several problems can arise.

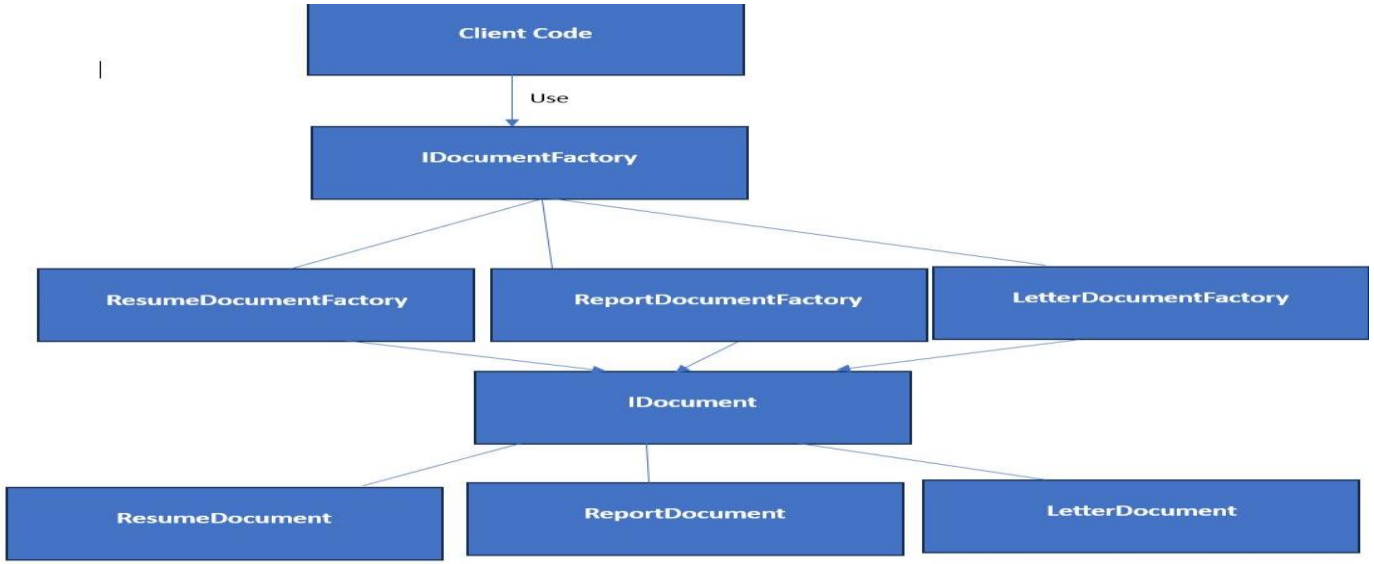
Problems

- **Tight Coupling -** Without the Factory Method pattern, the code that creates documents would need to know the exact classes (e.g. Resume, Report). This makes the system tightly coupled and harder to extend.
- **Difficulty in Adding New Document Types -** If a new document type is needed (e.g., Excel), you would have to modify the existing code to accommodate the new class. This violates the Open/Closed Principle (OCP), which states that classes should be open for extension but closed for modification.
- **Code Duplication -** The code for creating different document types would likely be duplicated in multiple places, leading to maintenance issues.

A Creation of Different Types of Documents System with Factory Pattern

A common real-world example of the Factory Design Pattern is the creation of different types of documents in a document processing application. Consider a scenario where you have a document editor application that can create different types of documents, such as Resume, Report, and Letter.

A diagram can help clarify the structure and interactions in the Factory Design Pattern. Here's a visual representation of the Factory Pattern for creating different types of documents.



How It Works

1. **Client Code** - Requests a document creation by using the factory interface (IDocumentFactory).
2. **Factory Interface** -The client interacts with the factory interface, requesting a document.

3. **Concrete Factory** - The specific factory (e.g., PdfDocumentFactory) creates and returns an instance of the corresponding document type (e.g., PdfDocument).
4. **Document Creation -** The client interacts with the IDocument interface to use the document, without needing to know the details of the document's creation.

