# 1-Delegate (hàm ủy quyền)

## Giới thiệu Delegate

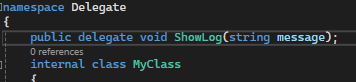
Delegate là một kiểu dữ liệu

Dùng để tham chiếu tới các hàm có tham số và kiểu trả về phù hợp với khai báo kiểu

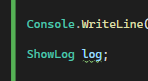
// delegate bien = phuongthuc()

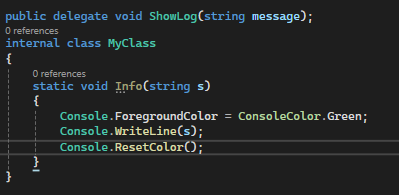
Khai báo delegate:

<access modifier> delegate <return type> <name>(<data type> parmetter);

Khai báo delegate giống như khai báo function nhưng có từ delegate và không có phần thân

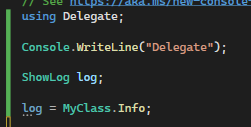
Bằng cách khai báo này ta sẽ có thêm một kiểu dữ liệu delegate mới có tên là ShowLog, kiểu ShowLog này sẽ được dùng để tạo ra những biến

Delegate là kiểu dữ liệu tham chiếu dùng để tham chiếu đến các phương thức nhưng phương thức đó phải tương đồng với khai báo của kiểu delegate (tức là phải giống nhau ở kiểu trả về và tham số truyền vào)

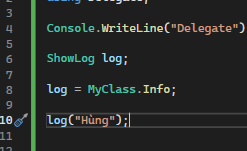
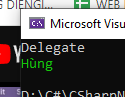
Ta sẽ tạo ra một phương thức tương đồng với phương thức ShowLog

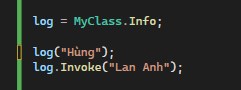
Phương thức này sẽ làm việc đó là đổi màu thông báo thành xanh sau đó hiển thị thông báo và cuối cùng là đổi màu lại như cũ

Ta thấy rằng phương thức này và kiểu delegate vừa tạo giống nhau về kiểu trả về là void và cùng nhận tham số là string. Do đó biến log có thể gán bằng phương thức Info

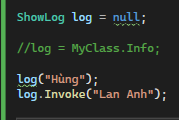
Lúc này trong biến log đang lưu trữ 1 phương thức

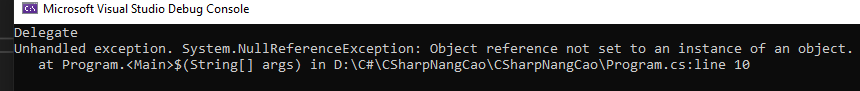
Để thi hành phương thức đang lưu trong biến delegate ta làm như sau:

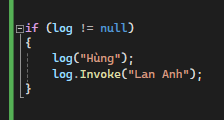
Kết quả:

Một cách nữa để thi hành phương thức đang lưu trong biến delegate

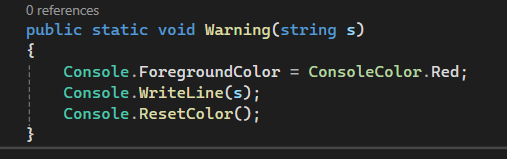
**<tên biến delegate>.Invoke(<Các tham số>)**

Biến log là 1 tham chiếu do đó nó có thể nhận giá trị null, khi mà nó nhận giá trị null mà chúng ta lại thi hành những phương thức lưu trong nó thì sẽ dẫn tới lỗi

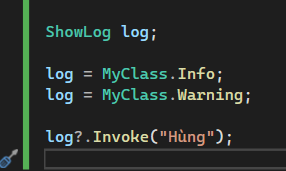


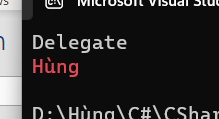
Vì vậy để an toàn trước khi thi hành những phương thức lưu trong delegate chúng ta cần phải kiểm tra biến đó khác null

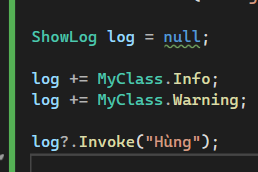
Hoặc chúng ta có thể viết ngắn gọn hơn như sau:

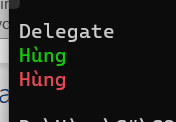
Ta sẽ thử tạo ra 1 phương thức tương đồng nữa với delegate ShowLog

Phương thức này đặt tên là Warning sẽ hiển thị ra thông báo màu đỏ

Ta gán biến delegate này cho phương thức mới tương đồng

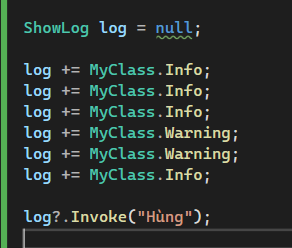
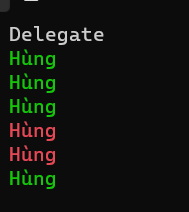
Kết quả:

Delegate có một tính chất đặc biệt là biến delegate có thể một lúc tham chiếu đến nhiều phương thức tức là có khả năng tạo ra một chuỗi những tham chiếu, để tạo ra một chuỗi các tham chiếu chúng ta sử dụng toán tử +=

Kết quả:

Mỗi phương thức khi được nối vào bằng toán tử += như một mắt xích và sẽ được gọi tuần tự

Một phương thức cũng có thể nối vào nhiều lần

## Action, Func

Đây là 2 kiểu delegate mà thư viện .NET định nghĩa sẵn, nó sử dụng tham số gereric để khai báo

Khi khai báo một biến delegate thay vì chúng ta phải khai báo một kiểu delegate cho trước



Thì chúng ta có thể sử dụng khai báo với Action và Func

### Action

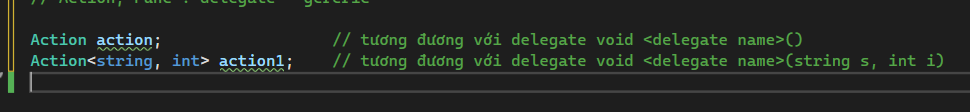
Nếu cần khai báo kiểu delegate mà không có trả về thì khai báo với Action

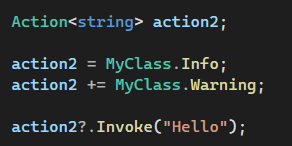
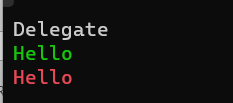
Trong trường hợp không có tham số:

Action action; // tương đương với delegate void <delegate name>()

Trường hợp có tham số:

Action<string, int> action1; // tương đương với delegate void <delegate name>(string s, int i)



 Kết quả:

### Func

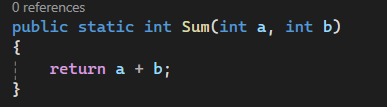
Func cũng giống như action nhưng cần phải khai báo thêm kiểu trả về:

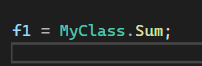
Tổng quát

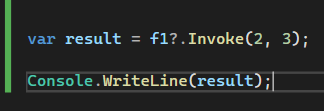
Func<data type param,..., return type> <delagate name>;

Ví dụ:

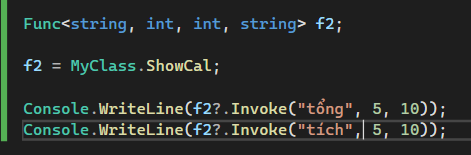
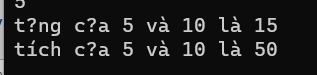
Khai báo 1 delegate có kiểu trả về là int và nhận 2 tham số cũng kiểu int

Hàm Sum tương đồng với delegate f1 vì vậy ta có thể gán f1 = Sum

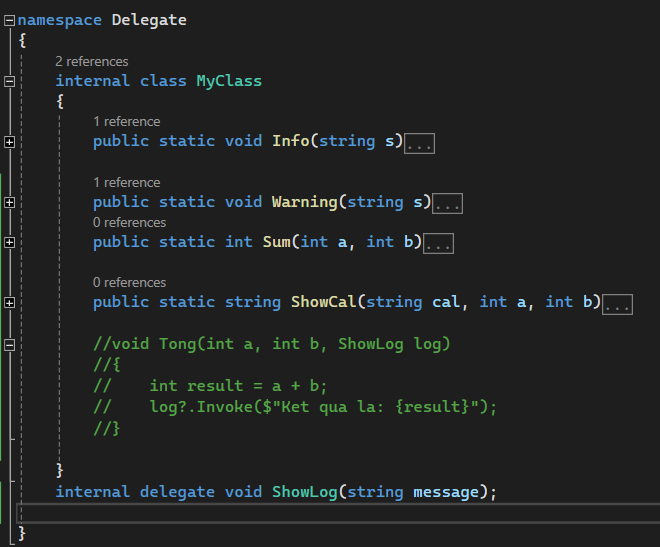
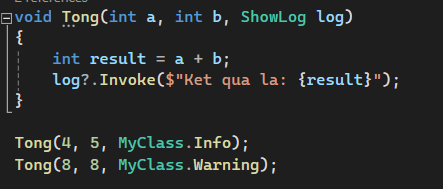
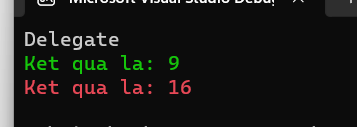




Tương tự:



## Sử dụng delegate trong tham số phương thức



# 2-Biểu thức Lambda

## Khái niệm

* Còn gọi là phương thức vô danh (anonymous function)
* Lambda là một biểu thức mà nó có thể nhận tham số đầu vào giống như phương thức sau đó thi hành những chỉ thị lệnh và có thể trả về một kết quả nào đó

## Khai báo

Gần giống như arrow fuction của js

**(tham\_so) => bieu\_thuc;**

Hoặc

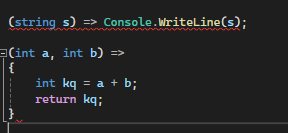
**(tham\_so) => {**

**Cac bieu\_thuc**

**Return bieu\_thuc\_tra\_ve**

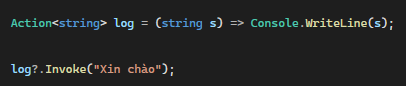
**}**

VD:

****

## Sử dụng Lambda

* Biểu thức lambda có thể được gán cho một biến kiểu delegate



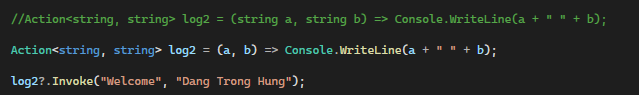
 Đây là kiểu khai báo đầy đủ

Do kiểu của param được xác định trong delegate vì vậy trong biểu thức lambda chúng ta không nhất thiết phải khai báo kiểu dữ liệu trong param vì vậy có thể rút gọn như sau



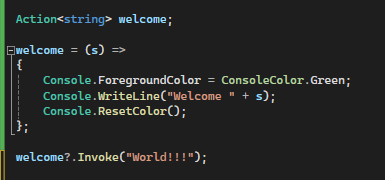
Và nếu chỉ có 1 param thì có thể bỏ cặp dấu ngoặc đơn:



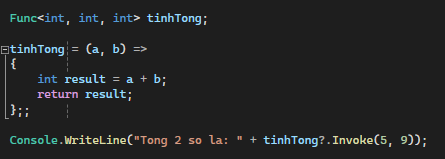
Tương tự với trường hợp có nhiều tham số: 

Tương tự với biểu thức lambda phức tạp:

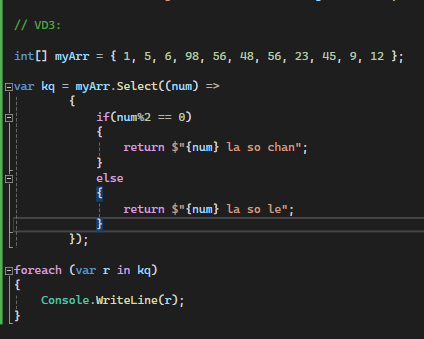
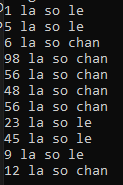
Không có kiểu trả về:



Có kiểu trả về:



Ví dụ 3:



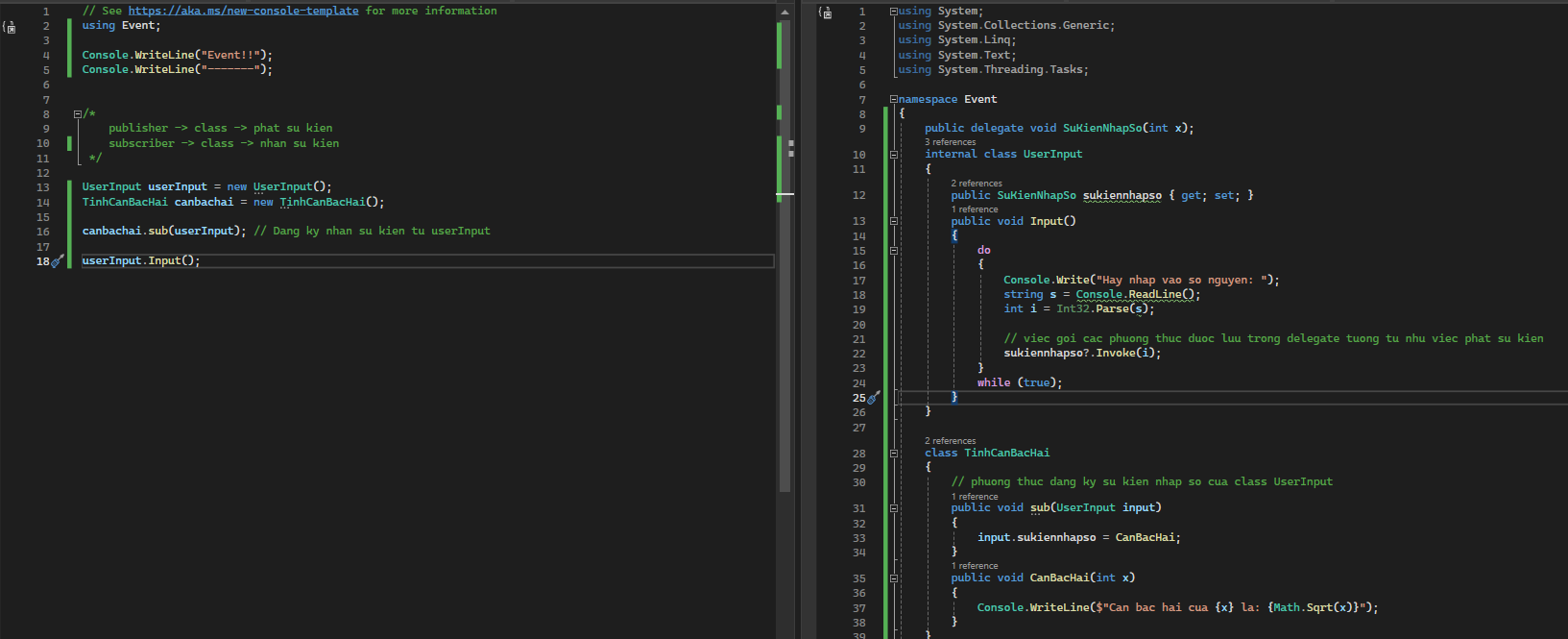
# 3-Mô hình lập trình hướng sự kiện (Event)

## Mô hình lập trình hướng sự kiện

Lập trình hướng sự kiện là việc xây dựng các class có thể phát đi những sự kiện (thông báo có một sự kiện nào đó xảy ra) những lớp này gọi là **publisher**

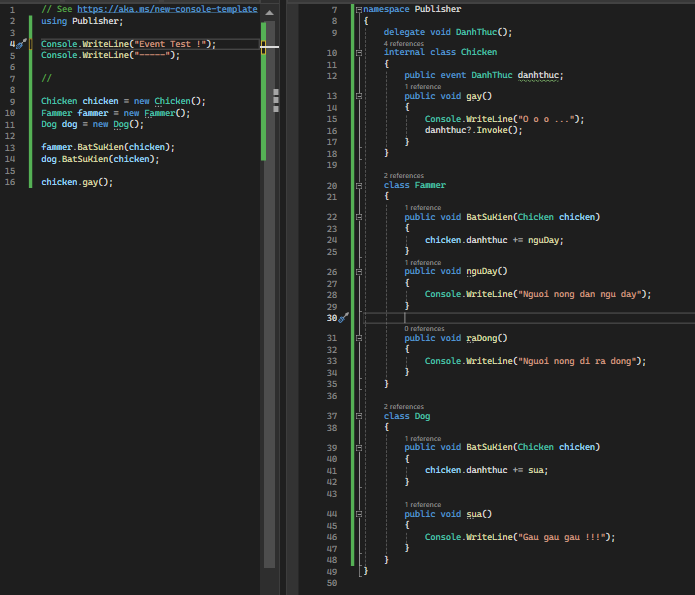
Những lớp nhận thông tin khi có sự kiện phát ra gọi là **subscriber**

## Sử dụng delegate và event

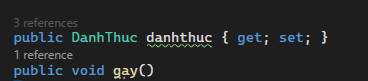


1. **Delegate SuKienNhapSo**:
   * Delegate **SuKienNhapSo** được khai báo với một tham số kiểu **int**. Delegate là một kiểu tham chiếu được sử dụng để đóng gói một phương thức có tham số đầu vào là một số nguyên và không trả về giá trị.
2. **Class UserInput**:
   * Trong class **UserInput**, có một event **sukienhapso** kiểu delegate **SuKienNhapSo**. Event này sẽ được kích hoạt khi người dùng nhập một số.
   * Phương thức **Input()** trong class **UserInput** là nơi nhập số nguyên từ người dùng và kích hoạt event **sukienhapso** bằng cách gọi **sukienhapso?.Invoke(i);** với **i** là số nguyên đã nhập.
3. **Class TinhCanBacHai**:
   * Class **TinhCanBacHai** có phương thức **CanBacHai** để tính căn bậc hai của một số.
   * Phương thức **sub(UserInput input)** của class **TinhCanBacHai** dùng để đăng ký phương thức **CanBacHai** với event **sukienhapso** của đối tượng **UserInput**. Điều này được thực hiện thông qua việc gán **input.sukienhapso += CanBacHai;**.
4. **Phần chính của chương trình**:
   * Trong hàm **Main**, một thể hiện của **UserInput** được tạo ra cùng với một thể hiện của **TinhCanBacHai**.
   * Sau đó, **TinhCanBacHai** đăng ký để lắng nghe event **sukienhapso** từ **UserInput**.
   * Cuối cùng, khi **userInput.Input();** được gọi và người dùng nhập một số, event **sukienhapso** sẽ được kích hoạt và phương thức **CanBacHai** từ **TinhCanBacHai** sẽ được thực thi, in ra căn bậc hai của số nguyên đó.

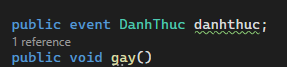
Về cơ bản, chương trình này tạo ra một cơ chế cho phép **UserInput** thông báo cho **TinhCanBacHai** mỗi khi có một số nguyên được nhập vào, và **TinhCanBacHai** sẽ xử lý số nguyên đó (trong trường hợp này là tính căn bậc hai).



