

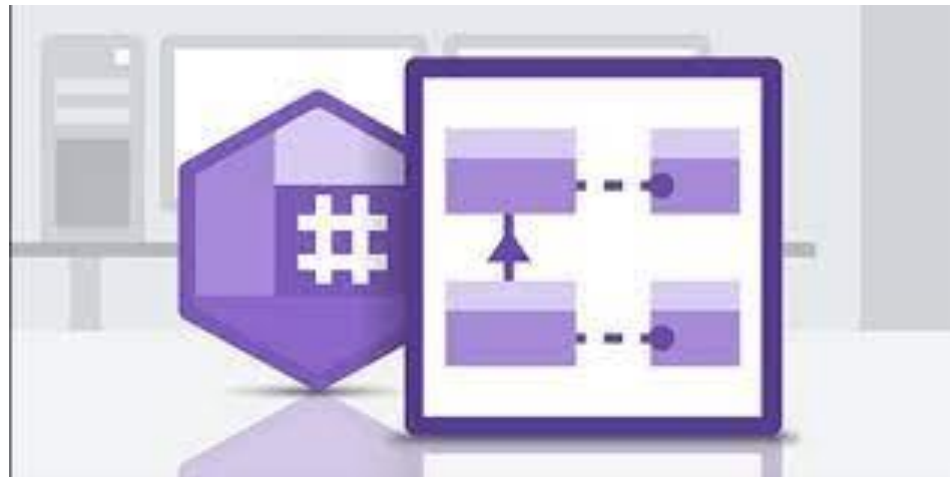
C# 14 – Các phương thức và kiểu nâng cao

Giảng viên: **ThS. Lê Thiện Nhật Quang**

Email: quangln.dotnet.vn@gmail.com

Website: <http://dotnet.edu.vn>

Điện thoại: **0868.917.786**



MỤC TIÊU

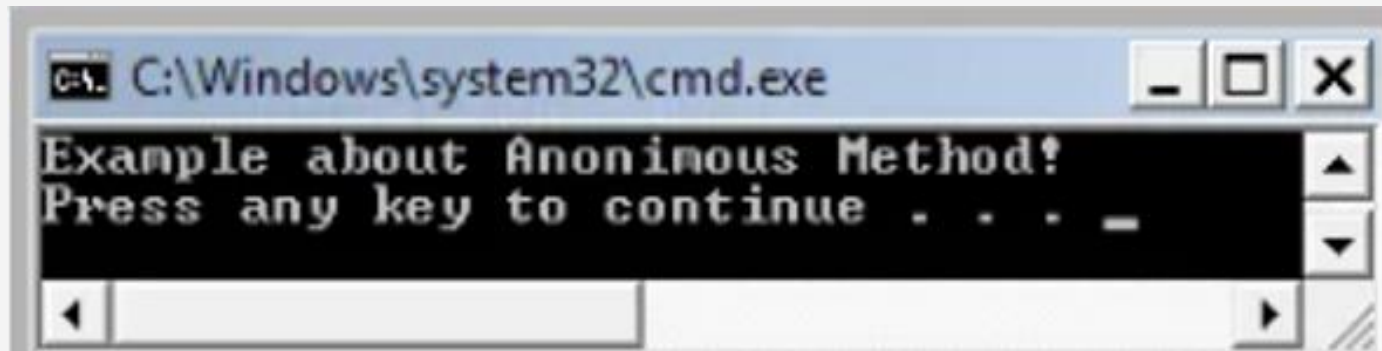
- Tìm hiểu các phương thức nặc danh - anonymous
- Định nghĩa phương thức mở rộng - extension
- Giải thích các kiểu nặc danh – anonymous
- Giải thích các kiểu partial
- Giải thích các kiểu nullable

1.1. PHƯƠNG THỨC KHÔNG TÊN

- Phương thức không tên (Anonymous method) là một khối mã lệnh không có tên và có thể được truyền như là một tham số delegate.
- Thông thường thì delegate có thể gọi một hoặc nhiều phương thức có tên và được áp dụng trong khi khai báo delegate.
- Trước khi có phương thức không tên, nếu ta muốn truyền một khối lệnh nhỏ tới một delegate thì ta luôn phải tạo một phương thức chứa khối lệnh nhỏ đó và truyền phương thức đó tới delegate; nhưng khi áp dụng phương thức không tên thì đơn giản ta chỉ cần truyền đi khối mã lệnh bên trong tới delegate mà không phải đặt nó vào một phương thức có tên cụ thể (tạo phương thức thực sự).