Example

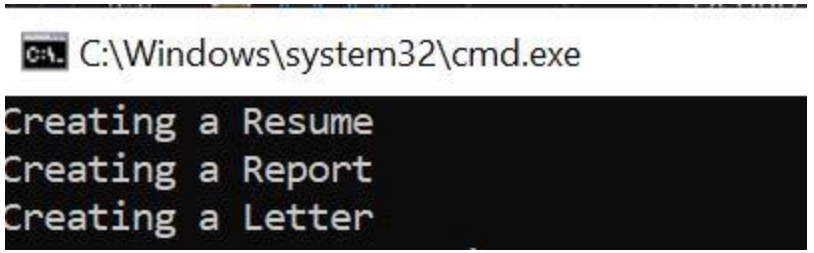
Here's the entire code example using the **Factory Pattern** in C#.

```
// Product interface
public interface IDocument
{
    void Create();
}

// Concrete Products
public class Resume : IDocument
{
    public void Create()
    {
        Console.WriteLine("Creating a Resume");
    }
}

public class Report : IDocument
{
    public void Create()
    {
        Console.WriteLine("Creating a Report");
    }
}

public class Letter : IDocument
{
    public void Create()
    {
        Console.WriteLine("Creating a Letter");
```

```
    }
}

// Factory interface
public interface IDocumentFactory
{
    IDocument CreateDocument();
}

// Concrete Factories
public class ResumeFactory : IDocumentFactory
{
    public IDocument CreateDocument()
    {
        return new Resume();
    }
}

public class ReportFactory : IDocumentFactory
{
    public IDocument CreateDocument()
    {
        return new Report();
    }
}

public class LetterFactory : IDocumentFactory
{
    public IDocument CreateDocument()
    {
        return new Letter();
    }
}

class Program
```

```
{
   static void Main()
   {
      // Client code using the Factory pattern
      IDocumentFactory resumeFactory = new ResumeFactory();
      IDocument resume = resumeFactory.CreateDocument();
      resume.Create(); // Output: Creating a Resume

      IDocumentFactory reportFactory = new ReportFactory();
      IDocument report = reportFactory.CreateDocument();
      report.Create(); // Output: Creating a Report

      IDocumentFactory letterFactory = new LetterFactory();
      IDocument letter = letterFactory.CreateDocument();
      letter.Create(); // Output: Creating a Letter
   }
}
```

Output



Advantages of the Factory Pattern in this Example.

- **Decoupling -** The client code is decoupled from the specific implementations (Resume, Report, Letter). It only depends on the IDocumentFactory and IDocument interfaces.
- **Easy to Extend -** If a new document type (e.g., Invoice) needs to be added, you can simply create a new InvoiceFactory and Invoice class without modifying any existing code.
- **Scalability -** The system can easily scale with new types of documents without changing the core logic.

Real-World Example - Transport System

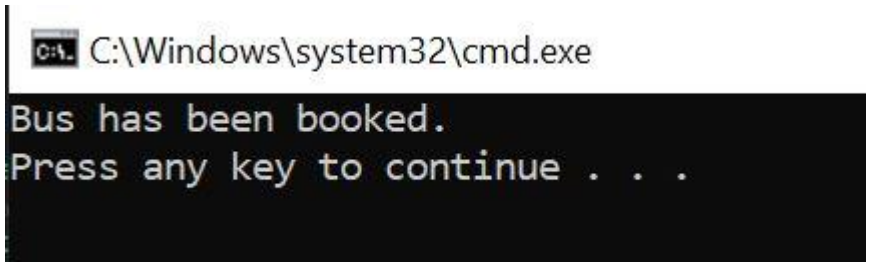
A transport booking system allows users to book various modes of transportation, such as **Car**, **Bus**, or **Bicycle**. Each type of transport has different characteristics and booking procedures, but the system should be able to handle them all through a unified interface. Using the **Factory Pattern**, the transport system can dynamically create and manage different vehicle objects.