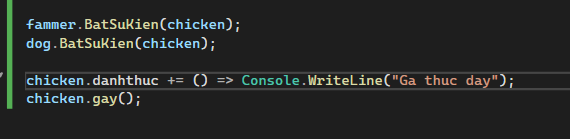
Lưu ý:

 biến delegate này giống như 1 thuộc tính vì vậy có thể get hoặc set

Nếu ta thêm từ khóa event thì biến này chỉ có thể gán += hoặc -=



Ta ccũng có thể gán trực tiếp delegate bằng biểu thức lambda như sau:



## Khai báo sự kiện với EventHandler

Trong C# cung cấp 1 delegate chuyên để khai báo các sự kiện có tên là **EventHandler**

namespace Event

{

public delegate void SuKienNhapSo(int x);

internal class UserInput

{

//public event SuKienNhapSo sukiennhapso;

public event EventHandler sukiennhapso; // delegate void KIEU(object? sender, EventArgs arg)

public void Input()

{

do

{

Console.Write("Hay nhap vao so nguyen: ");

string s = Console.ReadLine();

int i = Int32.Parse(s);

// viec goi cac phuong thuc duoc luu trong delegate tuong tu nhu viec phat su kien

sukiennhapso?.Invoke(this, new DuLieuNhap(i));

}

while (true);

}

}

class TinhCanBacHai

{

// phuong thuc dang ky su kien nhap so cua class UserInput

public void sub(UserInput input)

{

input.sukiennhapso += CanBacHai;

}

public void CanBacHai(object? sender, EventArgs e)

{

DuLieuNhap dulieunhap = (DuLieuNhap)e;

int x = dulieunhap.data;

Console.WriteLine($"Can bac hai cua {x} la: {Math.Sqrt(x)}");

}

}

class TinhBinhPhuong

{

// phuong thuc dang ky su kien nhap so cua class UserInput

public void sub(UserInput input)

{

input.sukiennhapso += BinhPhuong;

}

public void BinhPhuong(object? sender, EventArgs e)

{

DuLieuNhap dulieunhap = (DuLieuNhap)e;

int x = dulieunhap.data;

Console.WriteLine($"Binh phuong cua {x} la: {x \* x}");

}

}

class DuLieuNhap : EventArgs

{

public int data { get; set; }

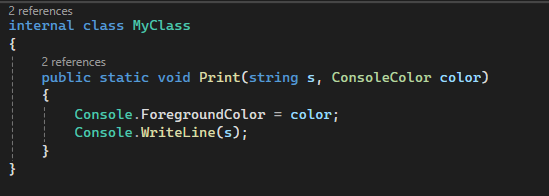
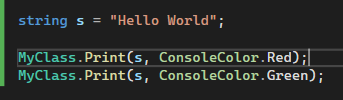
public DuLieuNhap(int x) => data = x;

}

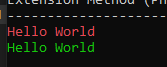
}

# 4-Phương thức mở rộng

Đây là cách thức ta mở rộng chức năng của các thư viện lớp mà không nhất thiết phải tạo ra các lớp mới kế thừa lớp có sẵn



Kết quả:

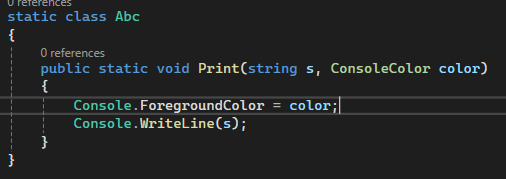


Bây giờ ta muốn viết 1 phương thức nhứ sau:

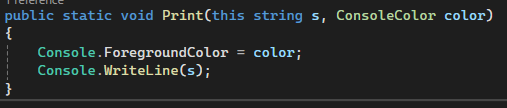


Để làm được điều này ta sẽ làm như sau:

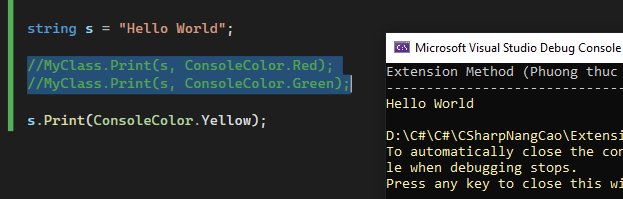
1. Đầu tiên ta viết code của các phương thức mở rộng trong 1 static class
2. Trong static class này tất cả các phương thức cũng phải là phương thức static



Tham số đầu tiền của phương thức sẽ là đối tượng của lớp cần mở rộng ở đây là s



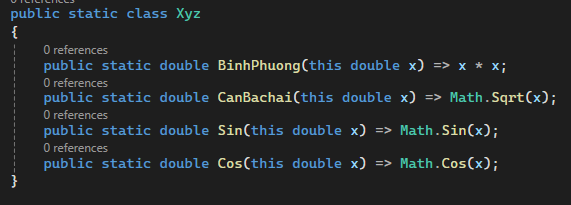
Đến đây ta đã tạo ra 1 phương thức mở rộng cho string

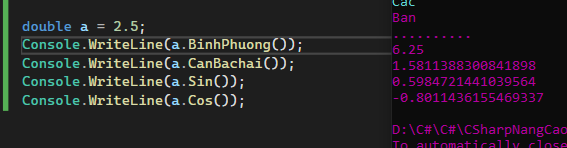


Tương tự:



Thêm 1 vài ví dụ:





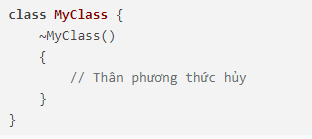
# 5-Hàm hủy, Quá tải toán tử, thành viên tĩnh của lớp, indexer

## Hàm hủy của lớp

Ở phần cơ bản về lớp, đã biết [phương thức khởi tạo](https://xuanthulab.net/su-dung-lop-thuoc-tinh-trong-c-tao-va-su-dung-doi-tuong-lop.html#Constructors) - phương thức này chạy khi tạo đối tượng lớp, ngược lại khi đối tượng bị hủy (giải phóng) nó sẽ tự động thi hành một phương thức gọi là **phương thức hủy (hàm hủy Finalizer hay Destructor)**. Dùng phương thức hủy khi có nhu cầu dọn dẹp, giải phóng tài nguyên chiếm giữ ...

*Khi sử dụng phương thức hủy lưu ý*:

* Một lớp chỉ được khai báo một phương thức hủy (trong khi có thể có nhiều phương thức tạo)
* Bạn không thể gọi phương thức hủy một cách chủ động được (do hệ thống quản lý NET CORE tự quyết định thi hành nó khi nào)
* Khai báo phương thức hủy: tên trùng tên lớp, phía trước code ký hiệu ~, phương thức không được có tham số.

****

**Khi nào hàm hủy được gọi - cơ chế dọn dẹp bộ nhớ với GC**

Trong .NET có dịch vụ hệ thống tên GC (garbage collector), nó là dịch vụ được chạy một cách tự động nhằm thu hồi bộ nhớ do .NET cấp phát trên HEAP không còn dùng đến quản lý bởi .NET. GC được thi hành tự động khi hệ thống thấy thiếu bộ nhớ - mặc dù bạn có thể yêu cấu .NET tiến hành thu hồi bộ nhớ không còn dùng đến ngày bằng cách gọi GC.Collect();

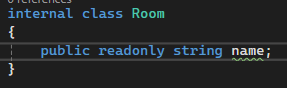
Bộ nhớ trên HEAP là nơi lưu các đối tượng được tạo ra từ lớp (toán tử new), khi đối tượng đó không còn biến nào tham chiếu (trỏ) đến thì nó sẽ được đánh dấu thu hồi - và khi GC thu hồi - **phương thức hủy** sẽ được thi hành.

## Phương thức tĩnh

Là phương thức được khai báo trong lớp với từ khóa static nhưng không thuộc về đối tượng của lớp và có thể được sử dungjt rực tiếp từ lớp mà không cần khởi tạo đối tượng

## Dữ liệu, thuộc tính tĩnh

## Thuộc tính, thành viên dữ liệu chỉ đọc



Là trường dữ liệu chỉ có thể được đọc mà không được phép gán

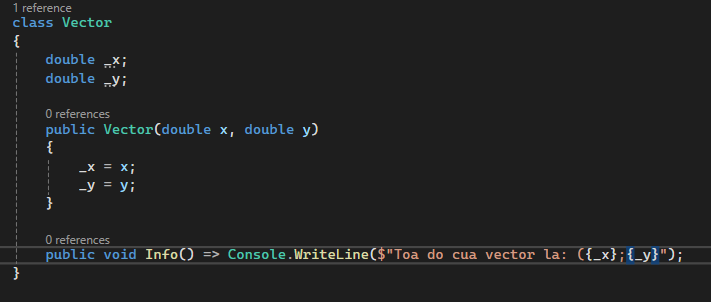
Lưu ý:

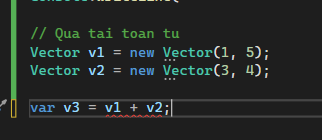
* Trường dữ liệu chỉ đọc không được phép gán giá trị nhưng có thể được gán giá trị qua phương thức khởi tạo

## Quá tải toán tử

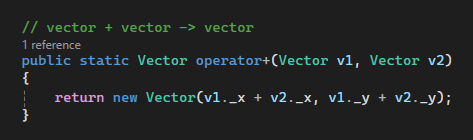
Là việc tạo ra toán tử mới cho đối tượng:

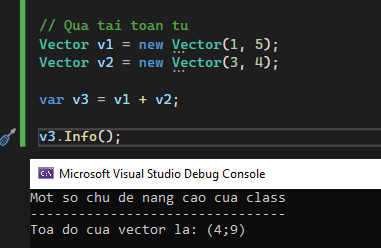
VD:



Giờ ta muốn tạo ra 1 toán tử + giữa 2 đối tượng để thực hiện cộng 2 vector trong toán học

Ta sẽ làm nhu sau:

  
kết quả:

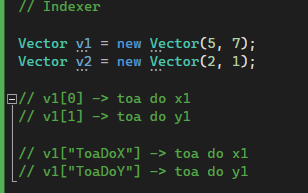


Tương tự với các toán tử khác (-, \*, /)

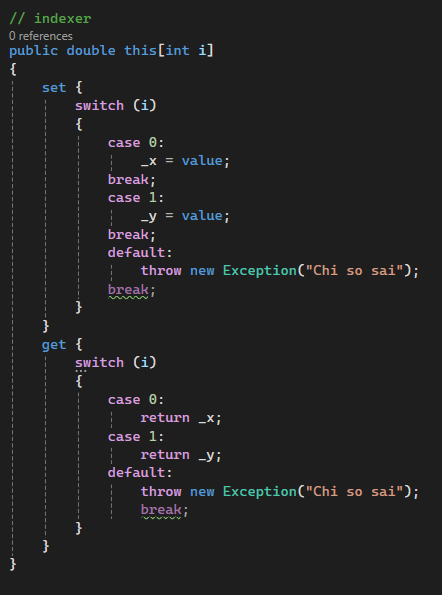
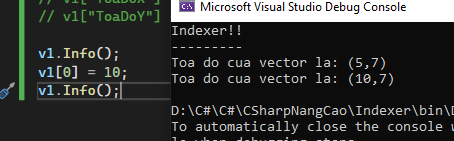
## Indexer

Là cách truy cập vào các trường dữ liệu của class

Ta sẽ đi xây dựng các indexer để truy cập tọa đọ của vector như sau:



Đầu tiên ta xây dựng 1 bộ indexer với 2 tham số 0 và 1, 0 tương ứng với x và 1 tương ứng với y

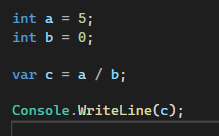
Trong class ta khai báo như sau:

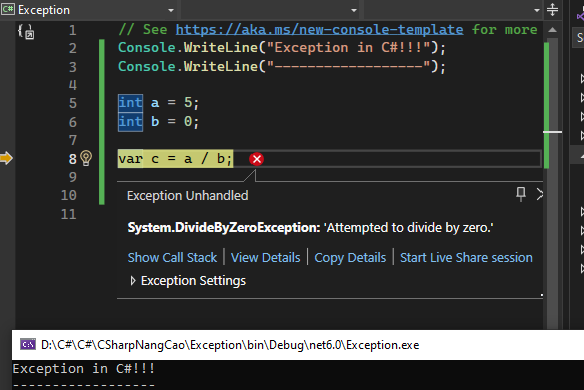
# 6-Exception

Khi thực thi một ứng dụng có thể có những lỗi không mong muốn xảy ra, với những lỗi như vậy thông thường ứng dụng sẽ kết thúc một cách đột ngột

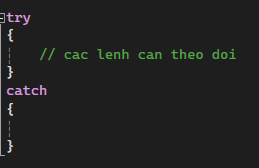
Cần có một cơ chế để giám sát những lỗi phát sinh đó, khi lỗi đó phát sinh chúng ta cần xử lý điều hướng chương trình tránh chương trình kết thúc đột ngột

Ví dụ:

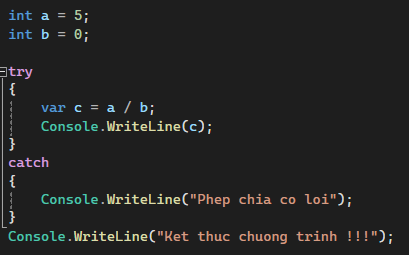


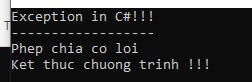
Chương trình kết thúc đột ngột nên lệnh console.WriteLine không được thực thi

## Try / Catch



Xử lý ví dụ ban đầu:

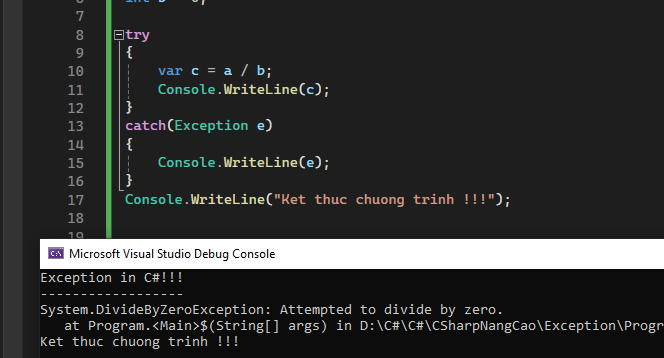
* Code trong khối try là phần có khả năng sẽ phát sinh lỗi và cần bắt lỗi,
* Khi code trong khối try phát sinh lỗi, chương trình sẽ thực hiện code ở trong khối catch

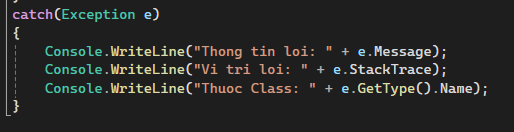


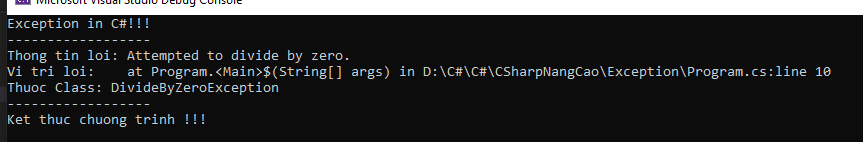
## Lớp Exception

Trong C# nếu chương trình có lỗi luôn phát sinh đối tượng lớp Exception hoặc đối tượng kế thừ từ lớp này

Đối tượng này cho biết thông tin về lỗi (tên lỗi, mô tả lỗi, nơi phát sinh lỗi,…

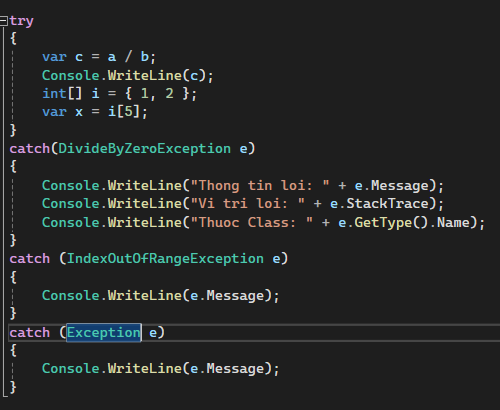




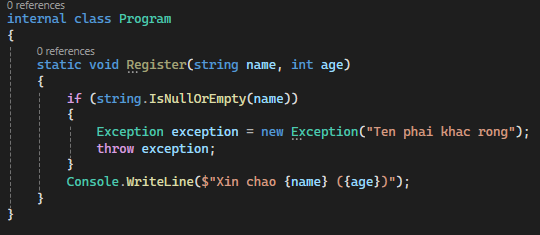


## Bắt nhiều Exception cụ thể

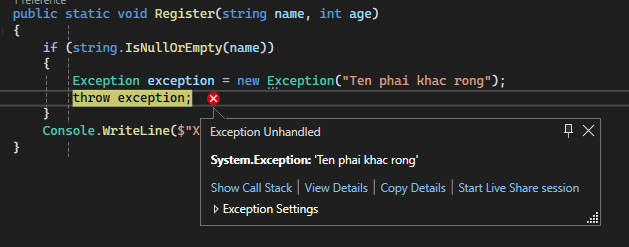
Trong thực tế code trong khối try không chỉ phát sinh ra một loại lỗi mà nó có thể phát sinh nhiều exception khác nhau



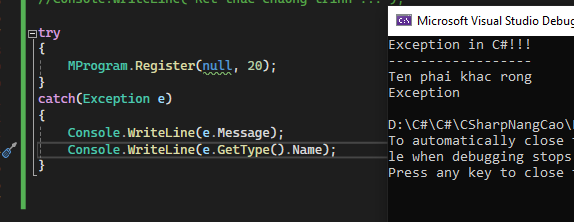
## Phát sinh các exception



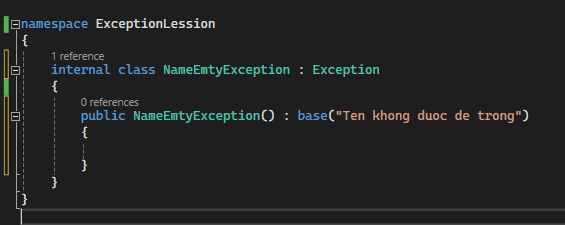
Kết quả:



Để bắt lỗi:



## Tạo exception



Sử dụng:



Kết quả:



# 7-IDisposable và từ khóa using

## Giao diện IDisposable ( IDisposable Interface)

Trong thư viện .NET đưa ra một giao diện interface có tên là IDisposable (System.IDisposable). Giao diện này chỉ có định nghĩa một phương thức:



Các lớp triển khai giao diện này chỉ việc định nghĩa nội dung phương thức này, mục đich code viết trong phương thức này là các thao tác để giải phóng các tài nguyên chiếm giữ - khi đối tượng bị hủy

**Tại sao cần tự giải phóng tài nguyên**

Như đã biết, trong .NET hầu hết các loại tài nguyên là được quản lý bởi CLR của hệ thống .NET, nên các tài nguyên, đối tượng không còn tham chiếu đến nó sẽ tự động được CLR thu hồi (GC). Đó là những tài nguyên quản lý được bởi .NET CLR

Tuy nhiên, vẫn có những loại tài nguyên mà CLR .NET không quản lý được như:

* Mở file - stream
* Các kết nối mạng, kết nối đến CSDL ...
* Những vùng bộ nhớ không quản lý được, các font chữ ...

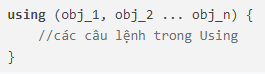
Với những loại tài nguyên này, .NET không biết tự giải phóng nó thế nào, nên bạn phải có một cơ chế chủ động làm việc này khi không còn dùng đến nữa.