1.2. VÍ DỤ PHƯƠNG THỨC KHÔNG TÊN

```
class Demo
{
    static void Main(string[] args)
    {
        System.Threading.Thread th = new System.Threading.Thread(
            delegate ()
            {
                Console.WriteLine("Example about Anonymous Method!");
            });
        th.Start();
    }
}
```



1.3. ĐẶC ĐIỂM CỦA PHƯƠNG THỨC KHÔNG TÊN

Phương thức không tên chỉ có thể được sử dụng thay cho phương thức có tên nếu phương thức không tên đó được gọi thông qua một delegate.

Mỗi phương thức không tên đều có những đặc điểm sau đây:

- Xuất hiện như một khối mã lệnh nội tuyến trong khai báo delegate.
- Là lựa chọn tốt nhất cho những khối mã lệnh nhỏ.
- Chấp nhận tham số với kiểu bất kỳ.
- Sử dụng được cả những tham số có kiểu generic
- Không được sử dụng những câu lệnh nhảy như goto, break, ... để chuyển quyền điều khiển ra ngoài phạm vi của phương thức.

1.4. TẠO PHƯƠNG THỨC KHÔNG TÊN

Phương thức không tên được tạo khi bạn khởi tạo hoặc tham chiếu đến một delegate với một khối lệnh không tên. Những điều sau đây ta cần lưu ý khi tạo phương thức không tên:

- Ta có thể tạo phương thức không tên nằm bên ngoài hoặc bên trong phương thức có tên.
- Một khi đã xuất hiện từ khóa delegate trong thân của một phương thức thì bắt buộc delegate đó phải được theo sau bởi một khối mã lệnh không tên.
- Phương thức không tên được định nghĩa phải bao gồm cặp ngoặc xoắn ({ }) khi tạo một đối tượng của delegate.
- Phương thức không tên không được có kiểu trả về.
- Phương thức không tên không được có bổ từ truy cập.

1.5. CÚ PHÁP PHƯƠNG THỨC KHÔNG TÊN

```
//Đầu tiên tạo một delegate trước
Access_modifier delegate Return_type Delegate_name(Parameters);
//Rồi sau đó mới tạo phương thức không tên
Delegate_name Delegate_instance = new Delegate_name(Parameters)
{
    Body_of_anonymous_method
};
```

- Access_modifier: Là bộ từ truy cập, bao gồm các bộ từ như public, protected, private, internal, protected internal.
- Delegate_name: Là tên của delegate do ta tự đặt (đương nhiên phải tuân theo quy tắc đặt tên trong C#).

1.6. MINH HỌA PHƯƠNG THỨC KHÔNG TÊN

```
class AnonymousMethods
```

```
{
```

```
    delegate void Display();
```

```
    //Tạo phương thức không tên nằm bên ngoài phương thức có tên
```

```
    static Display objDisplay = delegate ()
```

```
    {
```

```
        Console.WriteLine("This anonymous method is outside the named method.");
```

```
    };
```

```
    static void Main(string[] args)
```

```
    {
```

```
        //Tạo phương thức không tên nằm bên trong phương thức có tên
```

```
        Display objDisp = delegate ()
```

```
        {
```

