# 1-Cài đặt chương trình C# đầu tiên

## .net framework và .net core

.net framework:

* Là nền tảng phát triển và chạy ứng dụng

Net framework 1.0 (2002)

Net core (2016)

Cấu trúc dot net gồm 2 thành phần:

1. CLR (Common Language Runtime)

* Giống như 1 máy ảo, tại đây ứng dụng .net được thực thi

1. FCL (Framework Class Library)

* Các thư viện class

## Ngôn ngữ lập trình C#

* Được phát triển bởi microsoft
* Hướng đối tượng

## Cài đặt .net core SDK

<https://dotnet.microsoft.com/en-us/download>

## IDE

* Visual Studio Code
* Visual Studio

## Kiểm tra phiên bản .net:

A black background with white text

Description automatically generated

## Tạo dự án đầu tiên:

Visual sutio:

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Để chạy dự án:

Ctrl + F5

A screenshot of a computer

Description automatically generated

## Giải thích code:

A screenshot of a computer program

Description automatically generated

Namespace là khai báo 1 vùng chứa các code khác nhau

Để sử dụng namespace ta dùng: using <tên name space>

Class là một lớp

Hàm main là hàm khai báo code sẽ chạy điểm đầu ở đây: và trong 1 file chỉ có 1 hàm main

## Comment

Comment 1 dòng: //

Comment nhiều dòng: /\*\*/

Chú thích code:

A computer screen with text

Description automatically generated

Sau khi chú thích như này chúng ta có thể thấy kết quả như sau:

A screen shot of a computer

Description automatically generated

## Publish

A screenshot of a computer screen

Description automatically generated

Sau khi publish sẽ tạo ra thư trong mục bin và code này có thể chạy được trên các máy khác nhau

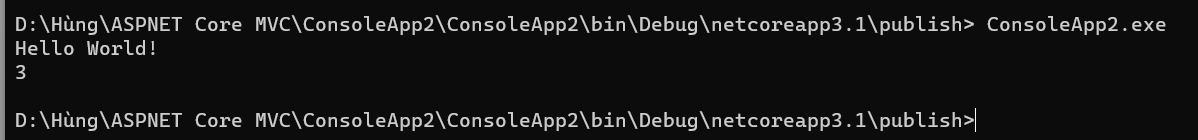
A screenshot of a computer

Description automatically generated

Để sử dụng các file đã publish này:

A black screen with white text

Description automatically generated



# 2-Lưu ý khi sử dụng ví dụ với .NET 6

## Sự thay đổi từ .NET 3 sang .NET 6 và C# 10

* Bắt đầu từ C# 10 không cần khai báo hàm Main (Vẫn khai báo cũng không sao)
* Toàn bộ các lệnh trong hàm Main cũ được đưa hết vào 1 file (gọi là top-level, không thuộc namespace và class nào)
* Trình biên dịch sẽ tự động hiểu đây là mã lệnh của hàm Main và tự sinh ra hàm main

Cũ Mới

A screen shot of a computer program

Description automatically generated A computer code with text

Description automatically generated

## Một số điểm cần lưu ý:

* Chỉ có 1 file chứa các top-level statement

A screenshot of a computer program

Description automatically generated

* Nếu một lệnh dotnet sử dụng các template truyền thống để phát sinh các bộ khung dự án, khi thực hiện lệnh hãy cho thêm tham số --framework net5.0, trên máy cũng cần cài đặt .NET 5 SDK



* Mô hình hosting mới:
  + Hợp nhất code của Startup.cs và Program.cs thành 1 file top-level là Program.cs

# 3-Biến, kiểu dữ liệu và nhập xuất

## Biến

* Biến là một chuỗi tên trỏ đến ô nhớ trong máy tính, thông qua biến chúng ta có thể lưu giá trị vào vào nó (lưu giá trị vào bộ nhớ) hay đọc giá trị lưu trong bộ nhớ thông qua tên biến

## Khai báo biến

Khởi tạo biến:

<Kiểu dữ liệu> <Tên biến>;