Example

```
// Transport interface
public interface ITransport
{
    void Book();
}

// Concrete Transport Classes
public class Car : ITransport
{
    public void Book()
    {
        Console.WriteLine("Car has been booked.");
    }
}

public class Bus : ITransport
{
    public void Book()
    {
        Console.WriteLine("Bus has been booked.");
    }
}

public class Bicycle : ITransport
{
    public void Book()
    {
        Console.WriteLine("Bicycle has been booked.");
    }
}

// Factory interface
public interface ITransportFactory
{
    ITransport CreateTransport();
}

// Concrete Factories
public class CarFactory : ITransportFactory
{
    public ITransport CreateTransport()
    {
        return new Car();
    }
}

public class BusFactory : ITransportFactory
{
    public ITransport CreateTransport()
    {
        return new Bus();
    }
```

```
}

public class BicycleFactory : ITransportFactory
{
    public ITransport CreateTransport()
    {
        return new Bicycle();
    }
}

// Client code
class Program
{
    static void Main(string[] args)
    {
        // Simulating user input for transport selection
        string transportType = "Bus"; // Can be "Car", "Bus", or "Bicycle"
        ITransportFactory factory = GetTransportFactory(transportType);

        if (factory != null)
        {
            ITransport transport = factory.CreateTransport();
            transport.Book(); // Booking the transport
        }
        else
        {
            Console.WriteLine("Invalid transport type.");
        }
    }

    // Factory selection logic
    public static ITransportFactory GetTransportFactory(string transportType)
    {
        switch (transportType)
        {
            case "Car":
                return new CarFactory();
            case "Bus":
                return new BusFactory();
            case "Bicycle":
                return new BicycleFactory();
            default:
                return null;
        }
    }
}
```

Output



Advantages in a Real-World Scenario

- **Decoupling Creation Logic -** The client code doesn't need to know the specific details of how each transport type is created. It only interacts with the ITransportFactory and ITransport interfaces.
- **Extensibility -** If new transport types (e.g., **Train**, **Taxi**, or **Scooter**) are introduced, you can add new factory and transport classes without changing the existing code.
- **Flexibility -** Different transport types can be instantiated dynamically based on user input or other conditions (e.g., availability, pricing, etc.).
- **Maintenance -** The code is easier to maintain because the transport creation logic is centralized in factories. Any change in the transport creation process is isolated from the rest of the system.
- **Reusability -** The transport booking logic is reusable and can be extended to accommodate new features, such as pricing strategies, vehicle conditions, or booking rules.

Applications of the Factory Design Pattern

- **GUI Frameworks -** Creating platform-specific components (Windows, macOS, Linux) through a factory.
- **Database Access -** Switching between different databases (e.g., MySQL, SQL Server) based on configurations.
- **Logging Libraries -** Creating different types of loggers (file logger, console logger, database logger) through a factory.
- **Game Development -** Creating different types of game entities (monsters, players, NPCs) dynamically based on the game state.
- **Parser Tools -** Dynamically creating different types of file parsers (e.g., XML, JSON, CSV) using a factory.

Angular Custom Pipe

Angular Custom Pipe

In this tutorials we will see how to create angular custom pipe with an example. We will understand that with the employee's name list. suppose we have created a student list without prefix Mr. or Mis. before the name of the employee. In future we need to add name add Mr. Or Mis. prefixed before the name of the employee so we will make a custom pipe for it.

In the above image we have the list of employees when we created list of employee we have not added Mr. or Mis prefix before the name. In future there is some change in the requirement need to add prefix before the name. T do that you can simply create a custom pipe and use it.