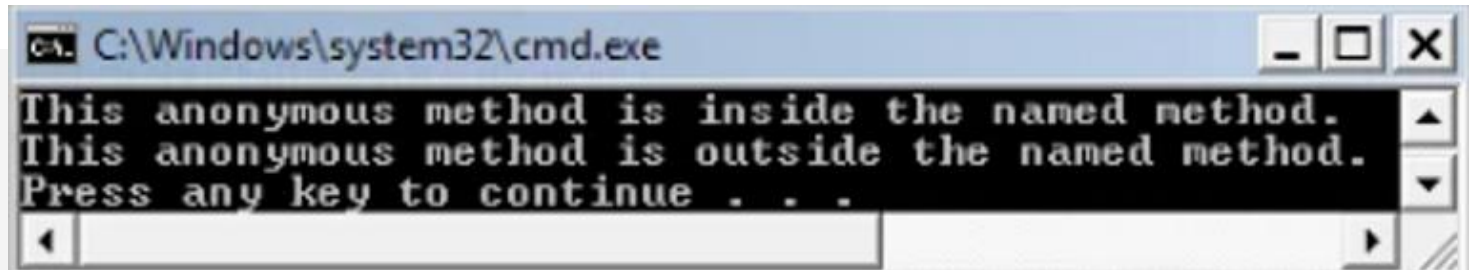
```
            Console.WriteLine("This anonymous method is inside the named method.");
```

```
        };
```

```
        objDisp();
```

```
    }
```

```
}
```

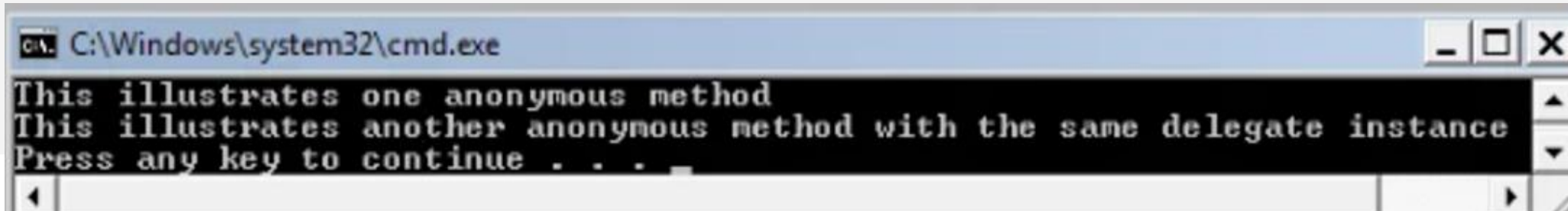


1.7. THAM CHIẾU TỚI NHIỀU PHƯƠNG THỨC KHÔNG TÊN

C# cho phép bạn tạo và khởi tạo một delegate để có thể tham chiếu tới nhiều phương thức không tên. Điều này được thực hiện bằng cách sử dụng toán tử +=. Toán tử += được dùng để bổ sung thêm các tham chiếu tới phương thức không tên hoặc có tên sau khi đã khởi tạo delegate. Ví dụ:

1.7. THAM CHIẾU TỚI NHIỀU PHƯƠNG THỨC KHÔNG TÊN (2)

```
class MultipleAnonymousMethods
{
    delegate void Display();
    static void Main(string[] args)
    {
        //delegate được khởi tạo với một tham chiếu đến phương thức không tên
        Display objDisp = delegate ()
        {
            Console.WriteLine("This illustrates one anonymous method");
        };
        //delegate được bổ sung thêm một phương thức không tên nữa
        objDisp += delegate ()
        {
            Console.WriteLine("This illustrates another anonymous method with the same delegate instance");
        };
        objDisp();
    }
}
```



```
C:\Windows\system32\cmd.exe
This illustrates one anonymous method
This illustrates another anonymous method with the same delegate instance
Press any key to continue . . .
```