Khai báo biến:

<Kiểu dữ liệu> <Tên biến> = <giá trị ghi cho biến>;

## Quy tắc đặt tên biến

* Có thể sử dụng ký tự a-z hoặc A-Z Bảng các key trong C#
* Có thể sử dụng số 0 – 9A screenshot of a computer screen

  Description automatically generated
* Có thể sử dụng dấu \_
* Không được bắt đầu bằng ký tự số
* Phân biệt chữ hoa và chữ thường
* Không đặt tên biến trùng với các key trong C#

## Ví dụ về đặt tên biến:

A black background with text and symbols

Description automatically generated

## Kiểu dữ liệu:

Kiểu dữ liệu C# cơ bản, định nghĩa sẵn (buil-in) trong C# cơ bản có:

* int  kiểu số nguyên *(có dấu, dùng 32 bit biểu diễn, từ -2,147,483,648 đến 2,147,483,647)*
* sbyte  kiểu số nguyên *(có dấu, dùng 8 bit biểu diễn, từ -128 đến 127)*
* byte  kiểu số nguyên *(không dấu, dùng 8 bit biểu diễn, từ 0 đến 255)*
* short  kiểu số nguyên *(có dấu, dùng 16 bit biểu diễn, từ -32,768 đến 32,767)*
* ushort  kiểu số nguyên *(không dấu, dùng 16 bit biểu diễn, từ 0 đến 65,535)*
* long  kiểu số nguyên *(có dấu, dùng 64 bit biểu diễn, từ -9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807)*
* ulong  kiểu số nguyên *(không dấu, dùng 64 bit biểu diễn, từ 0 đến 18,446,744,073,709,551,615)*
* float  Kiểu số thực chấm động *(dùng 32 bit biểu diễn phù hợp số có bảy chữ số, độ chính xác số dấu chấm động 1.5 × 10−45 đến 3.4 × 1038)*
* double  Kiểu số thực chấm động *(dùng 64 bit biểu diễn)*
* decimal  Kiểu số thực chấm động *(dùng 128 bit biểu diễn)*
* char  một ký tự *(dùng 16 bit biểu diễn ký tự Unicode)*
* bool  kiểu logic *(chỉ nhận giá trị false hoặc true)*
* string  chuỗi (xâu) ký tự *(tập hợp các ký tự theo thứ tự - một văn bản text)*
* object  đối tượng, biểu diễn các đối tượng C#, nó là kiểu cơ sở - mọi đối tượng C# đều kế thừa từ kiểu này.

Kiểu nguyên thủy:

* sbyte, byte, short, ushort, int, long, float, double, decimal, char, bool

Kiểu tham chiếu:

* object

## Xuất dữ liệu ra console

Console.WriteLine(<Dữ liệu>); Xuất dữ liệu và tự động xuống dòng

Console.Write(<Dữ liệu>); Xuất dữ liệu nhưng không xuống dòng

## Nhập dữ liệu:

Console.ReadKey(); // Người dùng nhập 1 ký tự bất kỳ sẽ thoát khỏi ReadKey

Console.ReadLine(); // Người dùng khi nào ấn Enter thì sẽ thoát khỏi ReadLine

## Format String

A white background with black text

Description automatically generated A math equation with black text

Description automatically generated

## Ép kiểu:

## Chuyển đổi kiểu ngầm định:

Thực hiện khi chuyển kiểu từ:

* Kiểu dữ liệu nhỏ sang kiểu dữ liệu lớn hơn
* Chuyển từ lớp con đến lớp cha

Int i = 10;

Long l = 100;

## Chuyển đổi tường minh:

Cú pháp:

**(<kiểu dữ liệu>) <biến cần ép kiểu>**

Int i = 300;

Byte b = (byte)i;

## Sử dụng phương thức, lớp hỗ trợ sẵn => Parse(), TryParse(), Convert

Parse và TypeParse dùng để ép kiểu chữ sang số

String s = “10”;

Int i = int.Parse(s);

Int Result;

Bool isSuccess;

String data1 = “10”, data2 = “HHH”

isSuccess = int.TryParse(data1, out Result);

## Khai báo ngầm định:

Var <tên biến> = <giá trị>;

Lưu ý cần đặt giá trị khi khởi tạo, và nó sẽ tự hiểu kiểu dữ liệu trong lần khai báo đầu tiên

Trong C#, **var** được sử dụng để khai báo biến cục bộ với kiểu được suy luận từ biểu thức khởi tạo của nó.