Bạn cần viết code giải phóng tài nguyên thích hợp ở [phương thức hủy (Finalize)](https://xuanthulab.net/mot-so-chu-de-lap-trinh-lop-c-nang-cao.html#destructor) và có thể là triển khai giao diện IDisposable để sử dụng với câu lệnh using

## Câu lệnh using

Khi một lớp nào đó, triển khai giao diện IDisposable thì có thể dùng với using, khi đó - hết lệnh using đối tượng sẽ tự động được gọi Dispose.

Cú pháp cơ bản như sau:



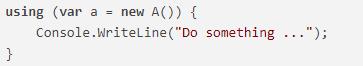
Trong đó, các obj\_1, obj\_2 ... là các đối tượng của những lớp triển khai giao diện IDisposabe

Hãy thử ví dụ sau:

Lớp A triển khai giao diện IDisposable



Sử dụng Using với lớp trên



Chạy code trên - kết quả là:





Đây chính là cách để chủ động giải phóng tài nguyên không quản lý được bởi .NET, khi tài nguyên đó không còn dùng đến nữa. Thư viện .NET có hàng trăm lớp triển khai IDisposable bạn có thể sử dụng với using như: Font, Brush, Stream, TextReader, WebResponse, Socket, DbConnection ...



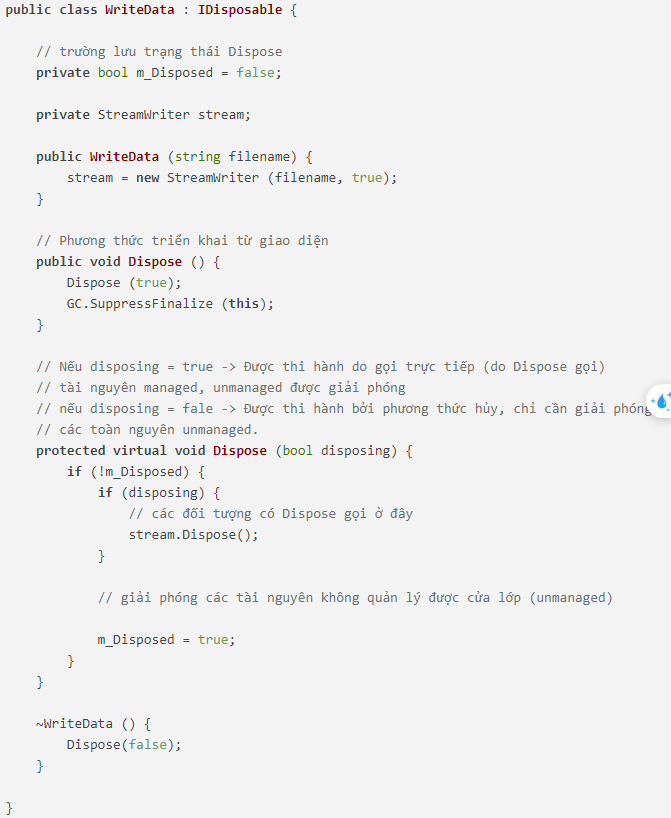
## Triển khai IDisposable cùng với hàm Hủy

Hàm hủy - phương thức hủy (còn gọi là Finalize) như trình bày ở mục [phương thức hủy (Finalize)](https://xuanthulab.net/mot-so-chu-de-lap-trinh-lop-c-nang-cao.html#destructor), nó là phương thức tự động chạy khi đối tượng không còn tham chiếu - và cũng dùng nó để viết code giải phóng tài nguyên

Vấn đề xảy ra khi một lớp vừa có hàm hủy vừa có Dispose() - tức triển khai IDisposable là: thao tác giải phóng tài nguyên có thể thực hiện hai lần - một lần khi ra khỏi using - một lần đối tượng mất tham chiếu. Hoặc khi đối tượng chủ động gọi Dispose nhiều lần. Điều này có thể dẫn đến lỗi.

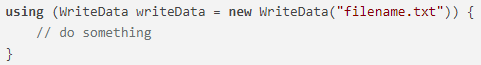
Cách giải quyết là cần có biến lưu lại trạng thái cho biết Dispose đã được thi hành hay chưa.

Ví dụ, một lớp WriteData có triển khai IDisposable có thể cần giải phóng một số tài nguyên nó chiếm giữ không quản lý được bởi .NET. WriteData cũng có sử dụng đối tượng lớp StreamWriter, đối tượng này cũng triển khai IDisposable, nên nó cần gọi thủ công Dispose() của nó khi không còn dùng đến nữa.

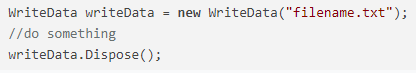


Mẫu trên bạn có thể áp dụng cho bất ký lớp nào muốn triển khai IDisposable, lúc đó bạn sử dụng using hay không using nó đều hoạt động đúng logic

Sử dụng với using, hết lệnh Dispose sẽ thi hành và mọi tài nguyên được giải phóng



Nếu không dùng using, thì chủ động gọi Dispose, tài nguyên cũng giải phóng đúng yêu cầu.



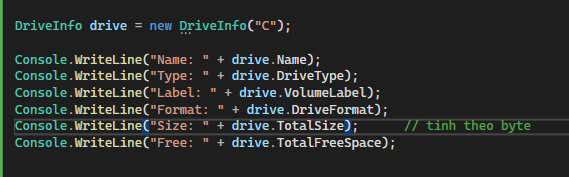
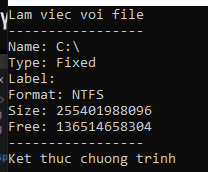
# 8-Làm việc với file

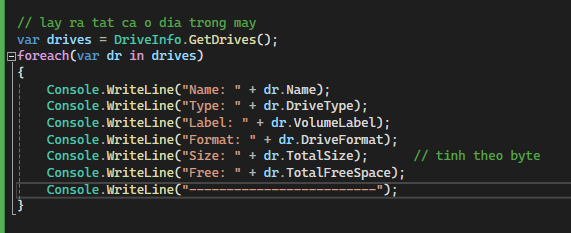
## Đọc thông tin ổ đĩa

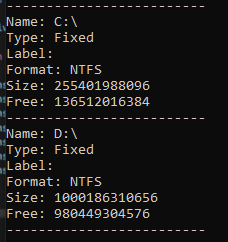
Trong .Net cung cấp 1 lớp DriveInfo cung cấp cho ta thông tin về một ổ đĩa nào đó



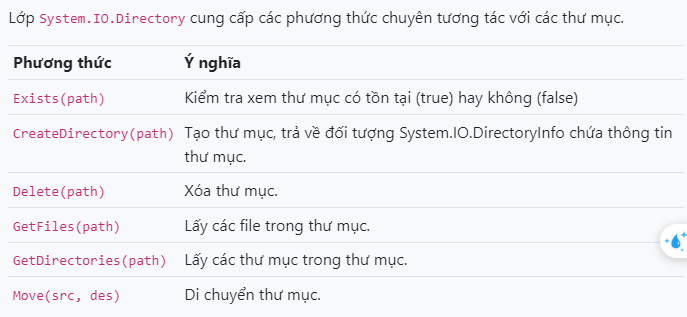
Ví dụ:



Lấy ra tất cả các ổ đĩa:

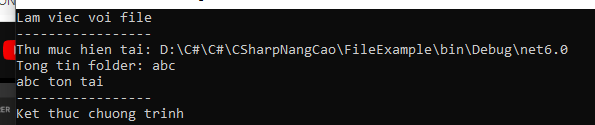
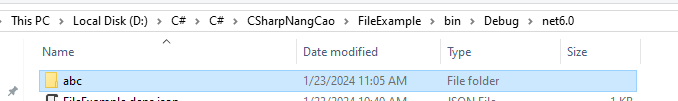


## Làm việc với thư mục ( lớp Derectory )

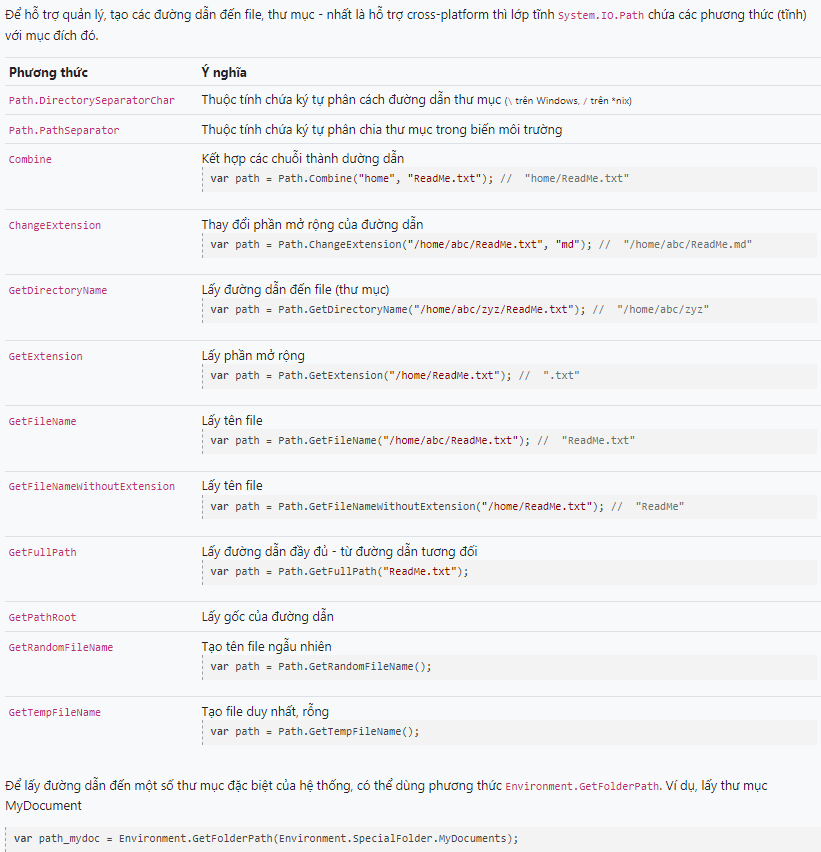




Kết quả:

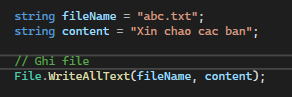
 

## Làm việc với đường dẫn (Lớp path)

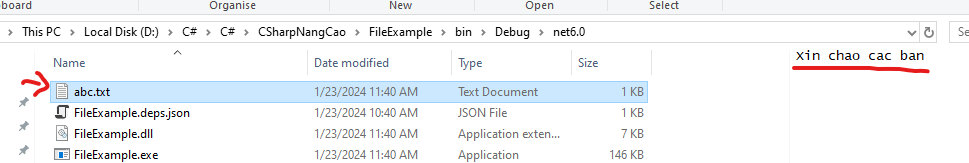


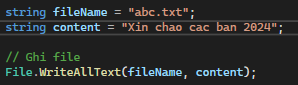
## Làm việc với file (Lớp file)

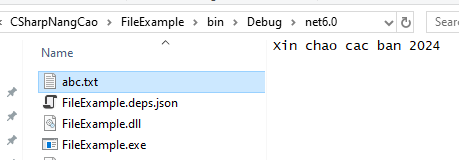
### Ghi file



Khi chạy code file abc.txt sẽ được tạo theo đường dẫn mặc định với folder:

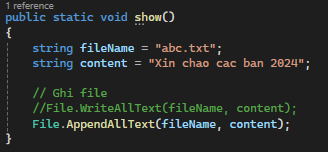
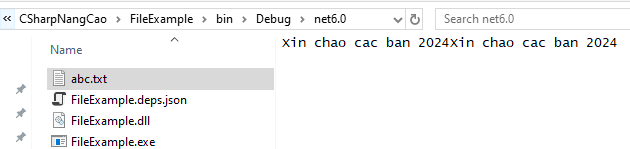


Khi ta thay đổi nội dung và chạy lại code

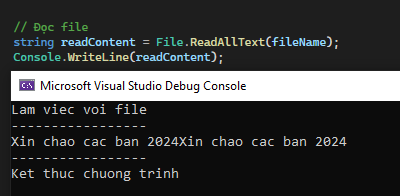


File sẽ ghi đè nội dung lên file cũ

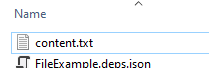
Để ghi thêm nội dung vào file có sẵn ta làm như sau:

### Đọc nội dung file



## Đổi tên file

## Copy file

File.Copy(<File cần copy>, <File mới>)

## Xóa file

File.Delete(<File cần xóa>)

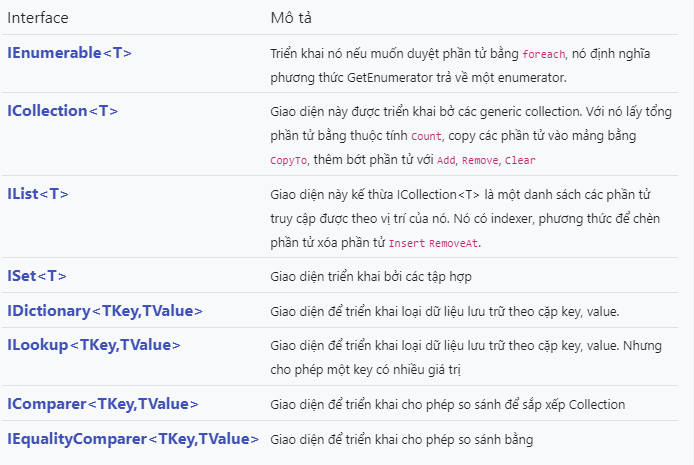
# 9-FileStream

## Stream

* Một luồng (stream) là một đối tượng được sử dụng để truyền dữ liệu.
* Khi dữ liệu truyền từ các nguồn bên ngoài vào ứng dụng ta gọi đó là đọc stream
* Khi dữ liệu truyền từ chương trình ra nguồn bên ngoài ta gọi nó là ghi stream.

# 10-Collection trong C#

Là các dữ liệu dạng tập hợp bao gồm các dữ liệu bên trong chứa tập hợp các phần tử hầu hết được triển khai(implement) từ các interface.



# 11-List

Cấu trúc dữ liệu danh sách biểu diễn tập hợp các phần tử, các phần tử trong danh sách phải cùng kiểu dữ liệu

* List khác với array là array có kích thước cố định còn List thì có kích thước động
* List có thể thu hẹp hay mở rộng về kích thước
* List nằm trong namespace **System.Collections.Generic**

Khởi tạo đối tượng List:

**List<data type> <list name> = new List<data type>();** Khởi tạo list không có phần tử ban đầu

**List<data type> <list name> = new List<data type>() {<el1>, <el2>,…};** Khởi tạo list có phần tử ban đầu

Thêm phần tử vào list:

Thêm nhiều phần tử (mảng các phần tử) vào list:

Đếm số phần tử list:

Truy cập phần tử trong list giống như tuy cập phần tử trong array:

<list>[index];

## Các phương thức trong list

|  |  |
| --- | --- |
| **Phương thức** | **Tác dụng** |
| Add(<Phần tử cần thêm>) | Thêm phần tử vào cuối danh sách |
| AddRange | Thêm một nhóm các phần tử vào cuối danh sách |
| Insert(<vị trí cần chèn>, <phần tử cần chèn>); | Chèn phần tử vào danh sách |
| InsertRange() | Chèn 1 nhóm phần tử |
| RemoveAt(<Index phần tử cần xóa>) | Xóa phần tử ở vị trí index |
| Remove(<value>) | Xóa phần tử theo giá trị phần tử (chỉ xóa phần tử đầu tiên tìm thấy) |
| Clear() | Xóa tất cả phần tử trong danh sách |
| Find(delegate) | Trả về phần tử đầu tiên tìm thấy thỏa mãn điều kiện return trong delegate |
| FindAll(delegate) | Trả về 1 tập hợp các phần tử thỏa mãn điều kiện return trong delegate |
| Sort(delegate) | Sắp xếp |

# 12-SortedList

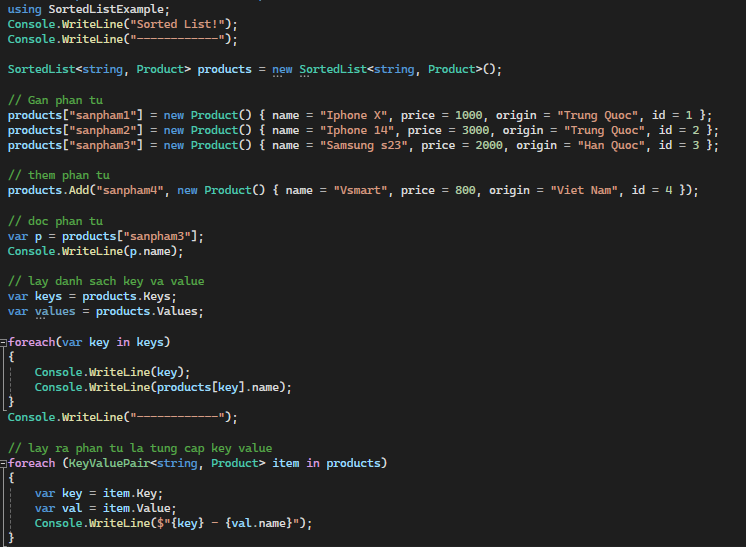
* Là một danh sách nhưng mỗi phần tử bao gồm key và value

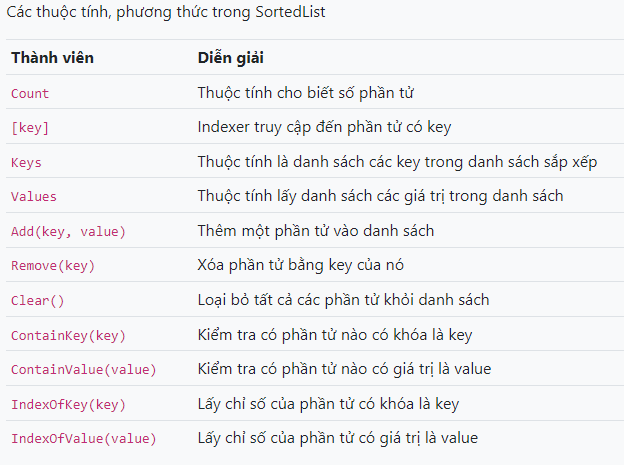
Khởi tạo:

SortedList<<key type>, <value type>> <sortedList name> = new SortedList<string, < value type >>();

Khi ta sử dụng sorted List bản chất bên trong nó 2 danh sách, 1 danh sách lưu trữ key và một danh sách lưu trữ value, 2 danh sách này liên kết lại với nhau mỗi key tương ứng với 1 giá trị lưu trữ

Việc sử dụng SortedList để truy cập vào danh sách thông qua các key nhằm tăng tốc độ truy vấn



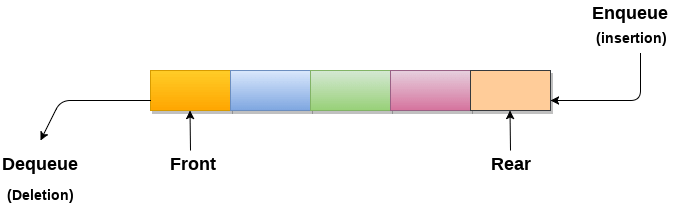


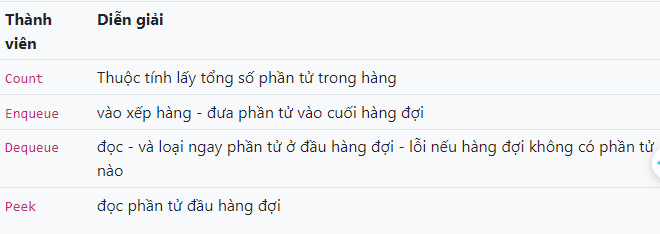
# 13-Queue and stack

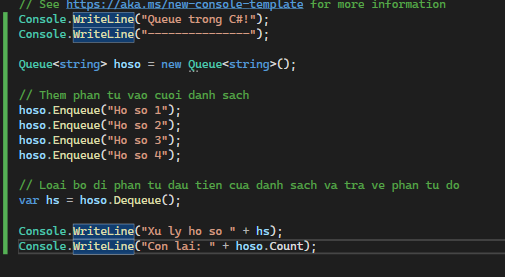
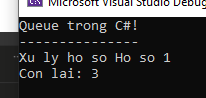
## Queue – Hàng đợi

Hàng đợi là mô hình **FIFO** (first in, first out - vào trước, ra trước hay đến trước được phục vụ trước), nó giải quyết các bài toán thực tế giống như xếp hàng mua vé máy bay ...