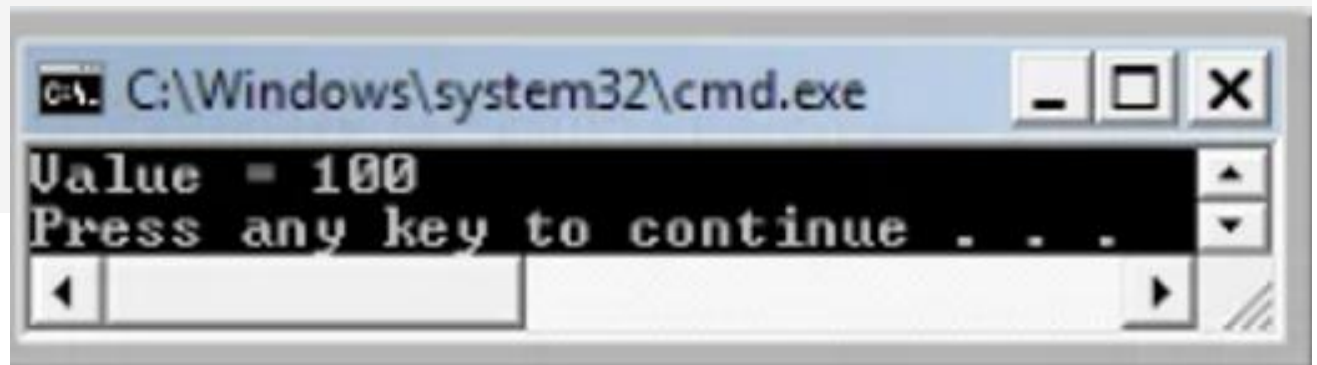
1.8. BIẾN NGOÀI (OUTER) CỦA PHƯƠNG THỨC KHÔNG TÊN

Ta có thể khai báo biến trong phương thức không tên, những biến này gọi là biến ngoài (outer). Các biến outer sẽ được 'chụp - capture' lại khi chúng được sử dụng đến; chúng sẽ được lưu trữ trong bộ nhớ cho đến khi delegate được thu dọn. Phạm vi hoạt động của các biến outer chỉ là nội trong phương thức không tên khai báo chúng.

1.9. TRUYỀN THAM SỐ

C# cho phép truyền tham số tới phương thức không tên. Kiểu của các tham số được xác định ngay tại thời điểm khai báo delegate và các tham số được định nghĩa trong cặp ngoặc tròn sau tên của delegate. Phương thức không tên có quyền truy cập tất cả các tham số này giống như phương thức có tên. Ta có thể truyền giá trị cho các tham số của phương thức không tên khi delegate được gọi. Ví dụ:

```
class Parameters
{
    delegate void Display(string str, int num);
    static void Main(string[] args)
    {
        Display objDisp = delegate (string str, int num)
        {
            Console.WriteLine(str + num);
        };
        objDisp("Value = ", 100);
    }
}
```



2.1. EXTENSION METHODS

Các phương thức mở rộng là các phương thức thêm vào lớp, cấu trúc, giao diện có sẵn mà không cần thiết phải kế thừa lớp để tạo ra các lớp mới, không cần biên dịch lại thư viện. Các phương thức mở rộng khai báo là những phương thức tĩnh, nhưng lại được gọi thông qua đối tượng lớp mà phương thức mở rộng đó khai báo.

Kỹ thuật dùng hàm mở rộng để bổ sung các tính năng cho thư viện sẵn có được dùng rất nhiều trong thư viện LINQ - để mở rộng chức năng cho các **IEnumerable**

2.2. VÍ DỤ CÁC PHƯƠNG THỨC MỞ RỘNG

```
using System.Collections.Generic;

namespace CS020_ExtensionMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> ls = new List<int>() {1,2,3,4};
            var ps = ls.Where(i => i >= 3);           // Lỗi vì List không có phương thức Where
        }
    }
}
```


2.2. VÍ DỤ CÁC PHƯƠNG THỨC MỞ RỘNG (2)

Giờ sửa lại code như sau:

```
using System.Collections.Generic;
using System.Linq;                                // Nạp thư viện LINQ

namespace CS020_ExtensionMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> ls = new List<int>() {1,2,3,4};
            var ps = ls.Where(i => i >= 3);          // Linq đã mở rộng thêm vào List phương thức Where
        }
    }
}
```

2.3. KHAI BÁO PHƯƠNG THỨC MỞ RỘNG

Hãy tạo một phương thức tĩnh sau, phương thức này in chuỗi ra màn hình với màu Console nhập vào.

```
public static void Print(string s, ConsoleColor color = ConsoleColor.Yellow)
{
    ConsoleColor lastColor = Console.ForegroundColor;
    Console.ForegroundColor = color;
    Console.WriteLine(s);
    Console.ForegroundColor = lastColor;
}
```