**var** phải được sử dụng trong một khối lệnh cục bộ hoặc script code, hoặc khi cố gắng sử dụng nó trong một ngữ cảnh không cho phép suy luận kiểu (như là trong khai báo biến toàn cục hay thuộc tính của một class).

## Hằng

Const <kiểu dữ liệu> <tên biến> = <giá trị>

Lưu ý cần đặt giá trị khi khởi tạo, và hằng số không thể bị ghi đè

# 4-Toán tử số học, gán

## Toán tử số học

### Các toán tử số học:

* Toán tử cộng: +
* Toán tử trừ: -
* Toán tử nhân: \*
* Toán tử chia lấy dư: %
* Toán tử chia lấy nguyên: /

### Để chia ra số thực

Chỉ cần ép kiểu 1 số hạng bất kỳ thành số thực

int a = 1;

int b = 2;

float result = (float)a / b;

### Thứ tự ưu tiên

* Giống như trong toán học: nhân chia trước cộng trừ sau, trong ngoặc trước ngoài ngoặc sau
* Mức độ ưu tên ngang nhau sẽ tính từ trái sang phải
* Mức độ ưu tiên: Trong ngoặc -> Căn bậc 2, lũy thừa -> nhân, chia -> cộng, trừ

## Toán tử gán

### Các toán tử gán:

* = += -= \*= /= %=

## Toán tử tăng ++ và giảm -- trong C#

* Cả 2 toán tử này đêu tăng 1 đơn vị haowjc giảm một đơn vị cho x
* Nhưng trong biểu thức

int x = 1;

int y = 2 + x++; -> y = 2 + x sau đó x = x + 1; -> kết quả x = 2, y = 3

int y = 2 + ++x; -> x = x + 1 sau đó y = 2 + x; -> kết quả x = 2, y = 4

# 5-Toán tử so sánh, logic và lệnh if, switch

# 6-Vòng lặp

# 7-Array

## Array torng C#

## Khai báo mảng:

<kieu du lieu>[] <ten mang>;

## Khởi tạo Mảng

Cần định nghĩa só lượng phần tử của mảng khi khởi tạo

<kieu du lieu>[] <ten mang>;

<ten mang> = new <kieu du lieu>[<so luong phan tu>];

<kieu du lieu>[] <ten mang> = new <kieu du lieu>[<so luong phan tu>];

<kieu du lieu>[] <ten mang> = new <kieu du lieu>[<so luong phan tu>] {<phan tu 1>, …};

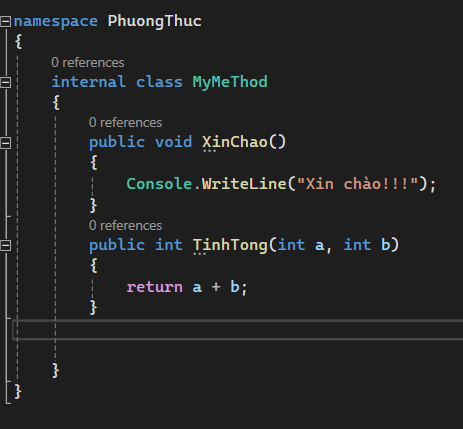
<kieu du lieu>[] <ten mang> = {<phan tu 1>, <phan tu 2>,…};

# 8-Phương Thức

* Phương thức (Method) gần giống với function trong lập trình hướng thủ tục
* Là một khối mã nhằm thực thi một chức năng, một tác vụ nào đó
* Phương thức có tính tái sử dụng

**Phương thức có thể thuộc một class nào đó khoặc không nhưng khi khai báo phương thức thì phải khai báo trong class**

## Khai báo phương thức:

Có kiểu trả về: 

<access modifier> <return type> <method name>(<param>,...) {

logic

return ...