Create an application

The following command uses the Angular CLI to create a Angular application. The application name in the following example is **angularcustompipe**.

ng new angularcustompipe

Add the following code in app.component.ts

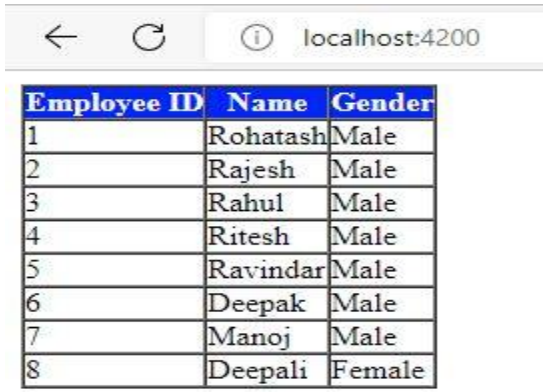
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  Employee: any[] = [
    {
       ID: '1', Name: 'Rohatash', Gender: 'Male'
    },
    {
     ID: '2', Name: 'Rajesh', Gender: 'Male'
    },
    {
     ID: '3', Name: 'Rahul', Gender: 'Male'
    },
    {
     ID: '4', Name: 'Ritesh', Gender: 'Male'
    },
    {
     ID: '5', Name: 'Ravindar', Gender: 'Male'
    },
    {
     ID: '6', Name: 'Deepak', Gender: 'Male'
    },
    {
     ID: '7', Name: 'Manoj', Gender: 'Male'
    },
    {
     ID: '8', Name: 'Deepali', Gender: 'Female'
    }
```

```
];
}
```

Add the following code in app.component.html

```
<table border="1" cellpadding="0" cellspacing="0">
<thead>
<tr style="background-color:blue; color:white">
<th>Employee ID</th>
<th>Name</th>
<th>Gender</th>

</tr>
</thead>
<tbody>
<tr *ngFor='let empobj of Employee'>
<td>{{empobj.ID }}</td>
<td>{{empobj.Name }}</td>
<td>{{empobj.Gender }}</td>
</tr>
</tbody>
</table>
```

Now run the application. It will display list of employee.



Change in Requirement

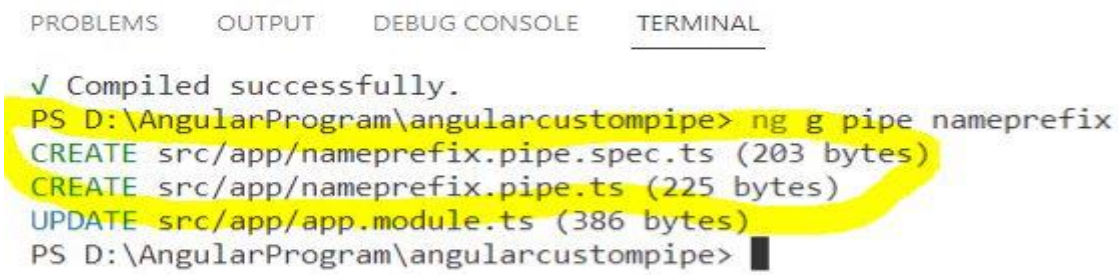
In future there is slightly change requirement, now they want to show the prefix which depend on the gender of the employee. So we need to add Mr. or Mis. prefix before the name of the employee. To do that we create a custom pipe.

Creating Angular Custom Pipe

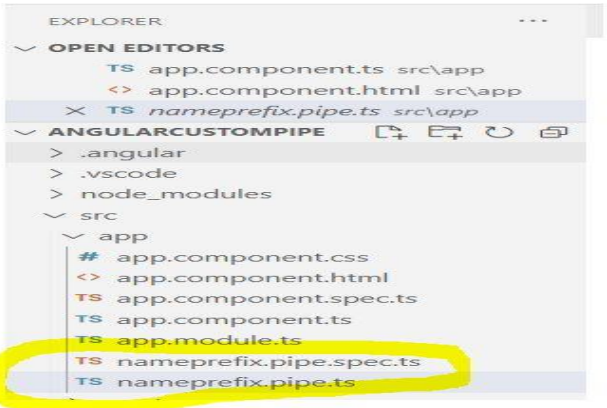
The following command uses the Angular CLI to create a Angular custom pipe. The application name in the following example is **nameprefix**.