Đây là một phương thức tĩnh bình thường, sử dụng nó để in một chuỗi ra màn hình, chuỗi nhập ở tham số thứ nhất

```
Print("Test string ...");
```

2.3. KHAI BÁO PHƯƠNG THỨC MỞ RỘNG (2)

Giờ nếu muốn hàm Print như một hàm mở rộng của đối tượng kiểu **string**, ta sẽ thực hiện như sau:

- Chuyển khai báo **Print** trong một lớp tĩnh
 - Đảm bảo Print là phương thức tĩnh
 - Tham số đầu tiên của hàm là kiểu **string** (lớp mà phương thức mở rộng sẽ thêm vào)
- cho thêm từ khóa **this** vào phía trước kiểu này - để cho biết sẽ mở rộng lớp string với phương thức này.

2.3. KHAI BÁO PHƯƠNG THỨC MỞ RỘNG (3)

```
public static class MyExtensionMethods {  
    public static void Print(this string s, ConsoleColor color = ConsoleColor.Yellow)  
    {  
        ConsoleColor lastColor = Console.ForegroundColor;  
        Console.ForegroundColor = color;  
        Console.WriteLine(s);  
        Console.ForegroundColor = lastColor;  
    }  
}
```

Bằng cách khai báo như trên, đã mở rộng string, thêm vào nó phương thức Print, hàm này sẽ được gọi trên đối tượng lớp string.

```
string s = "Chuỗi kiểm tra";  
s.Print(); // Chuỗi kiểm tra (có màu vàng)  
"Xin chào các bạn!".Print(ConsoleColor.Red); // in ra "Xin chào các bạn!" có màu đỏ
```

Bằng cách như vậy bạn có thể mở rộng các lớp có sẵn

3.1. KIỂU NẶC DANH – VÔ DANH

Là kiểu không có tên. C# cho phép bạn tạo ra các đối tượng kiểu vô danh bằng từ khóa **new**, cú pháp cơ bản để tạo ra đối tượng có kiểu vô danh như sau:

```
var obj = new {  
    thuoctinh1 = giatri1,  
    thuoctinh2 = giatri2  
}
```

Bằng cú pháp như vậy, tạo ra được đối tượng chứa các thuộc tính (**chú ý - thuộc tính là chỉ đọc**), bạn tạo ra đối tượng mà không cần phải khai báo lớp

Ví dụ: tạo đối tượng có 3 thuộc tính

```
var myProfile = new {  
    name  = DOTNET.VN,  
    age   = 20,  
    skill = "ABC"  
};
```

3.1. KIỂU NẶC DANH – VÔ DANH (2)

Để truy cập thuộc tính của toán tử vẫn dùng ký hiệu `.` và tên thuộc tính.

Console.WriteLine(myProfile.name);

Kiểu vô danh Anonymous Type - được dùng phổ biến trong LINQ (tìm hiểu phần sau)
Khi có đối tượng kiểu vô danh, nếu dùng nó truyền như tham số cho các phương thức
- coi nó như các **object** có thể gây lỗi khi build ứng dụng - trình biên dịch kiểm tra và báo lỗi. Để giải quyết vấn đề này có thể dùng đến khai báo kiểu **dynamic**

4.1. PHƯƠNG THỨC PARTIAL

- Giả sử rằng có một tổ chức lớn trong đó bộ phận IT được chia thành hai cơ sở và Melbourne và Sydney. Nhiệm vụ bây giờ là phải thu thập dữ liệu của cả hai cơ sở.
- Giả sử ta có một lớp hoặc một structure rất lớn với rất nhiều thành phần được định nghĩa bên trong, lúc này ta cần phải tìm cách chia lớp hay cấu trúc đó thành những file nhỏ để tiện quản lý, nhưng phải đảm bảo sau khi chia nhỏ thì có thể lại kết hợp lại được khi muốn thực thi chương trình.
- Partial là cách thức mà C# cho phép ta chia nhỏ một lớp hay cấu trúc thành những phần nhỏ hơn.