.NET cung cấp lớp Queue<T> để giả quyết giải thuật hàng đợi.



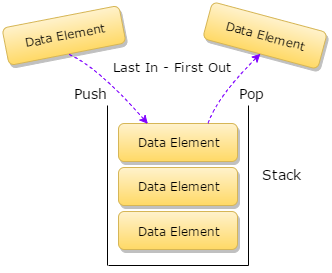


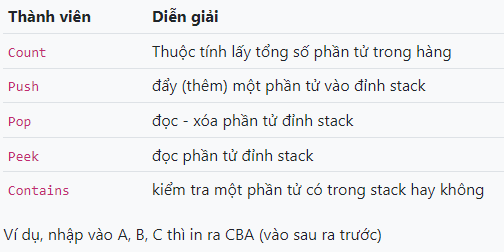
Queue gần giống như list nhưng khi thêm phần tử thì luôn thêm vào cuối danh sách còn khi xóa phần tử luôn xóa phần tử ở đầu danh sách

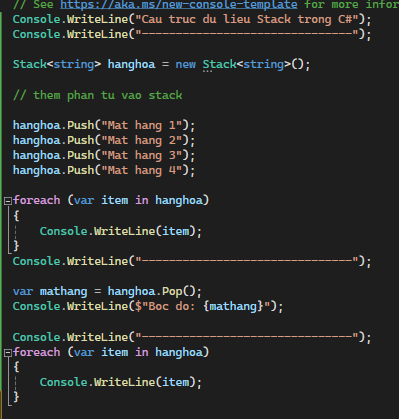
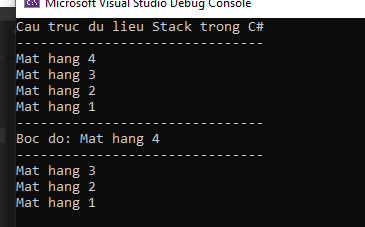
## Stack – Ngăn xếp

Ngăn xếp stack khá giống hàng đợi, nhưng khác đó là LIFO (last in, first out) - vào sau thì ra trước, nó giống như xếp hàng hóa vào các container, vào nhà kho - cái nào đưa vào sau thì khi thảo dỡ lại thực hiện đầu tiên, nó giống như xếp đĩa vào cọc đĩa CD cái nào đưa vào cọc trước sẽ được lấy ra sau ...



Trong C# với .NET nó cung cấp lớp Stack<T> để thực hiện giải thuật này.

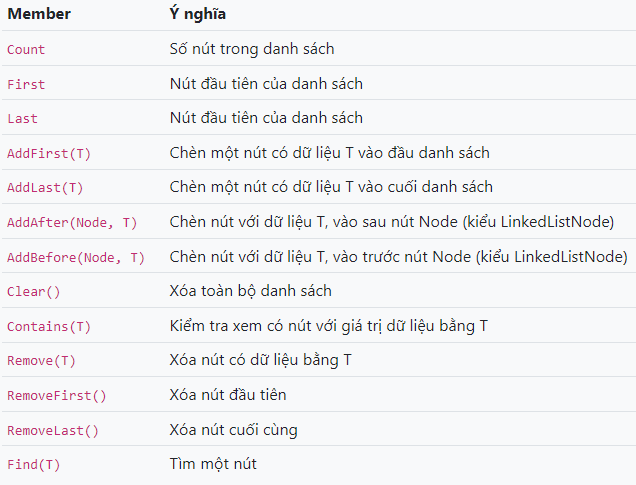




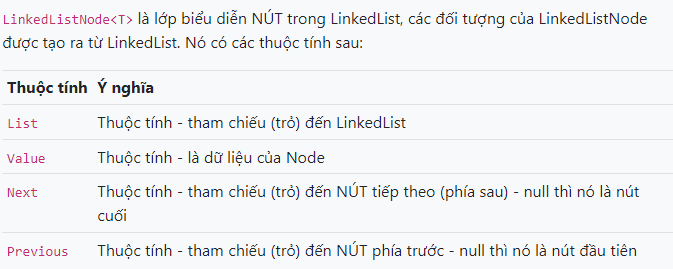
# 14-LinkedList

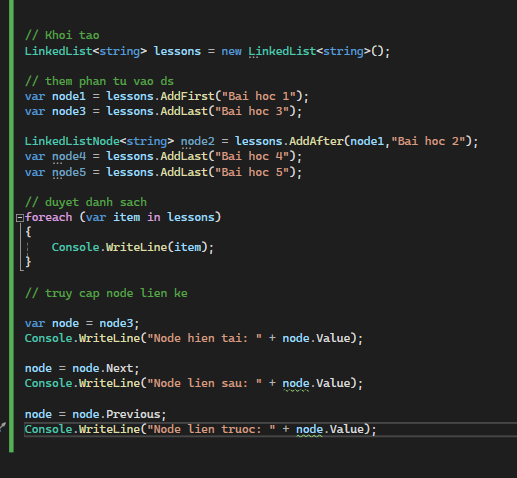
* Danh sach liên kết trong C# được triển khai bằng lớp LinkedList
* Mỗi phần tử trong danh sách gọi là 1 **node** (có kiểu dữ liệu là **LinkedListNode<T>**)

## Phương thức của LinkedList



## Phương thức của LinkedListNode



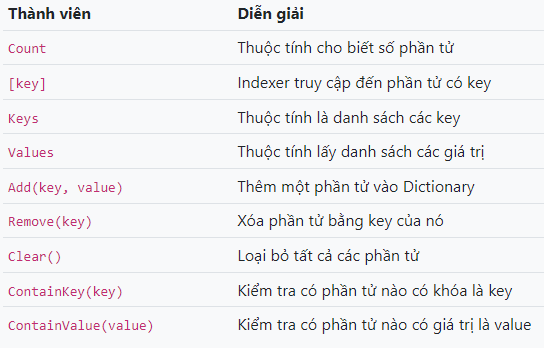


# 15-Dictionary

Lớp Dictionary<Tkey,TValue> khá giống SortedList, Dictionary được thiết kế với mục đích tăng hiệu quả với tập dữ liệu lớn, phức tạp.

Một đối tượng dữ liệu lưu vào Dictionary dưới dạng cặp key/value, truy cập đến phần tử thông qua key hoặc thông qua vị trí (index) của dữ liệu trong danh sách. Chú ý, không cho phép trùng key.

Các thuộc tính, phương thức trong Dictionary

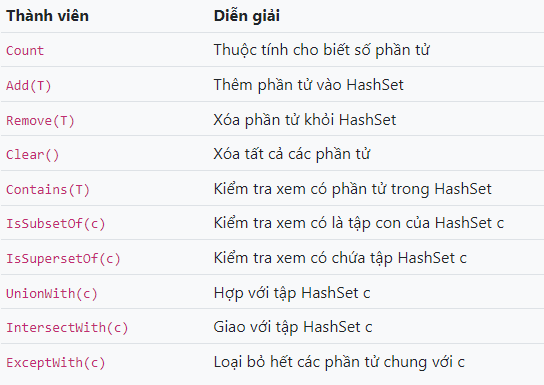


Gần giống như SortedList thường dùng cho tập dữ liệu lớn

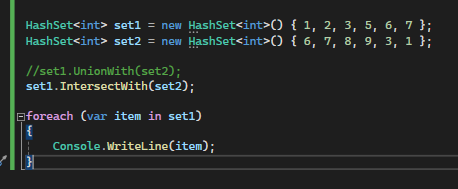
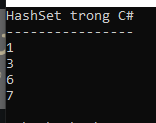
# 16-HashSet

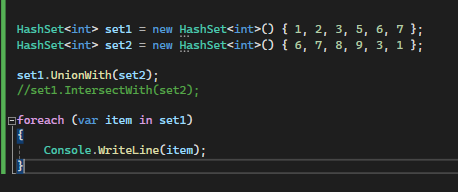
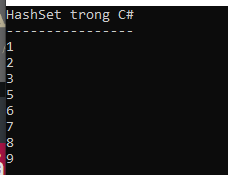
HashSet là tập hợp danh sách không cho phép trùng giá trị. HashSet<T> khác với các collection khác là nó cung cấp cơ chế đơn giản nhất để lưu các giá trị, nó không chỉ mục thứ tự và các phần tử không sắp xếp theo thứ tự nào. HashSet<T> cung cấp hiệu năng cao cho các tác vụ tìm kiếm, thêm vào, xóa bỏ ...

Một số phương thức trong HashSet



HashSet cung cấp 1 số phương thức dùng để tương tác giữa các tập hợp (Phép hợp, phép giao,…)



# 17-ObservableCollection

Lớp Generic ObservableCollection<T> là một tập hợp tương tự như List<T> ..., tức là nó mô tả một tập hợp dữ liệu có thể thay đổi số lượng bằng các phương thức quen thuộc như Add(), Remove(), Clear() ...

Nhưng với ObservableCollection<T> thì nó cung cấp thêm sự kiện thông báo nhi số lượng phần tử thay đổi như thêm, bớt ...(nghĩa là giám sát được biến động phần tử). Các sự kiện event này có tên là CollectionChanged, trong tham số mà sự kiện gửi đến, e.Action có cho biết hành động thay đổi trên tập hợp là gì (ví dụ: thêm NotifyCollectionChangedAction.Add, bớt NotifyCollectionChangedAction.Remove), clear tập hợp NotifyCollectionChangedAction.Reset ...

# 18-LINQ

* **LINQ (Language Integrated Query)**: Ngôn ngữ truy vấn tích hợp
* Đây là một thành phần của .NET được tích hợp vào C#
* Giúp chúng ta viết trong code C# những câu lệnh truy vấn tới các nguồn dữ liệu để lấy dữ liệu

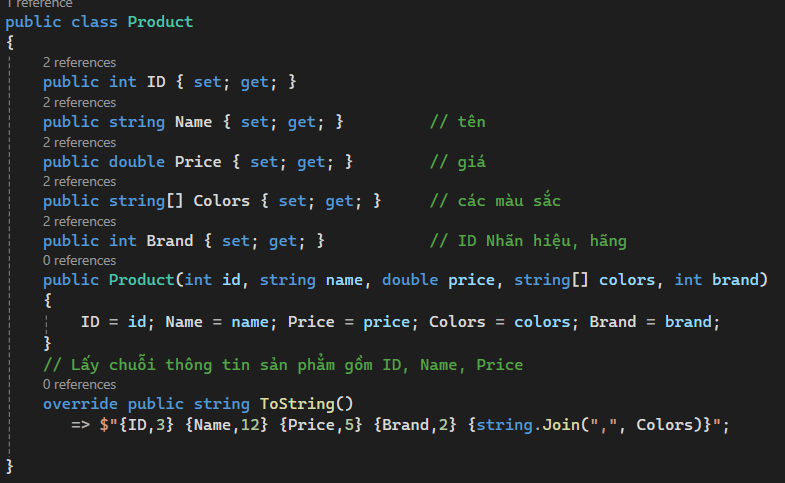
Câu truy vấn của LINQ khá giống câu truy vấn SQL

LINQ thực hiện trên những nguồn dữ liệu thường là những tập hợp phần tử triển khai từ các:

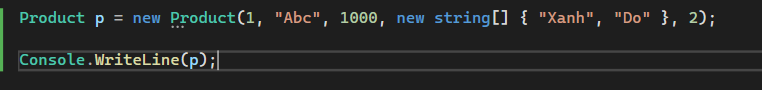
* Interface IEnumerable, IEnumerable<T> (Array, List, Stack, Queue,…)
* XML, SQL Server,… Những đối tượng này được nạp vào chương trình và được thể hiện thông qua những Interface Ienumerable,… sau đó truy vấn LINQ sẽ thực hiện trên nguồn dữ liệu này

Để sử dụng LINQ ta cần nạp thư viện **System.Linq**

Để tìm hiểu về LIQ đầu tiên ta sẽ tạo ra 1 nguồn dữ liệu: Ta sẽ ví dụ tạo ra 1 List product

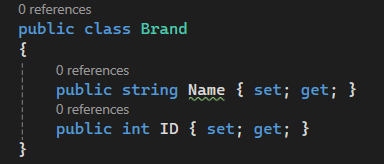
Ta có class sau: 

Khởi tạo 1 đối tượng Product:

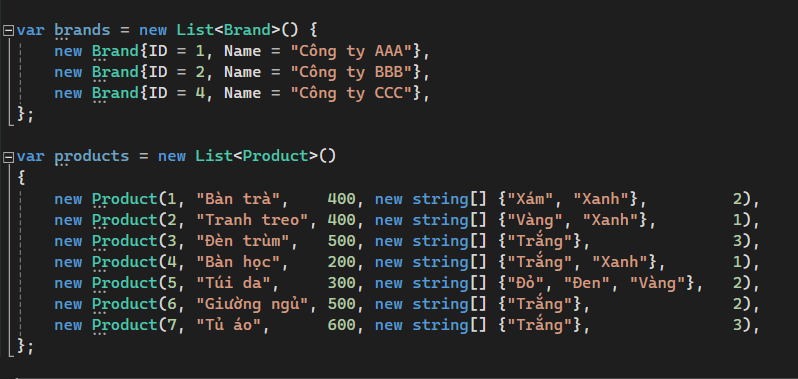




Ta có class Brand để biểu diễn nhãn hiệu:

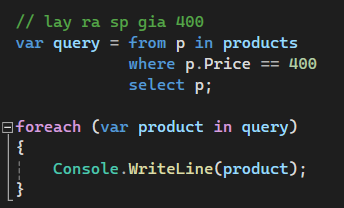


Ta sẽ tạo ra các danh sách Product và Brand mẫu:



## Viết câu truy vấn LINQ

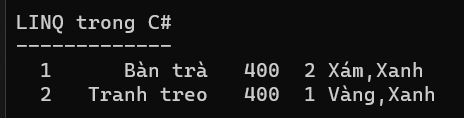
Ta sẽ thử lấy ra những sản phẩm có giá là 400:

**from p in products**: Đây là cú pháp để khai báo biến tạm thời **p** và lấy giá trị từ mỗi phần tử trong danh sách **products**. **products** có thể là một danh sách các đối tượng sản phẩm.

**where p.Price == 400**: Điều này là điều kiện lọc. Chỉ những phần tử **p** mà thỏa mãn điều kiện **p.Price == 400** mới được chọn. Nói cách khác, chỉ những sản phẩm có giá bán bằng 400 sẽ được bao gồm trong kết quả.

**select p**: Kết quả của câu truy vấn sẽ chứa các phần tử **p** đã được chọn từ bước trước đó. Trong trường hợp này, nó sẽ là danh sách các sản phẩm (hoặc các phần tử) có giá bán là 400.

Kết quả:



## API LINQ

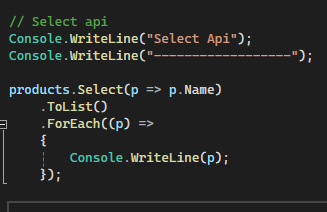
Khi ta nạp thư viện System.Linq nó sẽ mở rộng cho Ienumerable có thêm những phương thức mới, những phương thức này nhằm phục vụ lọc, truy vấn, lấy, biến đổi dữ liệu từ 1 nguồn dữ liệu

### Phương thức Select

<IEnumerable>.Select(<delegate>)

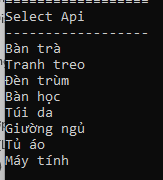
Delegate có tham số là kiểu của phần tử và trả về một giá trị nào đó tính toán dự trên các phần tử

Phương thức Select trả về 1 tập hợp (IEnumerabler) – Gần giống như phương thức **map** trong ES6

Được hiểu là select trên từng phần tử p của nguồn dữ liệu products lấy ra Name của p và tạo thành 1 tập hợp mới chỉ gồm p.name

Phương thức này trả về 1 IEnumerable vì vậy ta có thể hiển thị các phần tử bằng cách sử dụng foreach hoặc chuyển về List bằng phương thức ToList sau đó sử dụng phương thức ForEach như bên

Kết quả:

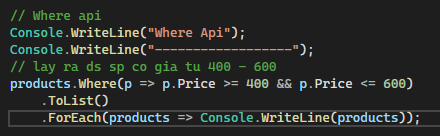
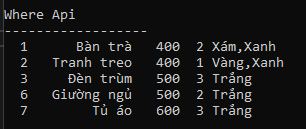


### Phương thức Where

<IEnumerable>.Where(<delegate>)

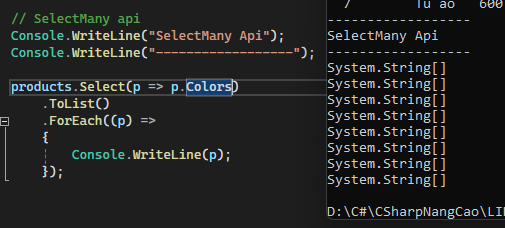
Delegate có tham số là kiểu của phần tử và trả về kiểu boolean, nếu trả về true thì phần tử tương ứng được lấy ra

Phương thức Select trả về 1 tập hợp (IEnumerabler) – Gần giống như phương thức **filter** trong ES6

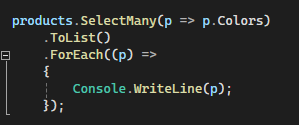
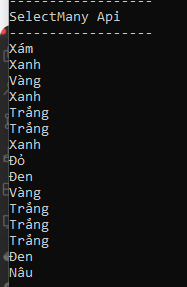


### Phương thức SelectMany

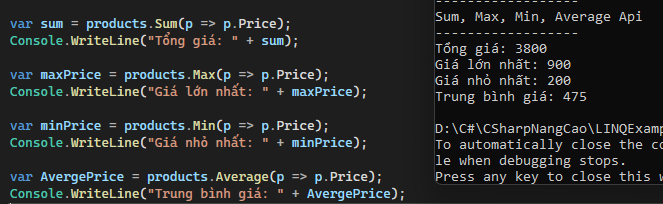
Phương thức này gần giống với Select nhưng ví dụ với Select:



Kết quả ta nhận được đó là các mảng màu sắc trong product do delegate trả về một mảng các màu sắc vì vậy kết quả của select trả về một tập hợp mà mỗi phần tử trong tập hợp là một mảng các màu sắc. Giờ ta muốn các màu sắc hiển thị cụ thể ta sẽ sử dụng SelectMany:



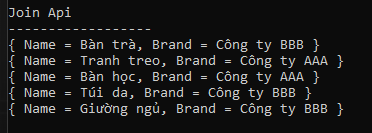
### Min, Max, Sum, Average

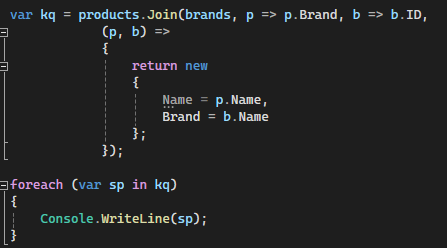


### Join

Phương thức Join dùng để kết hợp 2 nguồn dữ liệu

<IEnumerable>.Join(<Nguồn dữ liệu 1>, <Dữ liệu trong nguồn 2 dùng để kết hợp>, <Dữ liệu trong nguồn 2 dùng để kết hợp>, <delegate> )



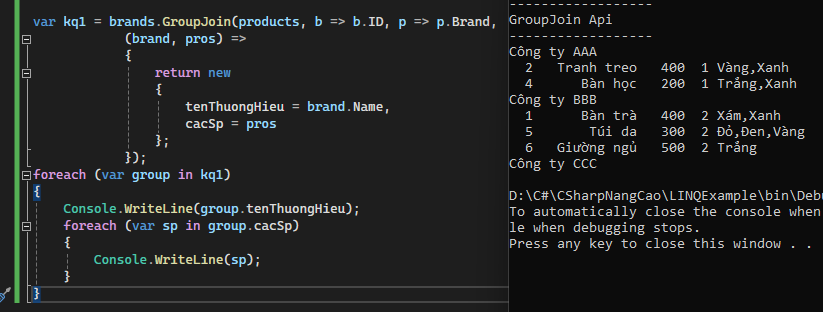


Ta thấy trong kết quả có 5 sp nhưng trong tập hợp products có 8 sp, vì trong tập hợp brands khoogn có brand nào có id = 3, nên join sẽ chỉ lấy ra các products nào có id nằm trong brands

### GroupJoin

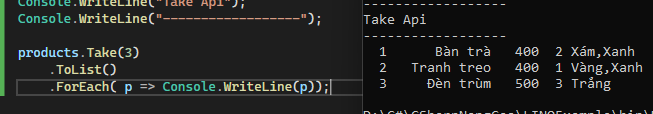
Hoạt động giống join nhưng trả về kết quả là những nhóm được nhóm lại theo nguồn ban đầu

<IEnumerable>.GroupJoin(<Nguồn dữ liệu nằm trong nhóm>, <dữ liệu ban đầu đem ra tạo nhóm>, <dữ liệu cần so sánh với dữ liệu ban đầu>, delegate(<Nhóm>,<Phần tử thuộc nhóm>))



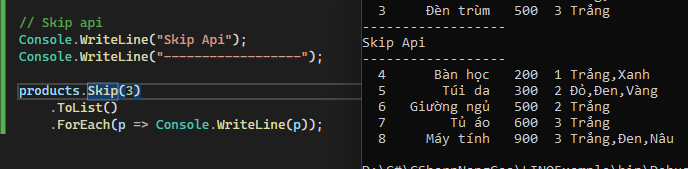
### Take

Lấy ra 1 số lượng phần tử đầu tiên



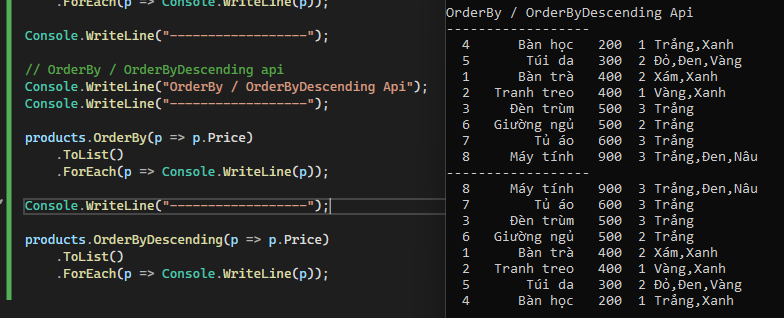
### Skip

Bỏ qua một số lượng phần tử đầu tiên và lấy phần tử còn lại



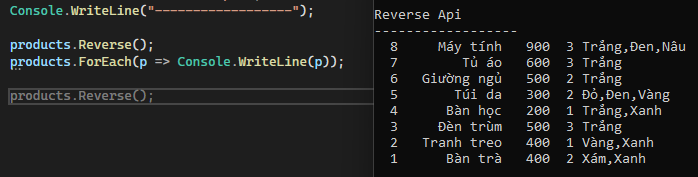
### OrderBy (tăng dần) / OrderByDescending (Giảm dần)

Sắp xếp các phần tử



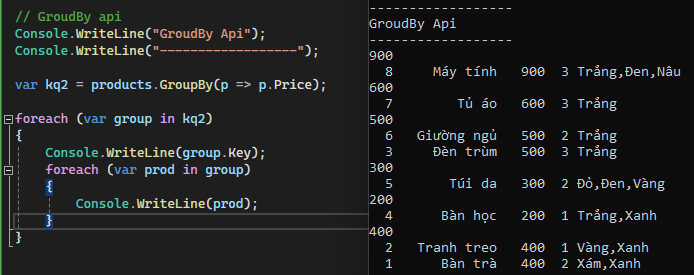
### Reverse

Đảo ngược thứ tự phần tử



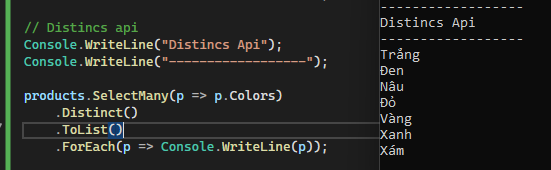
### GroupBy

Nhóm các phần tử theo nhóm



### Distinct

Loại bỏ các phần tử có cùng giá trị và chỉ giữ lại một phần tử



### Single/ SingleOrDefault

#### Single

Kiểm tra các phần tử thỏa mãn một điều kiện logic nào đó, nếu trong kết quả có 1 phần tử thỏa mãn điều kiện thì trả về phần tử đó, nếu không có phần tử nào thỏa mãn logic hoặc có nhiều phần tử cùng thảo mãn logic thì sẽ phát sinh lỗi

#### SingleOrDefault

Phương thức này chỉ khác phương thức single là trong trường hợp không tìm thấy sẽ không phát sinh lỗi mà trả về null

### Any

Trả về true/ false

Phương thức này giống như **some** trong ES6

### All

Trả về true/ false

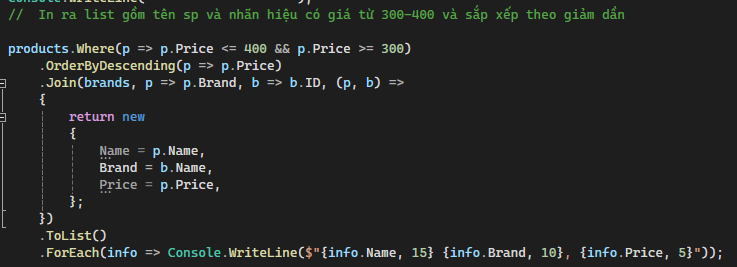
Phương thức này giống như **every** trong ES6

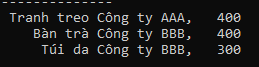
### Count

Đếm tất cả các sản phẩm thỏa mãn điều kiện

Ví dụ tổng hợp sử dụng API LINQ:

* In ra list gồm tên sp và nhãn hiệu có giá từ 300-400 và sắp xếp theo giảm dần





## Sử dụng cú pháp LINQ truy vấn nguồn dữ liệu

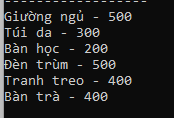
Cú pháp bao gồm 2 phần:

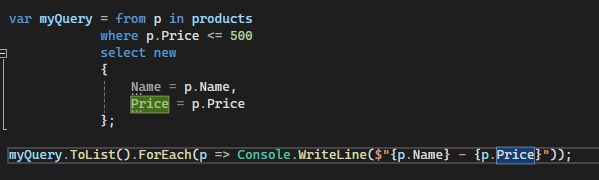
1. Xác định nguồn dữ liệu: Xuất phát từ các đối tượng triển khai từ IEnumerables

* From <tên phần tử> in IEnumerable

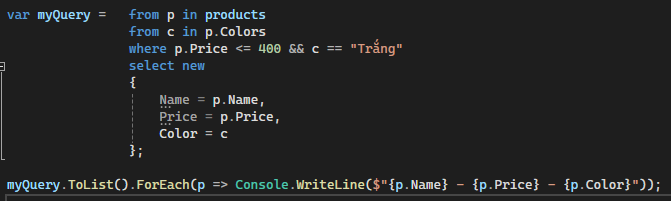
1. Biểu thức: Lọc dữ liệu (Where), sắp xếp (orderBy), Join (Kết hợp dữ liệu),…
2. Lấy dữ liệu: select, group by

### Trả về các phần tử sử dụng Select

Lấy ra tập hợp các sp có giá <= 500

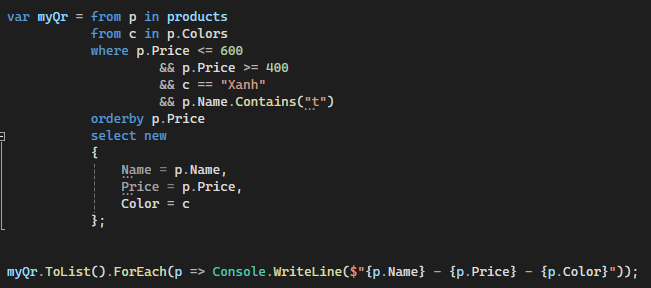


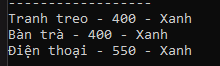
Lấy ra tập hợp các sp có giá <=400 và có màu trắng





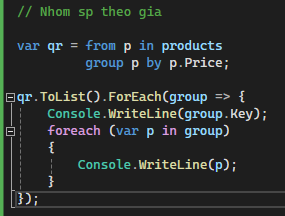
Lấy ra tập hợp gồm các phần tử có giá từ 400 – 600, có màu xanh, tên sp có chứa ký tự “t” và sắp xép theo thứ tự tang dần của giá





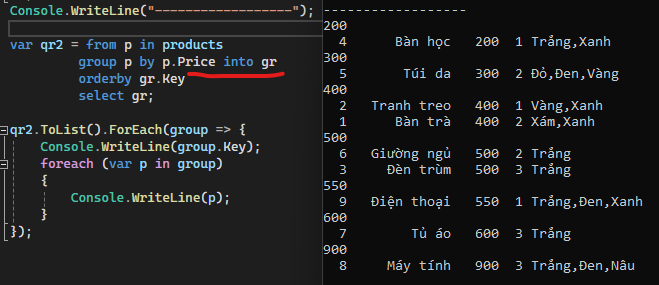
### Trả về các phần tử sử dụng GroupBy

Nhóm kết quả thành từng nhóm theo một thuộc tính nào đó, trả về 1 tập hợp các group, các group này có key và theo thuộc định để nhóm thỏa mãn key

VD: Nhóm phần tử theo giá

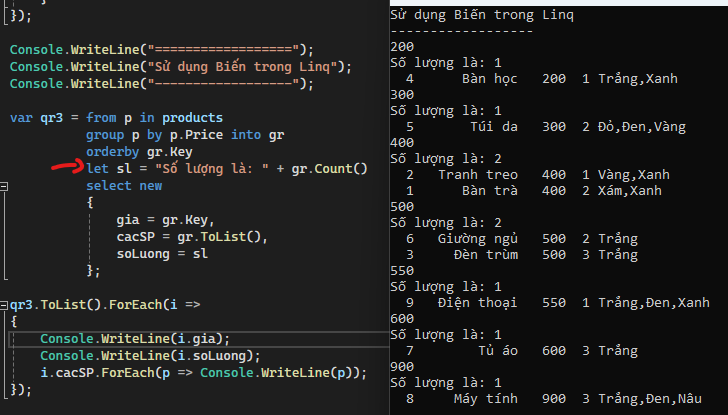


Ta có thể lưu trực tiếp các group vào 1 biến và thao tác trên biến đó bằng từ khóa into



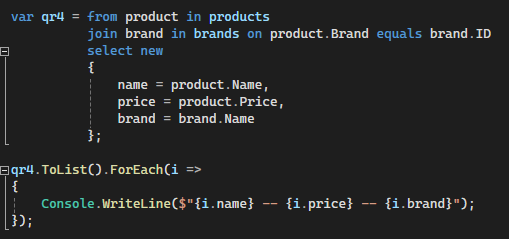
### Biến trong Linq

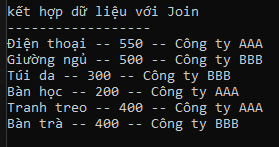
VD: Truy vấn và trả về các đối tượng chứa các thuộc tính giá, các sp, số lượng sp



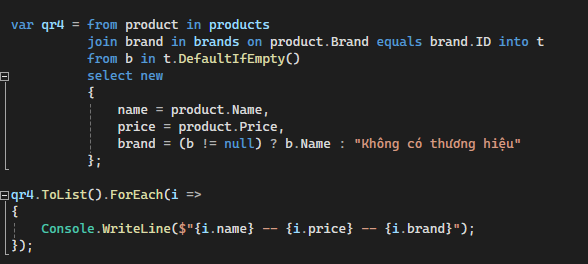
### Kết hợp dữ liệu với Join

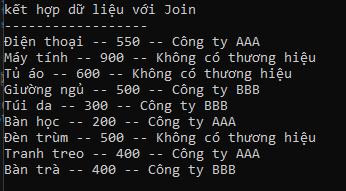
VD: Liệt kê ra các sp gồm tên sp, thương hiệu, giá sp





Trường hợp ta vẫn muốn lấy tất cả các sp kể cả sp không xác định brand





# 19-Lập trình bất đồng bộ (asynchronous) trong C#

## Giới thiệu và chuẩn bị

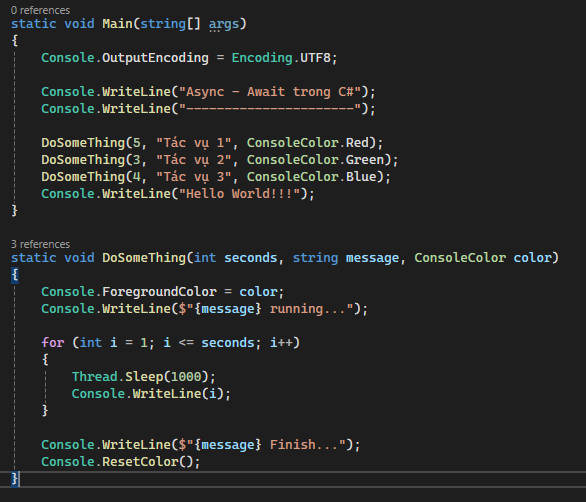


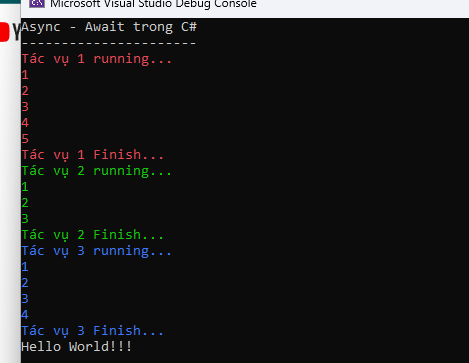
Trong .net cos 1 phương thức Thread.Sleep( miliseconds ) giống như setTimeOut trong JS

## Lập trình đồng bộ trong C#

* Đây là kỹ thuật lập trình cơ bản chạy đơn luồng, là cách mà chúng ta vẫn viết code từ trước đến nay

Ta có đoạn code sau:





Ta thấy rằng chương trình sẽ thực hiện 3 tác vụ sau đó mới hiển thị hello world

Đây là cách thức lập trình đồng bộ, các chỉ thị lệnh thi hành theo một tuần tự nhất định

## Lập trình bất đồng bộ trong C#

* .net cho phép chúng ta tạo ra nhiều các tác vụ, các tác vụ này có thể chạy song song, đồng thời với nhau và chạy trên nhiều thread khác nhau

### Task và Task<T>

* Một đối tượng lớp Task biểu diễn một tác vụ

Khởi tạo: có 2 cách

Task t1 = new Task(Action); // () => {}

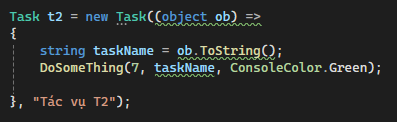
Tham số khởi tạo ở cách này là 1 deleage không có tham số và không có trả về



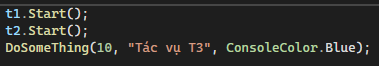
Task t2 = new Task(Action<Object>, Object); // (Object ob) => {}

Ở cách này:

* Tham số thứ nhất tương đương với một biểu thức lambda nhận tham số kiểu Object và không có kiểu trả về
* Tham số thứ 2 là một Object được dùng để truyền vào Action (Được dùng làm tham số của Action)

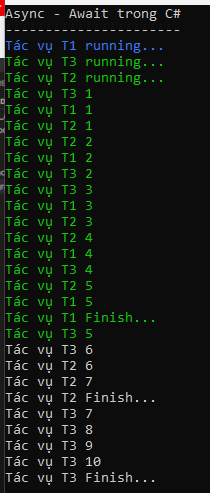
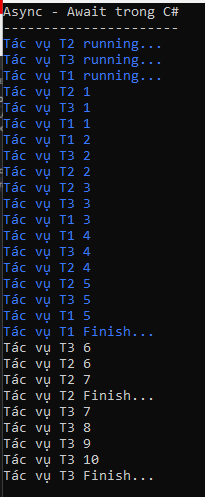
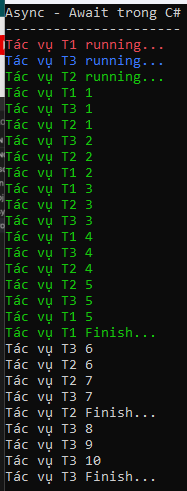


Khởi chạy tác vụ:



Những tác vụ này chạy song song trên các thread khác nhau

Kết quả:

Ta thấy rằng phần thay đổi màu đang bị nhiều tác vụ cùng thao tác 1 lúc để xử lý điều này ta sử dụng các sau:

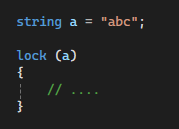
## Khóa truy cập tới các luồng khác

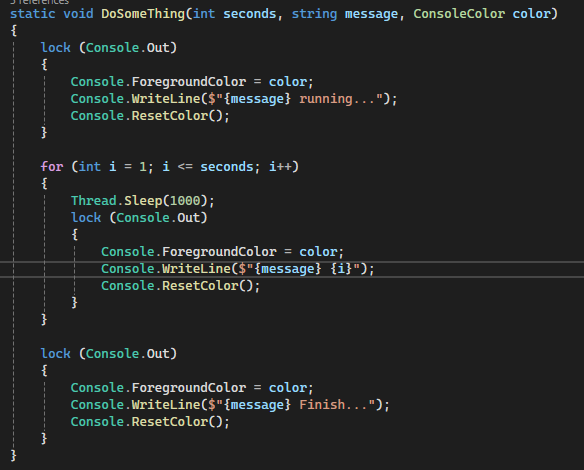
Giả sử ta có 1 biến tham chiếu a



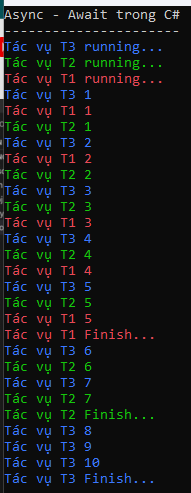
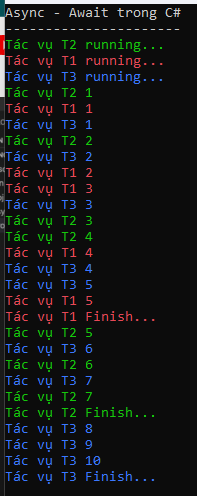
Biến này có thể được sử dụng bởi nhiều thread, đến một lúc nào đó ta muốn biến a này bị khóa lại chỉ được sử dụng bởi luồng hiện tại, các thread khác không được phép truy cập vào a, nếu những thread khác muốn truy cập vào a thì cần phải đợi cho thread hiện tại mở khóa biến a thì mới được truy cập

Để làm như vậy ta làm như sau:

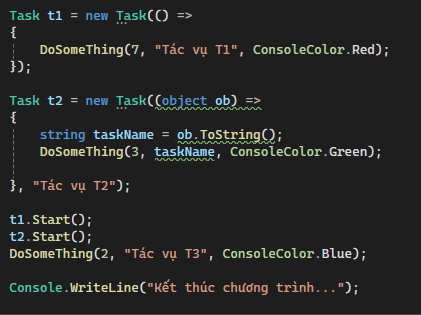
Như cậy trong trường hợp này biến a sẽ được khóa lại đến khi thực thi được hết khối lệnh bên trong



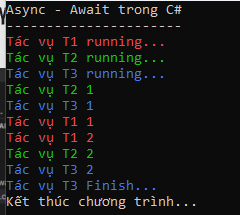
Kết quả:

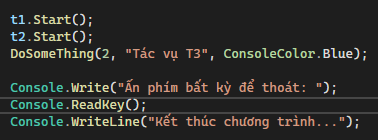
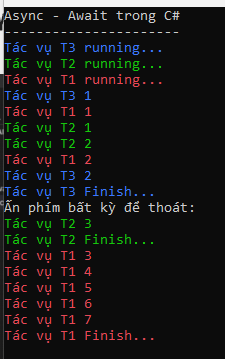
Ta thử chạy trường hợp sau:

Trường hợp này tác vụ 3 sẽ kết thúc trước tác vụ 1 và 2:

Kết quả:

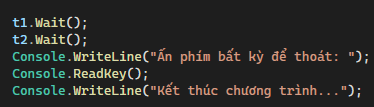
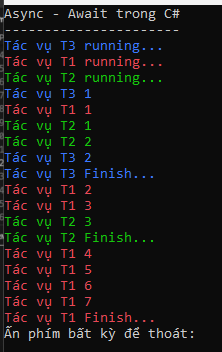
Ở đây tác vụ t1 và t2 vẫn đang chạy nhưng vì tác vụ 3 đã chạy xong nên nó sẽ chạy tiếp code bên dưới hiển thị Kết thúc chương trình và kết thúc hàm main

Để chứng minh điều này ta làm như sau:

 Kết quả:

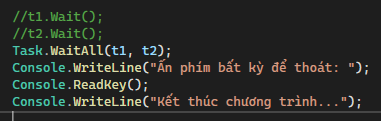
Chúng ta muốn rằng trước khi hiện ra “Ấn phím bất kỳ để thoát:” ta cần đảm bảo T1 và T2 đều phải kết thúc.

Để làm điều này ta làm như sau:



Phương thức **Wait()** đảm bảo tác vụ gọi phương thức phải hoàn thành thì mới thực hiện các tác vụ tiếp theo phía sau **wait**

Chúng ta cũng có thể sử dụng **WaitAll**

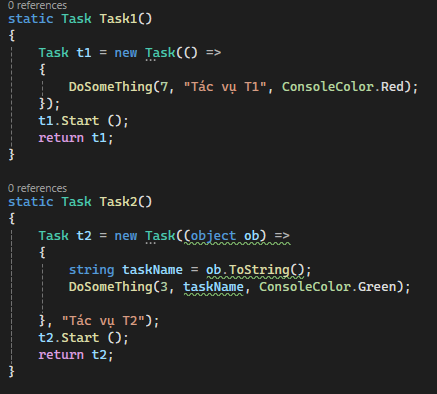


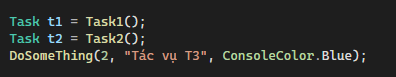
## Xây dựng phương thức bất đồng bộ

Trong thực tế các tác vụ như t1 và t2 thường được khai báo trong những phương thức

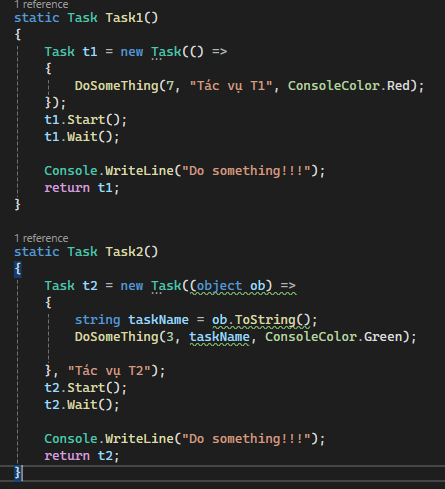
Chúng ta có thể khia báo các phương thức trả về **Task**

Ta khai báo 2 phương thức sau:

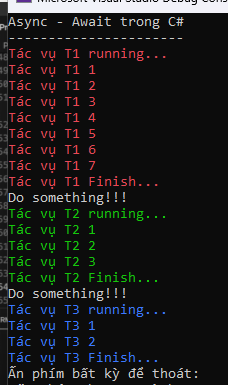




Trong phần khai báo trên giả sử ta muốn khi t1 hoàn thành thì sẽ thực hiện 1 việc gì đó khi t1 hoàn thành và t2 cũng vậy, thì ta sẽ nghĩ ngay đến như sau:



Nhưng khi ta chạy kết quả lại không như mong muốn:

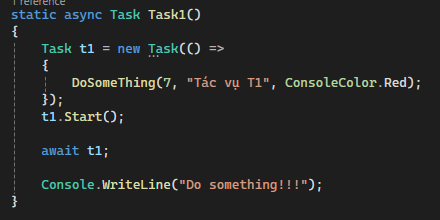
Ta nhận thấy khi ta gọi task 1 thông qua phương thức Task1 bên trong phương thức gọi Wait() điều này dẫn tới luồng chính gọi nó phải chờ cho Task1 hoàn thành toàn bộ và trả về thì mới tiếp tục thực thi đến dòng code phía dưới

Để giải quyết vấn đề này C# đã tạo ra 2 từ khóa để xử lý việc này

## Asynch – Await không có giá trị trả về

2 từ khóa này dùng để khai báo 1 phương thức thành phương thức bất đồng bộ

Để khai báo 1 phương thức là 1 phương thức bất đồng bộ ta sẽ them từ khóa async trước trên phương thức khi khai báo

Lúc này khi một phương thức có từ khóa **async** thì bắt buộc bên trong phương thức phải có **await.**

Từ khóa await theo sau nó là tác vụ.

Về hình thức await tương đương với việc gọi phương thức Wait()

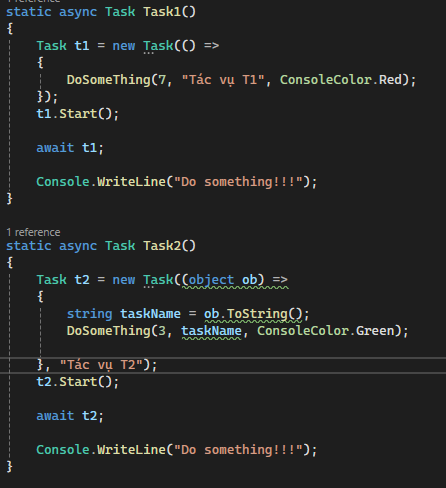
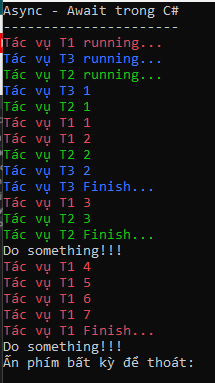
Nó đảm bảo những chỉ thị phía sau await chỉ được thi hành khi tác vụ t1 đã hoàn thành

Nhưng có một điểm khác biệt giữa t1.Wait() và await t1 là tại thời điểm await phương thức trả về Task luôn

Vì vậy phương thức không cần return

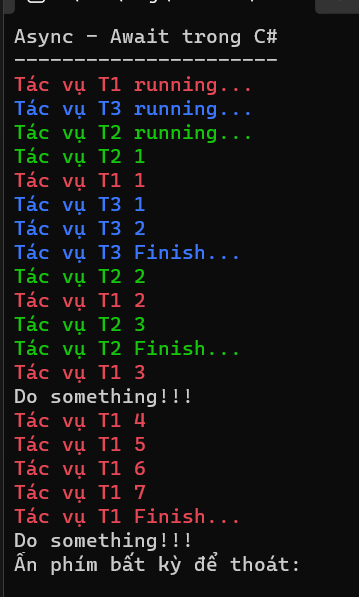
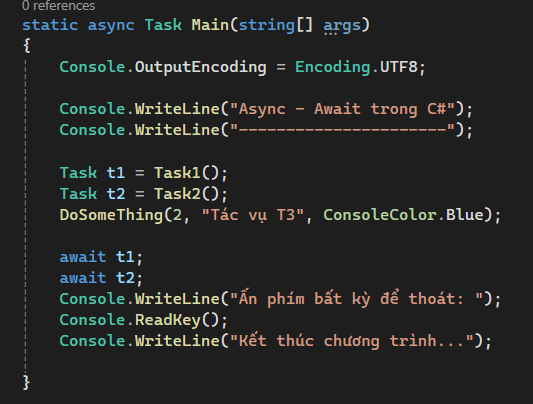
Và khi phương thức trả về như vậy nó sẽ không lock main thread

Tương tự với task 2

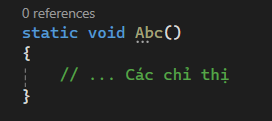
 

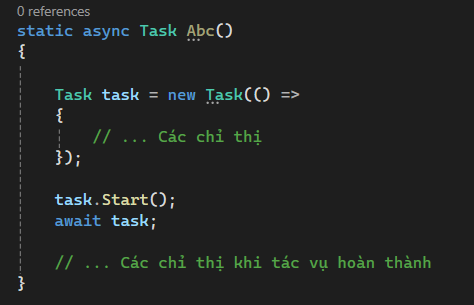
Như vậy chúng ta đã sử dụng từ khóa async await để tạo ra những phương thức bất đồng bộ, và những phương thức này được chạy trên các thread riêng, bên trong phương thức thì vẫn có khả năng chờ cho một tác vụ nào đó hoàn thành rồi mới thực hiện các tác vụ khác mặc dù chờ như vậy nhưng nó vẫn không lock main thread

Chúng ta cũng có thể biến hàm main thành 1 hàm bất đồng bộ:



Đây là cách biến 1 phương thức bất kỳ có dạng trả về là void thành phương thức bất đồng bộ:





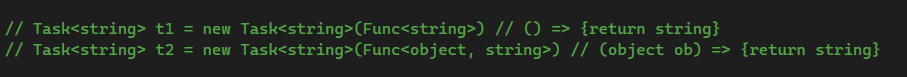
* Ta cần thêm từ khóa async và thay void bằng Task
* Trong phương thức ta cần tạo ra các tác vụ
* Những chỉ thị trong phương thức sẽ được đưa vào trong tác vụ thông qua delegate
* Sau đó tác vụ phải được khởi chạy
* Bên trong phương thức bất đồng bộ phải sử dụng từ khóa await để chờ cho tác vụ hoàn thành
* Phương thức Abc sẽ trả về tại thời điểm await

## Async – await có giá trị trả về

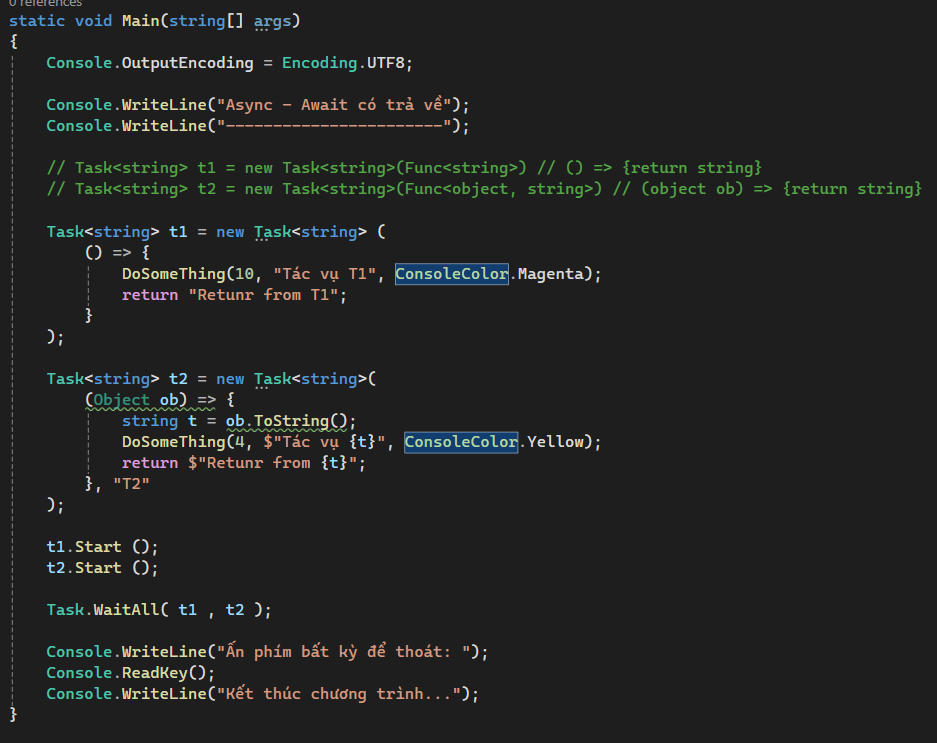
Trong trường hợp bất đồng bộ có giá trị trả về ta sử dụng lớp **Task<T>**

Trong tường hợp chúng ta muốn tạo ra tác vụ mà khi tác vụ đó chạy xong nó có một giá trị trả về

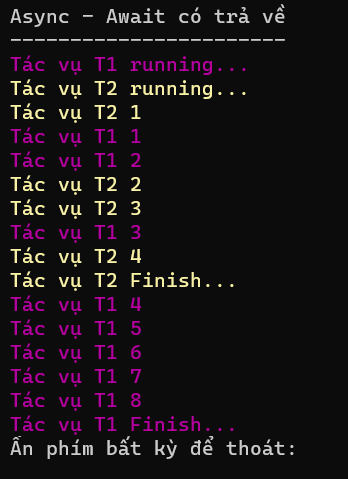
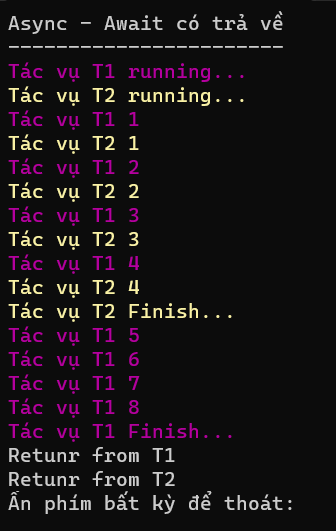
Task<Kiểu trả về> t1 = new Task<Kiểu trả về>(Func<Kiểu trả về>)



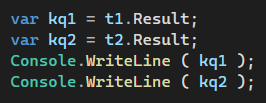
VD:

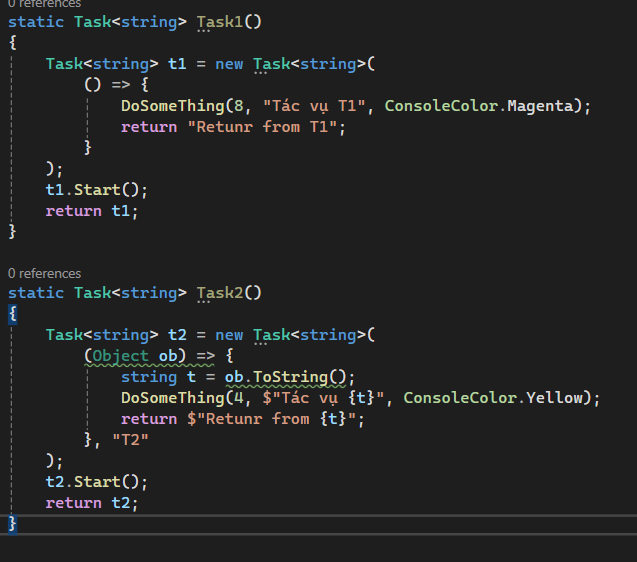
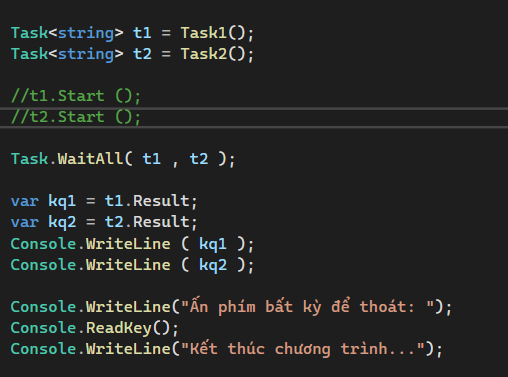


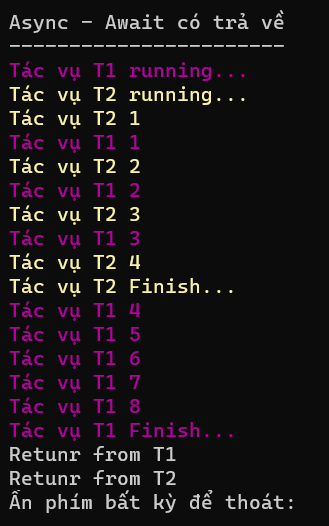
Kết quả:

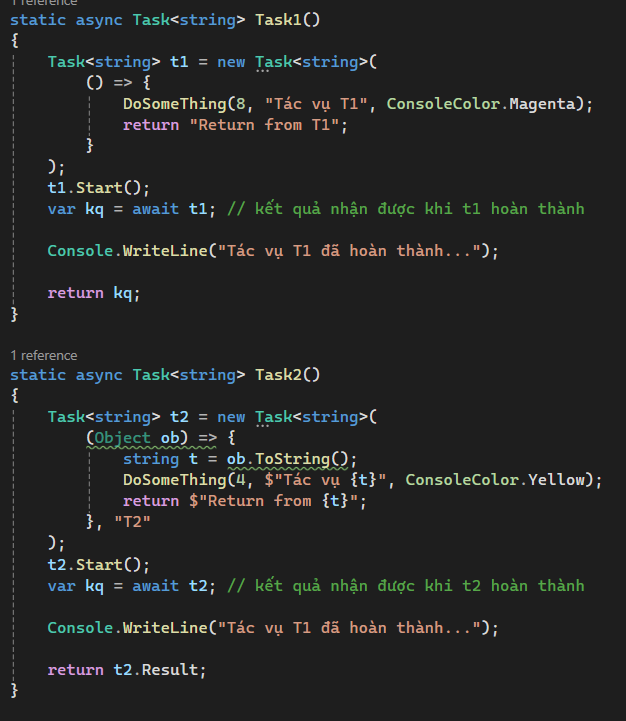
Chúng ta thấy rằng cách hoạt động giống như async – await không có return

Nhưng ở đây chúng ta có thể lấy ra giá trị trả về sau khi tách vụ này chạy xong thông qua **Result**



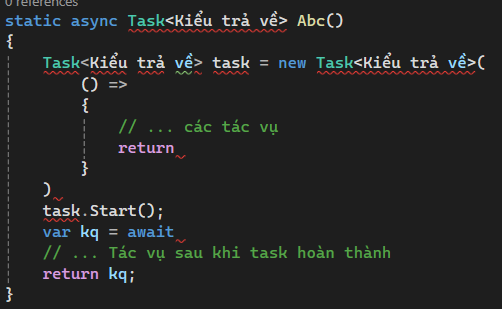
Tương tự như phần trước ta cũng có thể tạo ra phương thức bất đồng bộ cho trường hợp này:

Kết quả tương tự:

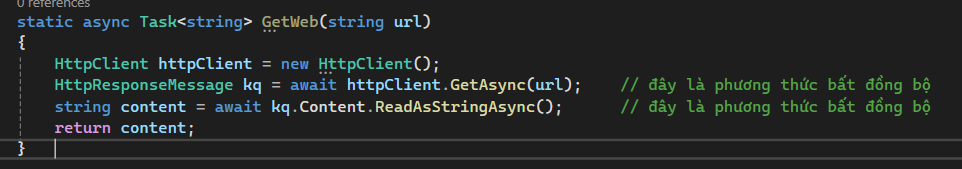
Trong phương thức này không trả về 1 Task mà trả về dữ kiệu kiểu string (là dữ liệu khai báo trong generic)

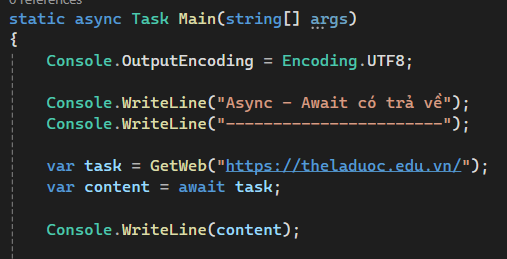
Tổng quát:





## Ví dụ tạo một phương thức để tải 1 trang web





Kết quả:



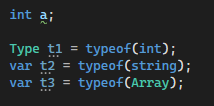
# 20-Type

## Giới thiệu về lớp type

* Lớp Type là lớp chứa thông tin về một kiểu dữ liệu nào đó
* Ví dụ type chứa thông tin về: class, struct,… int, bool,…
* Type thường dùng cho kỹ thuật Reflection (Lấy thông tin của kiểu dữu liệu, ở thời điểm thực thi)

VD: ta khai báo có một biến **int a;**

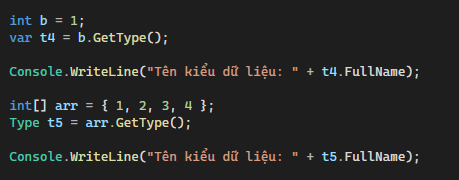
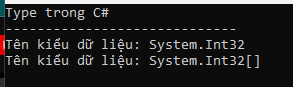
* Ở thời điểm thực thi, ta có thể truy ngược lại kiểu int có những thuộc tính, phương thức, trường dữ liệu nào?
* Để làm được diều đó ta có đối tượng Type chứa các thông tin về kiểu int



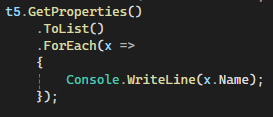
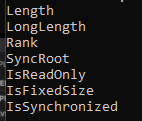
Để lấy được Type của một đối tượng nào đó ta sử dụng phương thức **GetType()**



Khi có đối tượng Type ta có thể truy vấn để lấy ra thông tin:

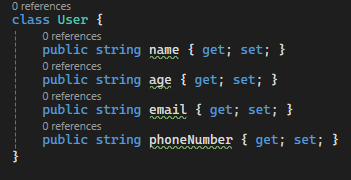
 

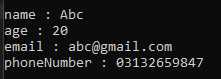
Để lấy ra các thuộc tính:

## Đọc thuộc tính của một đối tượng

Ta có class sau:

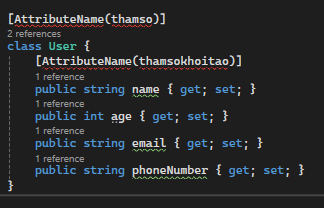


## Sử dụng Attribute trong C#

* Thuộc tính attribute là một phần của siêu dữu liệu cung cấp thông tin bổ sung cho một lớp hoặc các thành viên của lớp
* Sử dụng atttribute bổ sung cho class, thuộc tính của class, phương thức,.. thì các attribute này được sử dụng bởi các thư viện, framework, nền tảng sau này
* Những thông tin lấy sẽ được lấy ra sử dụng ở thời điểm thực thi

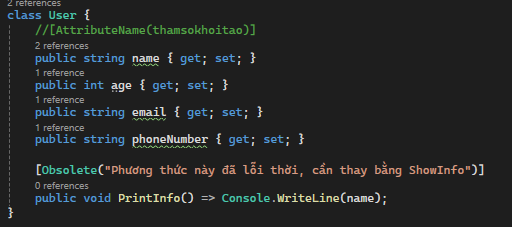
Giả sử ta có 1 thuộc tính là AttributeName

Để khai báo 1 attribute cho class trước khi khai báo class ta sử dụng **[<tên attribute>(tham số)]**

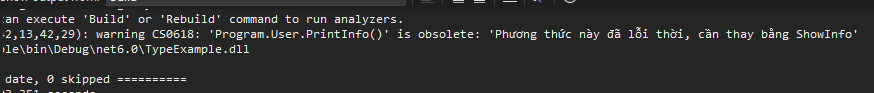
Để khai báo 1 attribute cho thuộc tính nào đó của class class trước khi khai báo thuộc tính class ta sử dụng **[<tên attribute>(tham số)]**

Bằng cách này các đối tượng tạo ra từ class User thì đối tượng đấy sẽ có thêm thông tin Attribute và thuộc tính name cũng có thêm thông tin Atttribute

VD:

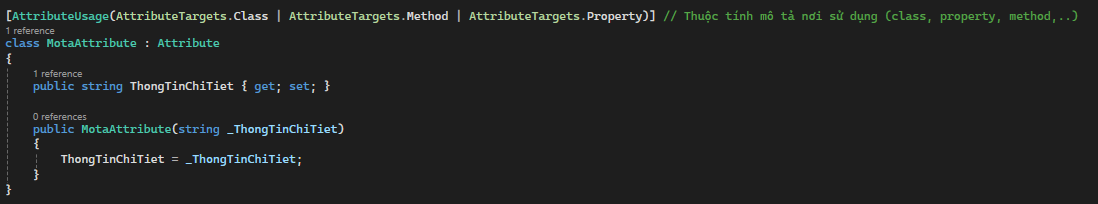


Khi sử dụng phương thức này trình biên dịch sẽ hiển thị cảnh báo:



## Thực hành tạo Atttribute

Ta sẽ tạo 1 attribute là mota

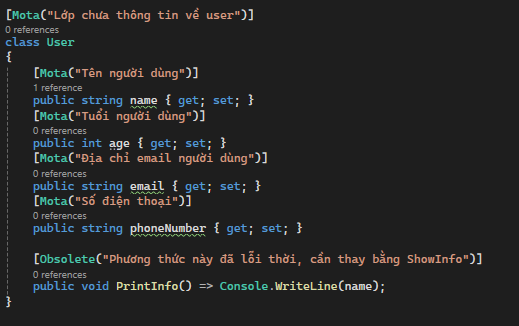


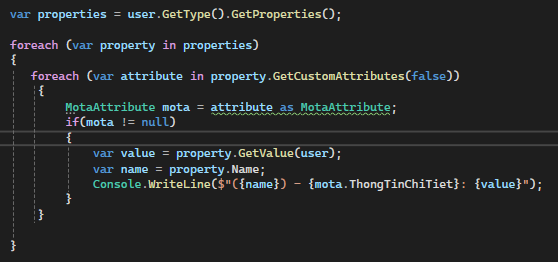
- Tạo class có tên cú pháp <Tên attribute>Attribute

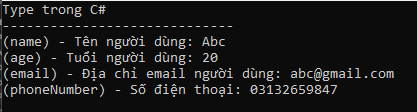
- Class này kế thừa từ class Attribute

- Bản thân class phải sử dụng AttributeUsage để khai báo Attribute sử dụng cho thành phần nào

Sử dụng:







# 21-Attribute Annotation (Attribute chú thích)

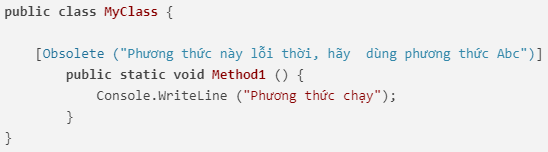
## Khái niệm về Annotation (Attribute)

Một thuộc tính chú thích (Annotation / Attribute) tác động vào một thành phần nào đó của chương trình (lớp, phương thức, thuộc tính) nó là một phần của siêu dữ liệu (metadata - loại dữ liệu cung cấp thêm thông tin về đối tượng nào đó). Annotation giúp thêm thông tin vào lớp, phương thức, thuộc tính những đoạn code mở rộng. Tính năng này trong Java gọi là Annotation, trong C# gọi là Attribute.

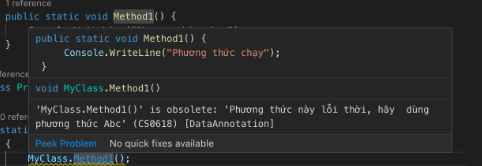
Các thuộc tính chú thích có thể được truy xuất tra cứu ở thời điểm thực thi bằng kỹ thuật gọi là **reflection**, truy xuất ngược từ đối tượng biết được nguồn gốc mà đối tượng đó sinh ra (lớp).

## Obsolete

Obsolete, để đánh dấu một phương thức, lớp ... nào đó là lạc hậu



Kết quả:



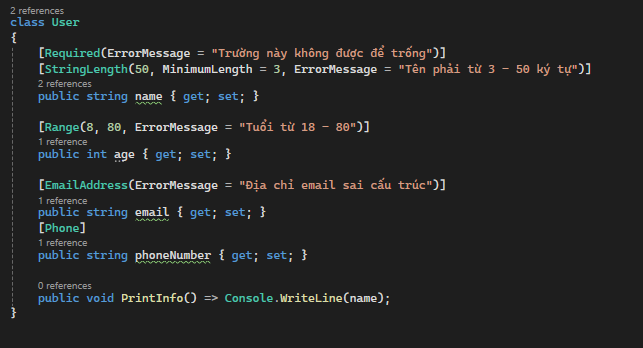
## Phân loại Data Annotation/Attribute trong C#

Các Data Annotation/Attribute trong C# định nghĩa trong namespace System.ComponentModel.DataAnnotations, có các loại gồm:

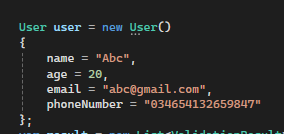
1. Các Attribute để kiểm tra dữ liệu (Validation Attribute)
2. Các Attribute hiện thị (Display Attribute), điều khiển dữ liệu trong lớp hiện thị thế nào trong UI
3. Modelling Attribute

|  |  |
| --- | --- |
| Required | Dữ liệu phải được thiết lập (!=null)  [Required (ErrorMessage="{0} cần thiết lập")] |
| StringLength | Thiết lập độ dài trường dữ liệu  [StringLength (20,MinimumLength=3, ErrorMessage="Phải dài 3 đến 20 ký tự")] |
| DataType | Chỉ ra dữ liệu phải liên kết phù hợp với một kiểu nào đó  [DataType(DataType.Text)]  [DataType(DataType.PhoneNumber)]  [DataType(DataType.EmailAddress)]  /.. Date, DateTime, Html, ImageUrl, MultilineText, Password, Time, Url |
| Range | Chỉ ra dữ liệu phải trong khoảng nào đó  [Range(18,99, ErrorMessage="Tuổi từ 18 đến 99")]  [Range(typeof(DateTime), "1/2/2004", "3/4/2004",  ErrorMessage = "Value for {0} must be between {1} and {2}")] |
| Phone | [Phone] dữ liệu phải là dạng số điện thoại |
| EmailAddress | [EmailAddress] dữ liệu phải là dạng email |

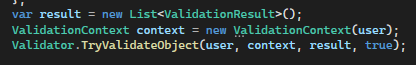
VD:



Sau đó ta có dữ liệu khởi tạo như sau:

Giờ chúng ta cần phải kiểm tra dữ liệu khởi tạo xem có thỏa mãn hay không?

## Validate data

Để validate dữ liệu ta sẽ sử dụng lớp là validate context

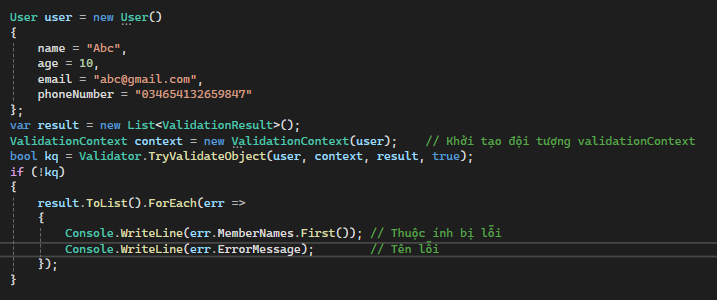
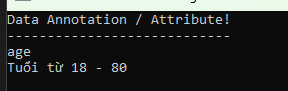
* Khởi tạo đội tượng ValidationContext(<đối tượng cần kiểm tra>);
* Để kiểm tra chúng ta cần gọi đến một Phương thức tĩnh của lớp Validator là **TryValidateObject(**

**<đối tượng cần kiểm tra>,**

**<ngữ cảnh kiểm tra>,**

**<kết quả các lỗi kiểm tra nếu có>,**

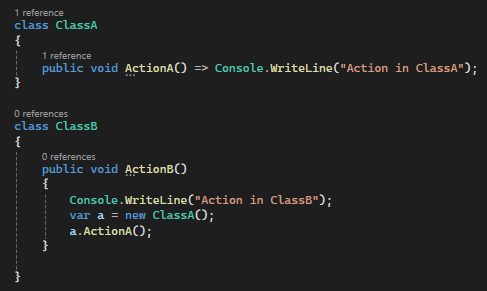
**<true: kiểm tra tất cả các thuộc tính của đối tượng>)**

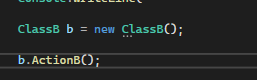
* Kết quả tra về của phương thức này là boolean trả về true nếu tất cả validate đều phù hợp
* 
* Kết quả:
* 

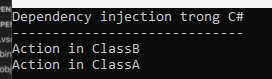
# 22-DI Dependency Injection

## Dependency (Phụ thuộc)

Giả sử ta có 2 class sau:

trong phuong thức ActionB có gọi phương thức ActionA của class A

Kết quả khi ta tthi hành code:



Như vậy chức năng của class B phụ thuộc vào class A hay nói cách khác class B dựa vào class A để hoàn thành nhiệm vụ. Lúc này ta nói class A là dependency của class B

Trong thiết kế ứng dụng việc một class có được các dependency của nó như thế nào thì nó hình thành nên 1 số nguyên lý thiết kế ứng dụng. Trong đó **Dependency Injection** là một nguyên lý, nó đưa ra cách thức 1 dependency được đưa vào class sử dụng nó

Để tìm hiểu về **Dependency Injection** trước tiên ta sẽ tìm hiểu về Inverse of Control (IoC)

## IoC Inverse of Control (IoC) / Dependency inversion

* Inverse of Control (IoC) hay còn gọi là đảo ngược điều khiển là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.
* Các thành phần nó dựa vào để làm việc chính là các Dependency

### Thiết kế truyền thống - tham chiếu trực tiếp đến Dependency

Giả sử: ta có Class A muốn hoàn thành nhiệm vụ gì đó cần phải sử dụng chức năng của class B mà chức năng của class B muốn thực hiện được lại phải thông qua chức năng của 1 class C

Giả sử ta có code như sau:

namespace Tham\_Chieu\_Truc\_Tiep\_Den\_Dependency {

    class ClassA {

        public void CongViecA() {

            Console.WriteLine("Thực hiện công việc A");

            // thực hiện công việc B

            ClassB objB = new ClassB();

            objB.CongViecB();

        }

    }

    class ClassB {

        public void CongViecB() {

            Console.WriteLine("Thực hiện công việc B");

            // thực hiện công việc C

            ClassC objC = new ClassC();

            objC.CongViecC();

        }

    }

    class ClassC {

        public void CongViecC() {

            Console.WriteLine("Thực hiện công việc C");

        }

    }

}

Trong hàm main ta gọi:

static void Main(string[] args)

        {

            Console.OutputEncoding = Encoding.UTF8;

            Console.WriteLine("Dependency injection trong C#");

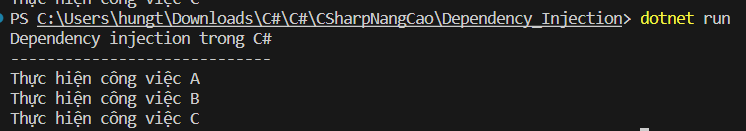
            Console.WriteLine("-----------------------------");

            ClassA objectA = new ClassA();

            objectA.CongViecA();

        }

Kết quả:



Ta có thể viết như sau:

namespace Tham\_Chieu\_Truc\_Tiep\_Den\_Dependency {

    class ClassA {

        // Field

        ClassB \_classB;

        // constructor

        public ClassA(ClassB classB) {

            \_classB = classB;

        }

        public void CongViecA() {

            Console.WriteLine("Thực hiện công việc A");

            // thực hiện công việc B

            \_classB.CongViecB();

        }

    }

    class ClassB {

        // Field

        ClassC \_classC;

        // constructor

        public ClassB(ClassC classC) {

            \_classC = classC;

        }

        public void CongViecB() {

            Console.WriteLine("Thực hiện công việc B");

            // thực hiện công việc B

            \_classC.CongViecC();

        }

    }

    class ClassC {

        public void CongViecC() {

            Console.WriteLine("Thực hiện công việc C");

        }

    }

}

Trong hàm main ta gọi:

static void Main(string[] args)

        {

            Console.OutputEncoding = Encoding.UTF8;

            Console.WriteLine("Dependency injection trong C#");

            Console.WriteLine("-----------------------------");

            ClassC objC = new ClassC();