```
ng g pipe nameprefix
```

when you type ng g pipe nameprefix and press enter, it create automatically two files which are shown in below image within the app folder. it also update the app.module.ts file.



We will see created file in the project file structure.



Now, open **nameprefix.pipe.ts** file and then copy and paste the following code in it.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'nameprefix'
})
export class NameprefixPipe implements PipeTransform {

  transform(name: string, gender: string): string {
    if (gender.toLowerCase() == "male")
       return "Mr. " + name;
    else
       return "Mis. " + name;
  }

}
```

Registering the Custom Pipe in Application Module file

When we create pipe by default it will register in **app.module.ts** file

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { NameprefixPipe } from './nameprefix.pipe';

@NgModule({
  declarations: [
    AppComponent,
    NameprefixPipe
  ],
  imports: [
    BrowserModule
```

```
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Now in the final step we add the create pipe in **app.component.html** file

```
<table border="1" cellpadding="0" cellspacing="0">
<thead>
<tr style="background-color:blue; color:white">
<th>Employee ID</th>
<th>Name</th>
<th>Gender</th>

</tr>
</thead>
<tbody>
<tr *ngFor='let empobj of Employee'>
<td>{{empobj.ID }}</td>
<td>{{empobj.Name| nameprefix:empobj.Gender}}</td>
<td>{{empobj.Gender }}</td>
</tr>
</tbody>
</table>
```

Now run the application and validate with prefix. It will work fine.

Subject vs BehaviorSubject in RxJS

Both are **types of Observables** that can **emit data to multiple subscribers**, but they differ in **how they store and emit the data**.

1. Subject

A **Subject** is both an **Observable** and an **Observer**.
It **does not store** any previous value — it only emits **new values** to **active subscribers**.

Key Points

- Does **not have an initial value**.
- New subscribers **don't get previous values** — only future emissions.
- Used when you want to **broadcast data** to multiple subscribers.

Example — Subject

```
import { Subject } from 'rxjs';
const subject = new Subject<string>();
subject.subscribe(val => console.log('Subscriber 1:', val));
subject.next('Hello');
subject.next('Angular');
subject.subscribe(val => console.log('Subscriber 2:', val));
subject.next('RxJS');
```

Output

```
Subscriber 1: Hello
Subscriber 1: Angular
Subscriber 1: RxJS
Subscriber 2: RxJS
```

Notice:

“Subscriber 2” missed the previous values (Hello, Angular).
It only received RxJS — the value emitted **after** it subscribed.

2. BehaviorSubject

A **BehaviorSubject** is a **special type of Subject** that:

- Requires an **initial value** when created.
- Always **stores and replays the latest value** to new subscribers immediately.

Key Points

- Has an **initial (default)** value.
- Always emits the **last emitted value** to new subscribers.
- Ideal for **state management** or **data sharing** between components/services.

Example - BehaviorSubject

```
import { BehaviorSubject } from 'rxjs';
const behaviorSubject = new BehaviorSubject<string>('Initial Value');
behaviorSubject.subscribe(val => console.log('Subscriber 1:', val));
behaviorSubject.next('First Update');
behaviorSubject.next('Second Update');
behaviorSubject.subscribe(val => console.log('Subscriber 2:', val));
behaviorSubject.next('Third Update');
```

Output

```
Subscriber 1: Initial Value
Subscriber 1: First Update
Subscriber 1: Second Update
Subscriber 2: Second Update ✅ (got last value immediately)
```

```
Subscriber 1: Third Update
Subscriber 2: Third Update
```

Notice

Subscriber 2 **instantly received the last emitted value** (Second Update) even though it subscribed later.

Key Differences

Feature	Subject	BehaviorSubject
Initial Value	❌ No	Required
Stores Last Value	❌ No	Yes
When Subscribed Later	Gets only new values	Gets last + new values
Common Use Case	Event emitters, user actions	Data/state sharing, user info
Number of values emitted	Only future	Last + future
Best For	Broadcasting	State management