4.2. ĐẶC ĐIỂM VÀ ƯU ĐIỂM CỦA PARTIAL

Ta có thể áp dụng partial cho class, structure và interface. Dưới đây là một số đặc điểm cũng như ưu điểm của partial:

- Tách mã tạo khỏi mã ứng dụng.
- Dễ phát triển và bảo trì code hơn.
- Dễ gỡ lỗi (debug) hơn.
- Ngăn ngừa lập trình viên có thể vô tình sửa đổi mã lệnh.

File 1	File 2
<pre>partial struct Sample { <MethodOne>; }</pre>	<pre>partial struct Sample { <MethodTwo>; }</pre>

Lưu ý: C# không hỗ trợ partial để định nghĩa kiểu liệt kê, tuy nhiên, ta có thể áp dụng partial cho kiểu generic.

4.3. HỢP NHẤT CÁC THÀNH PHẦN TRONG QUÁ TRÌNH BIÊN DỊCH

Sau khi dùng partial để chia nhỏ class, structure hay interface thì ta có thể đặt chúng ở những vị trí khác nhau, và ta hoàn toàn có thể hợp nhất chúng tại thời điểm biên dịch chương trình. Những thành phần được chia nhỏ có thể gồm:

- Các chú thích XML
- Các Interface
- Các tham số kiểu generic
- Các biến lớp
- Các biến cục bộ
- Các phương thức
- Các Property

4.4. PARTIAL CLASS

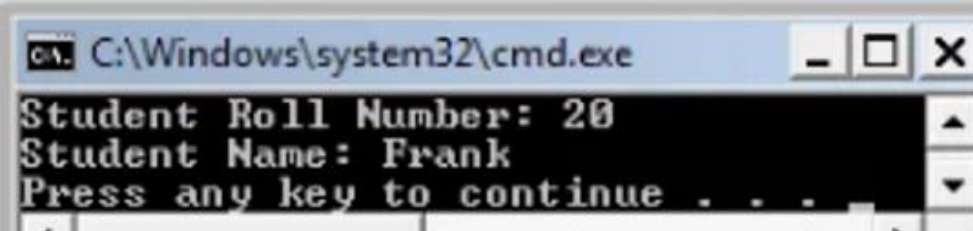
Ta có thể sử dụng partial để chia lớp thành những file nhỏ hơn, ví dụ ta muốn chia lớp thành hai file, một file chứa các thành phần private và một file chứa các thành phần public thì rõ ràng hiệu quả quản lý cũng như bảo mật sẽ tốt lên nhiều; hơn nữa điều này cũng cho phép các nhà phát triển cùng một lúc làm việc trên các thành phần khác nhau của cùng một lớp, và sẽ giúp tăng tốc độ hoàn thành.

4.5. VÍ DỤ PARTIAL CLASS

```
//Program: StudentDetails.cs
public partial class StudentDetails
{
    public void Display()
    {
        Console.WriteLine("Student Roll Number: " + _rollNo);
        Console.WriteLine("Student Name: " + _studName);
    }
}
```

```
//Program StudentDetails2.cs
public partial class StudentDetails
{
    int _rollNo;
    string _studName;
    public StudentDetails(int number, string name)
    {
        _rollNo = number;
        _studName = name;
    }
}
```

```
public class Students
{
    static void Main(string[] args)
    {
        StudentDetails objStudents = new StudentDetails(20, "Frank");
        objStudents.Display();
    }
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of a C# program: "Student Roll Number: 20", "Student Name: Frank", and "Press any key to continue . . .". The text is white on a black background.

Trong ví dụ trên, lớp StudentDetails được chia thành hai file là StudentDetails.cs và StudentDetails2.cs, trong đó StudentDetails.cs chứa phương thức Display(), còn StudentDetails2.cs chứa các biến lớp và một hàm tạo; lớp phương thức Main() tạo một thể hiện của lớp StudentDetails và gọi phương thức Display().

4.6. PHƯƠNG THỨC PARTIAL

- Xét một lớp partial có tên Shape và lớp này được tách thành hai file. Xét một phương thức có tên Create() và có một dấu hiệu trong lớp Shape.
- Lớp partial Shape chứa phần định nghĩa của phương thức Create() trong file Shape.cs. Phần còn lại của lớp partial Shape được đặt trong file RealShape.cs và nó chứa phần thực thi của phương thức Create().
- Vậy thì, phương thức Create() là phương thức partial và được định nghĩa ở hai file.
- Một phương thức partial là một phương thức mà phần khai báo của nó có kiểu partial. Phương thức có thể tùy chọn thực thi trong phần khác nhau của lớp partial hoặc kiểu hoặc cùng phần của lớp hoặc kiểu.

4.7. VÍ DỤ PHƯƠNG THỨC PARTIAL

Dưới đây là ví dụ minh họa việc khai báo và định nghĩa phương thức partial có tên Create():

Khai báo phương thức Create():