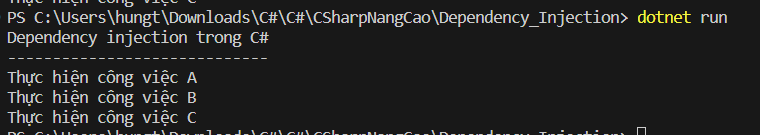
            ClassB objB = new ClassB(objC);

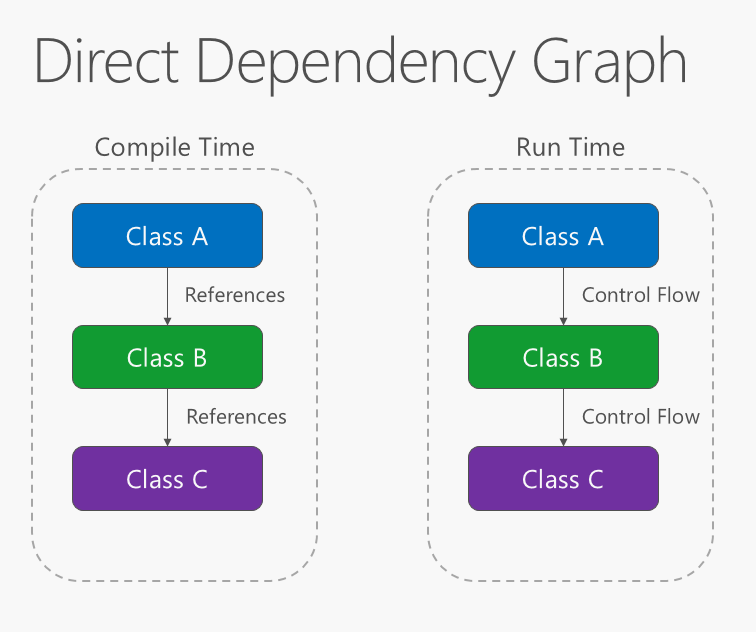
            ClassA objA = new ClassA(objB);

            objA.CongViecA();

        }

Kết quả:





Vào thời điểm viết code(Compile Tiem) ClassA tham chiếu trực tiếp đến ClassB và ClassB tham chiếu trực tiếp đến ClassC

Vào thời điểm thực thi (Runtime) khi tạo ra đối tượng ClassA nó đã tham chiếu trực tiếp đến 1 đối tượng trực tiếp tạo ra classB, classB tham chiếu đến một đối tượng tạo ra ClassC

Ta nhận thấy sự phụ thuộc ở 2 thời điểm viết code và thực thi là giống nhau

### Thiết kế theo cách đảo ngược phụ thuộc Inverse Dependency

Ở phần này ta code lại như sau:

namespace Inverted\_Dependency {

    interface IClassB {

        public void CongViecB();

    }

    interface IClassC {

        public void CongViecC();

    }

    class ClassC : IClassC {

        public void CongViecC() {

            Console.WriteLine("Thực hiện công việc C");

        }

        // Constructor

        public ClassC() {

            Console.WriteLine("ClassC được tạo");

        }

    }

    class ClassB : IClassB {

        IClassC c\_dependency;

        public void CongViecB() {

            Console.WriteLine("Thực hiện công việc B");

            c\_dependency.CongViecC();

        }

        // Constructor

        public ClassB(IClassC classC) {

            c\_dependency = classC;

            Console.WriteLine("ClassB được tạo");

        }

    }

    class ClassA {

        IClassB b\_dependency;

        public void CongViecA() {

            Console.WriteLine("Thực hiện công việc A");

            b\_dependency.CongViecB();

        }

        // Constructor

        public ClassA(IClassB classB) {

            b\_dependency = classB;

            Console.WriteLine("ClassA được tạo");

        }

    }

}

Trong hàm main:

static void Main(string[] args)

        {

            Console.OutputEncoding = Encoding.UTF8;

            Console.WriteLine("Dependency injection trong C#");

            Console.WriteLine("-----------------------------");

            // Tham chiếu trực tiếp đến dependency

            IClassC objC = new ClassC();

            IClassB objB = new ClassB(objC);

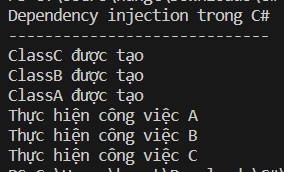
            ClassA objA = new ClassA(objB);

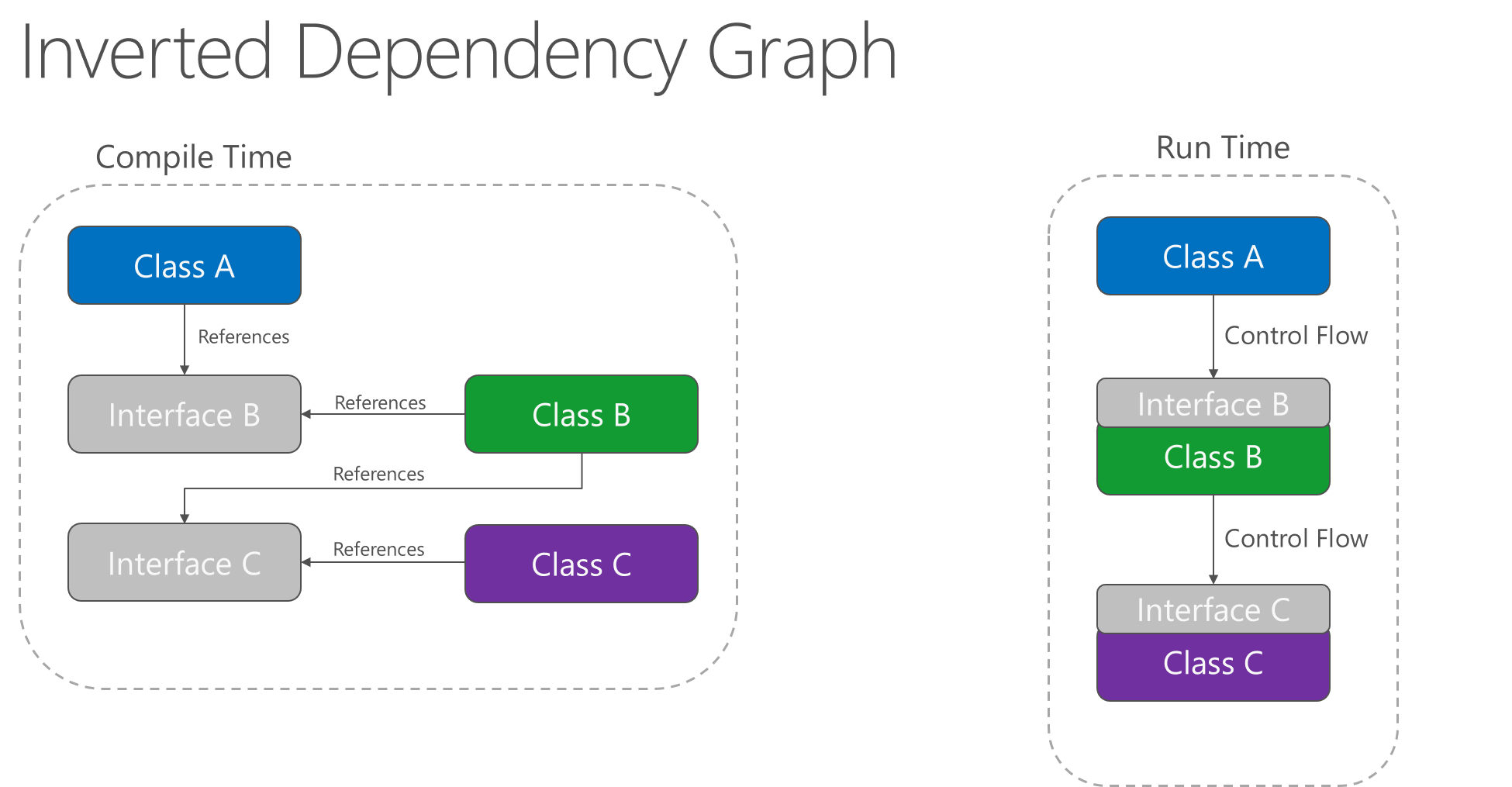
            objA.CongViecA();

            // Inverse Dependency

        }

Kết quả:





* ClassA tham chiếu đến classB và thi hành chức năng trong classB và tương tự giữa classB và classC
* Thời điểm viết code classA không tham chiếu trực tiếp đến classB mà thông qua một Interface hoặc thông qua 1 lớp abstract của classB

Trong trường hợp 2 này có một số đặc điểm mà ta cần lưu ý:

* ClassA tham chiếu tới 1 dependency thông qua InterfaceB, vì vậy khi thực thi ta có thể thay thế vào InterfaceB bất kỳ một class nào được triển khai từ interfaceB
* Điều này có nghĩa là dependency cụ thể được quyết định bởi InterfaceB có thể là classB1, classB2,… miễn là class được triển khai từ interfaceB
* Tương tự với InterfaceC

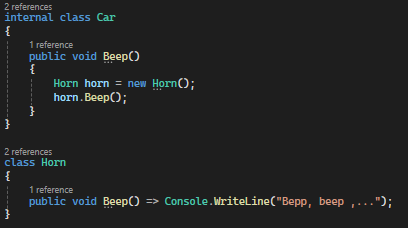
Điều này chính là đảo ngược phụ thuộc, A phụ thuộc B nhưng B là cái gì sẽ được quyết định bởi khai báo trong A

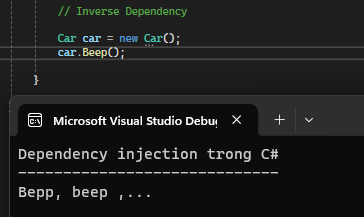
## Dependency Inject

* Là 1 nguyên tắc triển khai từ IoC
* Nguyên lý hoạt động của DI là ta phải thiết kế ra 1 cơ chế nào đó để khi ClassA cần 1 Dependency thì khi thực thi cơ chế đó sẽ tự động tiêm Dependency đó và class A và Dependency đó cụ thể là gì được quyết định bới Interface khai báo trong class A
* Để khai báo những cơ chế đó rất phức tạp nên ở đây ta sẽ sử dụng những thư viện có sẵn mà nó hỗ trợ DI

Khi viết code ta cần viết làm sao để đưa các Dependency từ bên ngoài vào bên trong 1 lớp

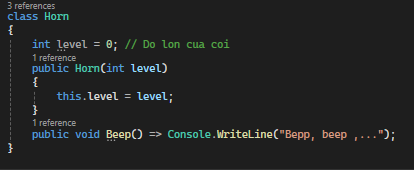
## Bài toán

Ta có 2 class sau:

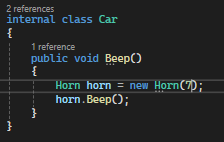
Vấn đề ở đây là class Car tự tạo ra đối tượng cụ thể của Horn do đó ta không thể tiêm vào từ bên ngoài lớp Car một lớp Horn nào khác. Cách thiết kế như này thì nếu thay đổi Horn sẽ dẫn tới việc thay đổi Car

VD:

Class Horn thay đổi như sau:



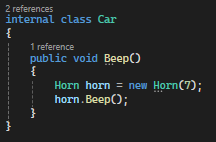
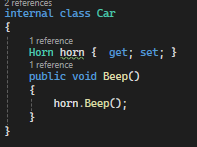
Thì điều này đồng nghĩa ta phải viết code lại trong class Car thì class Car mới sử dụng được



Viết code kiểu này khi ta thay đổi code của dependency thì kéo theo những đối tượng sử dụng dependency đó phải thay đổi theo

Do đó để linh hoạt hơn các Dependency không được tạo trực tiếp trogn các lớp mà nó sẽ được tiêm từ bên ngoài vào và lớp đó chỉ được sử dụng

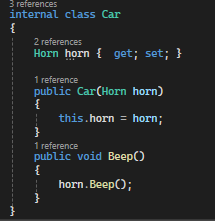
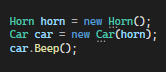
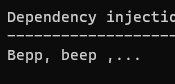
## Thiết kế với sự phụ thuộc lỏng lẻo DI

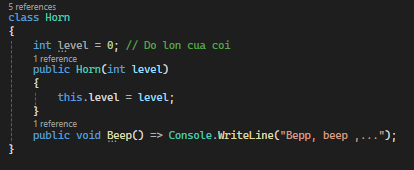
Thay vì lớp Car tự tạo ra lớp Horn thì sẽ khai báo 1 thuộc tính là Horn

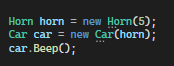
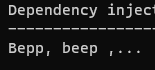
=>

Đối tượng Horn cụ thể tức là các Dependency có thể đưa từ bên ngoài vào lớp Car theo nhiều cách

### Inject thông qua phương thức khởi tạo

Cách này sẽ đưa dependency vào thông qua phương thức khởi tạo

Lúc này khi ta thay đổi code dependency

### Inject thông qua setter

### Inject thông qua Interface

## Thư viện Dependency Inject

Khi ta viết code mà lớp đó có khả năng inject các dependency vào trong nó thông thường ta sẽ không trực tiếp tạo ra các đối tượng bằng cách sử dụng toán tử new mà ta sẽ sử dụng 1 thư viện

Thư viện hỗ trợ Dependency Injection thông thườngcó chức năng cơ bản như sau:

* Đối tượng: DI Container
  + Đối tượng này cho phép đăng ký các service (kiểu lớp) vào trong nó
  + Hỗ trợ lấy ra service (get service), khi nó lấy ra đối tượng thì nó sẽ khởi tạo đối tượng đó, nếu các đối tượng đó cần dependency mà dependency đó chưa có thì nó sẽ tự động khởi tạo ra các dependency và tự động inject vào dịch vụ

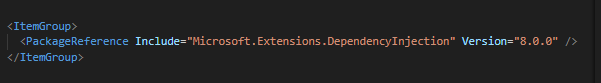
Các thư viện DI:

* Unity
* Ninject
* Microsoft.Extensions.DepedencyInjection (.NET)

### Tích hợp thư viện Microsoft.Extensions.DepedencyInjection

dotnet **add** package Microsoft.Extensions.DependencyInjection

Sau khi cài đặt



Trong thư viện này DI Container chính là lớp ServiceCollection

Ta sẽ đăng ký các service vào ServiceCollection

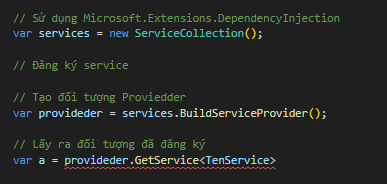
Từ ServiceCollection ta có một đối tượng ServiceProvider cho phép lấy ra các service đã được đăng ký trong ServiceCollection

### ServiceLifetime

**ServiceLifetime:** Mỗi dịch vụ (lớp) khi đăng ký vào ServiceCollection thì có một đối tượng ServiceDescriptor chứa thông tin về dịch vụ đó, căn cứ vào ServiceDescriptor để ServiceCollection khởi tạo dịch vụ khi cần. Trong ServiceDescriptor có thuộc tính Lifetime để xác định dịch vụ tạo ra tồn tại trog bao lâu. Lifetime có kiểu ServiceLifetime (kiểu enum) có các giá trị cụ thể:

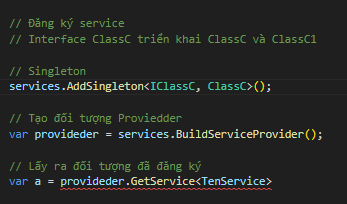
|  |  |  |
| --- | --- | --- |
| Scoped | 1 | Một bản thực thi (instance) của dịch vụ (Class) được tạo ra cho mỗi phạm vi, tức tồn tại cùng với sự tồn tại của một đối tượng kiểu ServiceScope (đối tượng này tạo bằng cách gọi ServiceProvider.CreateScope, đối tượng này hủy thì dịch vụ cũng bị hủy). |
| Singleton | 0 | Duy nhất một phiên bản thực thi (instance of class) (dịch vụ) được tạo ra cho hết vòng đời của ServiceProvider |
| Transient | 2 | Một phiên bản của dịch vụ được tạo mỗi khi được yêu cầu |

Các bước thực hiện

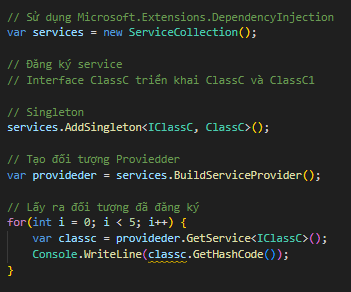
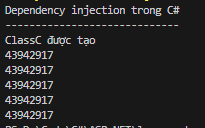


### Đăng ký service kiểu Singleton

* Chỉ tồn tại 1 phiên bản trong suốt vòng đời của Providder

Ta hiểu dịch vụ có tên là IClassC sẽ là đối tượng của lớp ClassC

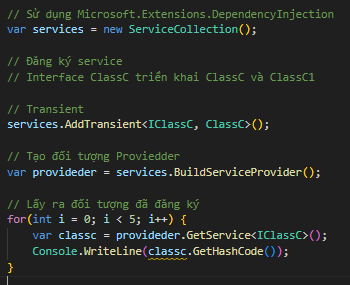
Do đăng ký kiểu singleton nên đối tượng này chỉ là 1 đối tượng duy nhất

Ta nhận thấy các mã Hash là giống hệt nhau vậy là nhưng đối tượng được tạo ra cùng là 1

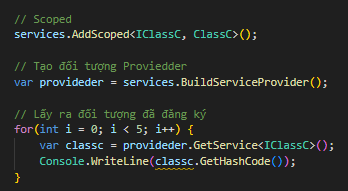
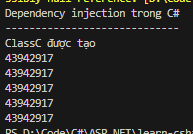
### Đăng ký service kiểu Transient

* Mỗi lần truy cập lây ra dịch vụ một đối tượng mới của dịch vụ sẽ được tạo ra:

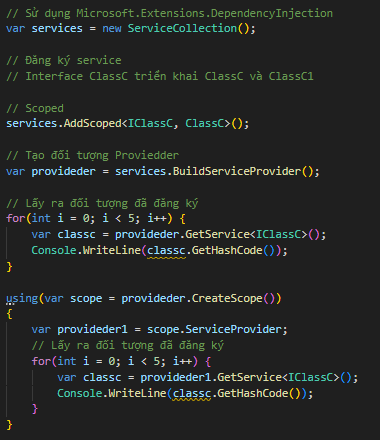
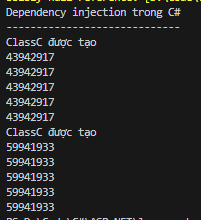
 

### Đăng ký service kiểu Scoped

* Trong mỗi phạm vi chỉ có 1 đối tượng service được tạo ra

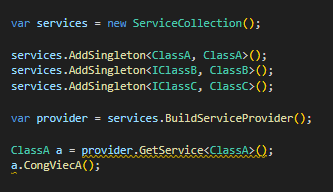
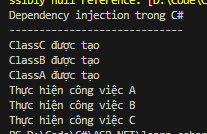
 

Trong trường hợp ta tạo 1 scoped mới:



### Tiêm các phụ thuộc

* Giờ ta sẽ làm cho các dịch vụ được đăng ký vào sẽ tự động được tạo ra và khi tạo ra các dịch vụ đó, nếu dịch vụ đó cần các dependency thì các dependency đó cũng tự động tạo ra và inject và đối tượng cho chúng ta

## Đăng ký dịch vụ với delegate và Factory