}

Không có kiểu trả về:

<access modifier> void <method name>(<param>,...) {

logic

}

## Phương thức tĩnh

* Là phương thức có thể gọi mà không cần phải khởi tạo đối tượng từ class

Cú pháp:

Có kiểu trả về:

<access modifier> static <return type> <method name>(<param>,...) {

logic

return ...

}

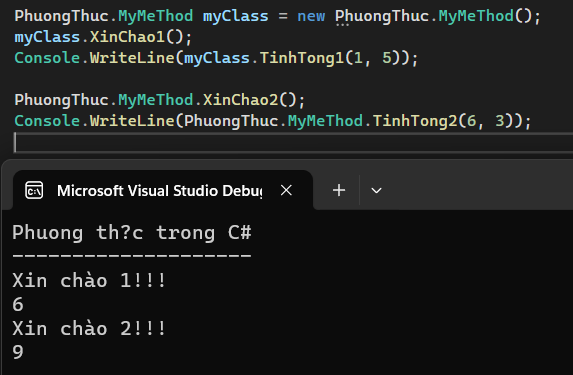
Không có kiểu trả về:

<access modifier> static void <method name>(<param>,...) {

logic

}

## Gọi phương thức

Đối với phương thức static

<namespace>.<class>.<method name>(<param>);

Đối với phương thức không static

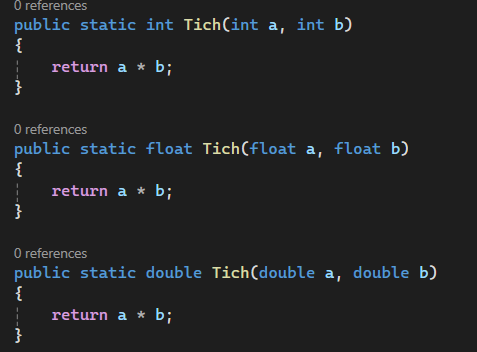
// Khởi tạo đối tượng

// Gọi phương thức thông qua đối tượng

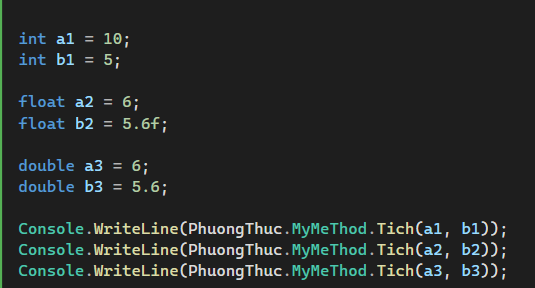
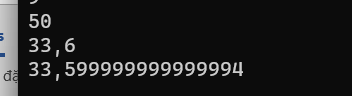
## Overload phương thức

Là việc một phương thức có thể có nhận loại parameter khác nhau

Để overload 1 phương thức ta khai báo 1 phương thức có cùng tên, cùng kiểu tham số



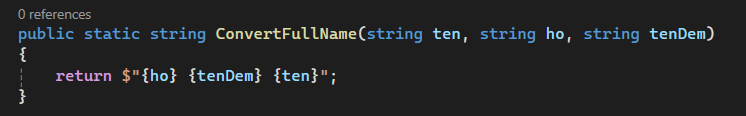
Khi đó ta gọi phương thức theo tham số kiểu nào sẽ trả về kết quả tương ứng