```
namespace Demo
{
    /// <summary>
    /// Lớp Shape là lớp partial và trong nó định nghĩa một phương thức partial
    /// </summary>
    public partial class Shape
    {
        partial void Create();
    }
}
```

4.7. VÍ DỤ PHƯƠNG THỨC PARTIAL (2)

Định nghĩa phương thức Create():

```
namespace Demo
{
    public partial class Shape
    {
        partial void Create()
        {
            Console.WriteLine("Creating Shape");
        }

        public void Test()
        {
            Create();
        }
    }
    class Program
    {
        static void Main(String[] args)
        {
            Shape s = new Shape();

            s.Test();
        }
    }
}
```


4.7. VÍ DỤ PHƯƠNG THỨC PARTIAL (3)

Bằng cách tách phần định nghĩa và phần thực thi hai tệp thì hai nhà phát triển có thể làm việc trên chúng hoặc thậm chí sử dụng công cụ tạo mã để tạo định nghĩa của phương thức.

Ví dụ sau thể hiện việc khai báo, định nghĩa và thực thi phương thức `Create()` trong cùng một file.

```

/// <summary>
/// Lớp Shape chứa cả phần định nghĩa và thực thi của phương thức Create().
/// </summary>
public partial class Shape
{
    partial void Create();

    partial void Create()
    {
        Console.WriteLine("Creating Shape");
    }

    public void Test()
    {
        Create();
    }
}

```

```

class Program
{
    static void Main(String[] args)
    {
        Shape s = new Shape();
        s.Test();
    }
}

```

we

Ta chỉ có thể khai báo phương thức Create() trong một phần của lớp Shape và không có phần thực thi Create() ở bất kỳ đâu. Trong trường hợp đó thì trình biên dịch sẽ loại bỏ tất cả các tham chiếu đến Create(), bao gồm bất kỳ lời gọi phương thức nào.

4.8. MỘT SỐ HẠN CHẾ KHI LÀM VIỆC VỚI PHƯƠNG THỨC PARTIAL

- Từ khóa partial là bắt buộc khi định nghĩa hoặc triển khai một phương thức partial
- Phương thức partial phải có kiểu trả về là void
- Ngầm định là private
- Phương thức partial có thể trả về ref nhưng không trả về out
- Phương thức partial không được chứa bất kỳ bộ từ truy cập nào như public, private, ... hoặc các từ khóa như virtual, abstract, sealed, ...

4.9. TÍNH HỮU ÍCH CỦA PARTIAL

- Một dự án lớn trong một tổ chức sẽ có liên quan đến việc tạo ra nhiều struct, class và interface.
- Nếu những phần này được lưu trữ trong một tệp duy nhất thì việc sửa đổi và bảo trì chúng trở nên rất khó khăn.
- Ngoài ra, nhiều lập trình viên làm việc trong dự án không thể sử dụng tệp cùng một lúc để sửa đổi.
- Do đó, kiểu partial có thể được sử dụng để chia thành các tệp riêng biệt, cho phép các lập trình viên làm việc trên chúng một cách đồng thời.

4.10. THỪA KẾ LỚP PARTIAL

- Lớp partial có thể được kế thừa giống như bất kỳ lớp nào khác trong C#.
- Nó có thể chứa các phương thức ảo được định nghĩa trong các tệp khác nhau, có thể được ghi đè trong các lớp dẫn xuất của nó.
- Ngoài ra, lớp partial có thể được khai báo như một lớp trừu tượng bằng cách sử dụng từ khóa virtual.
- Các lớp trừu tượng partial có thể được kế thừa.

4.10. VÍ DỤ THỪA KẾ LỚP PARTIAL