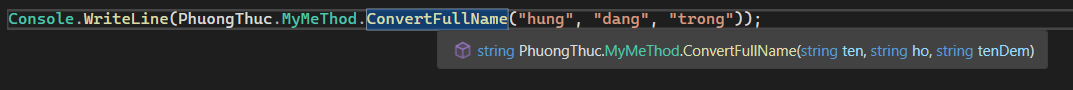
 

## Truyền tham số với tên trong phương thức

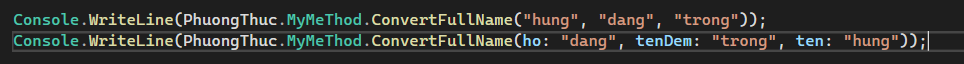
Bình thường chúng ta truyền tham số vào phương thức theo thứ tự khai báo nhưng ta có thể truyền tham số theo tên

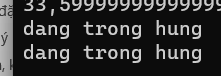
Ví dụ:





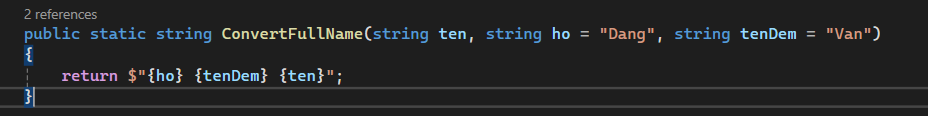
Ta có thể viết như sau:

 Kết quả:



## Tham số mặc định:

Ta có thể sử dụng tham số mặc định trong trường hợp người dùng không truyền đủ tham số như sau:



Lưu ý: giá trị mặc định phải luôn được đặt ở cuối cùng

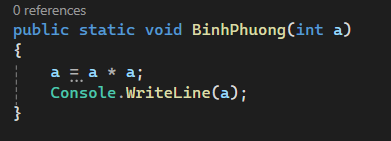


Kết quả:



## Tham số tham chiếu và tham trị (ref & out keyword)

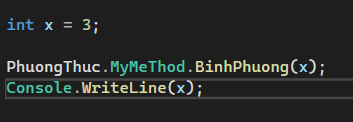
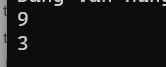
Giả sử ta có 1 phương thức như sau:



Sau đó ta có 1 biến x như sau:



Tiếp tục ta gọi phương thức BinhPhuong và truyền x vào làm tham số:

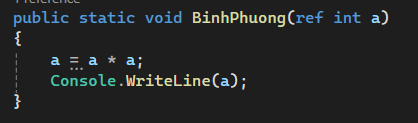
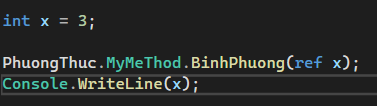
 

Ta nhận thấy rằng x vẫn bằng 3 sau khi được truyền qua phương thức BinhPhuong tức là x vẫn giữ nguyên. Đây gọi là truyền tham số kiểu tham trị

Khi chúng ta sử dụng 1 biến làm tham số cho một phương thức thì giá trị của biến được dung ffeer khởi tạo ra một biến cục bộ trong phương thức, như vậy biến cục bộ a và biến x là 2 biến khác nhau, giá trị trong biến x chỉ được đụng để khởi tạo và gán vào giá trị cục bộ của a mà thôi

Trong thực tế nhiều lúc chúng ta cần truyền tham số này dưới dạng tham chiếu tức là nó không phải là khởi tạo ra biến cục bộ nữa mà nó sẽ tham chiếu đến ô nhớ của biến được trueyenf vào làm tham số ở đây là x điều này dẫn đến việc a thay đổi sẽ dẫn đến thay đổi của x

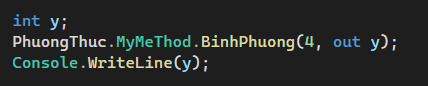
Chúng ta khai báo như sau:

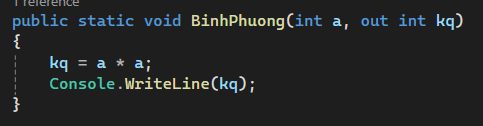
 

Kết quả:



Ngoài sử dụng ref chúng ta có thể dùng out một cách tương tự chỉ là với ref ta cần phải khởi tạo giá trị trước còn out thì không cần khởi tạo trước



Kết quả: 