```
namespace Demo
{
    abstract partial class Geometry
    {
        public abstract double Area(double val);
    }
}
```

```
class Cube : Geometry
{
    public override double Area(double side)
    {
        return 6 * (side * side);
    }
    public override void Volume(double side)
    {
        Console.WriteLine("Volume of cube: " + (side * side));
    }
    static void Main(string[] args)
    {
        double number = 20.56;
        Cube objCube = new Cube();
        Console.WriteLine("Area of Cube: " +
            objCube.Area(number));
        objCube.Volume(number);
    }
}
```

5.1. NULLABLE

- Trong C# thông thường chỉ những biến có kiểu tham chiếu mới có thể được gán giá trị null.
- Tuy nhiên, C# cung cấp kiểu nullable để cho phép cả những biến có kiểu giá trị có thể được gán giá trị null.
- Biến có kiểu nullable sẽ được chỉ định hoặc là bằng cách sử dụng một giá trị đặc biệt hoặc là một biến bổ sung. Biến bổ sung này có thể chỉ định giá trị null hoặc khác null cho biến yêu cầu; còn giá trị đặc biệt chỉ hữu ích khi giá trị đã sử dụng phải có tính nhất quán giữa các ứng dụng.
- Việc tạo và quản lý các trường bổ sung cho các biến này làm tốn bộ nhớ hơn và trở nên nhàm. Vấn đề này có thể được giải quyết bằng cách sử dụng nullable.

5.2. TẠO KIỂU NULLABLE

Kiểu nullable là cách thức mà ta có thể định nghĩa giá trị null cho kiểu giá trị; điều này có nghĩa là một biến có kiểu giá trị có thể chứa giá trị null. Kiểu nullable là một thể hiện của structure `System.Nullable<T>`. Để tạo kiểu nullable ta có hai cách như sau:

```
Data_type? Variable_name = Value;
```

Hoặc:

```
System.Nullable<Data_type> Variable_name = Value;
```

```
int? n = null;
```

```
n = 123;
```

```
System.Nullable<float> f = null;
```

```
f = 456.0f;
```

5.3. CÁC ĐẶC ĐIỂM CỦA NULLABLE

- Biến có kiểu nullable có thể chứa giá trị null.
- Biến có kiểu nullable làm việc như một biến thông thường.
- Có thể trả lại giá trị được gán hoặc giá trị mặc định cho biến kiểu nullable.
- Khi đem gán một biến kiểu nullable cho một biến không phải kiểu nullable thì cần phải sử dụng toán tử (??) để thực hiện theo cú pháp như sau:

```
Variable_name_non_nullable = Variable_name_nullable??Default_value;
```

5.3. CÁC ĐẶC ĐIỂM CỦA NULLABLE (2)

```
int? n = null;  
int m = 1000;  
m = n??0; //m sẽ nhận giá trị 0 vì n chứa giá trị null
```

Lưu ý: Một trong những ứng dụng của kiểu nullable là để tích hợp C# với cơ sở dữ liệu trong đó các cột của bảng nào đó có thể chứa giá trị null. Nếu không có kiểu nullable thì không có cách nào để thể hiện chính xác những giá trị null trong các cột của bảng đó.

Một ví dụ khác cũng để thấy được ý nghĩa của kiểu nullable là nếu ta có một biến kiểu bool thì tất nhiên biến đó sẽ chứa hoặc là giá trị true hoặc là giá trị false, nhưng nếu ta lại muốn biến đó chứa giá trị không true mà cũng không false thì có chuyển kiểu của biến đó về kiểu bool?, khi đó biến có thể chứa thêm giá trị null.