# 9-Chuyển số thành chữ

# 10-Class

## Cú pháp tạo class:

<Access Modifiers> **class** **Class\_Name** {

// khai báo các thành viên dữ liệu (thuộc tính, biến trường dữ liệu)

// khai báo các thành viên hàm (phương thức)

}

## Access modifier

* Public Có thể truy cập từ bất kỳ đâu
* Internal (Mặc định) Chỉ truy cập được trong cùng assembly (dll, exe)
* Protected Chỉ truy cập trong nội bộ class hoặc class kế thừa
* Private (Mặc định cho thuộc tính trong class) Chỉ có thể truy cập từ các thành viên của class chứa nó

Khai báo class

<Tên class> <tên instance>;

Khởi tạo class

<Tên class> <tên instance> = new <Tên class>(<parameter>);

Một class có thể có nhiều phương thức khởi tạo khác nhau

A screen shot of a computer program

Description automatically generated

Tùy vào cách khởi tạo instance như thế nào mà sẽ có một constructor function tương ứng

## Trường dữ liệu

Mặc định access modifier của trường dữ liệu là private nếu ta không điền gì

A screen shot of a computer

Description automatically generated

Chúng ta có thể khởi tạo luôn giá trị cho trường dữ liệu:

A black background with green and white text

Description automatically generated

Để truy cập vào trường dữ liệu ta sử dụng toán tử chấm (.)

A computer screen with text and images

Description automatically generated

Ở đây có lỗi vì phạm vi truy cập của trường dữ liệu là private

A black screen with colorful text

Description automatically generated

Kết quả:

A screen shot of a computer

Description automatically generated

Đối với các trường dữ liệu private chúng ta không thể truy cập trực tiếp nhưng vẫn có thể thao tác được với nó thông qua các phương thức (không phải private)

## Thuộc tính

Thuộc tính gần giống như trường dữ liệu nhưng nó có cơ chế accessor (get, set)

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Kết quả:

A black screen with white text

Description automatically generated

Thuộc tính dùng để truy cập hoặc thiết lập 1 dữ liệu của class thông qua 2 phương thức get và set

Một thuộc tính không nhất thiết phải có cả get và set

Chúng ta có thể tạo tự động get và set

A black and white text

Description automatically generated

A black background with white text

Description automatically generated

Kết quả:

A black screen with white text

Description automatically generated

## Sự khác nhau giữa field và property

Trong ngữ cảnh của lập trình hướng đối tượng (OOP) và trong C#, trường dữ liệu (field) và thuộc tính (property) là hai khái niệm khác nhau như sau:

1. **Trường Dữ Liệu (Field):**
   * Là biến được khai báo trong một lớp để lưu trữ dữ liệu của đối tượng.
   * Trường dữ liệu thường được khai báo với từ khoá **private** hoặc **protected** để giữ cho dữ liệu ẩn đi và tránh trực tiếp truy cập từ bên ngoài lớp.
   * Trường dữ liệu không có logic đặc biệt khi đọc hoặc ghi giá trị.
   * A black screen with white text

     Description automatically generated

1. **Thuộc Tính (Property):**
   * Là một thành phần của lớp cung cấp một cách để đọc hoặc ghi giá trị của một trường dữ liệu.
   * Thuộc tính thường được sử dụng để kiểm soát việc truy cập đến trường dữ liệu và thêm logic vào quá trình đọc hoặc ghi giá trị.
   * Thuộc tính có thể có phương thức đọc (get) và phương thức ghi (set).
   * A screen shot of a computer program

     Description automatically generated

Khi sử dụng thuộc tính, bạn có thể thêm logic kiểm tra giá trị, kiểm soát quyền truy cập, và thực hiện các hoạt động phức tạp hơn so với việc trực tiếp truy cập trường dữ liệu. Sử dụng thuộc tính giúp encapsulation và tăng tính linh hoạt trong quản lý dữ liệu của lớp.

## Hủy đối tượng

Là việc thu hồi bộ nhớ đã cấp phát cho instance khi khởi tạo từ class

Phương thức hủy như sau:

A computer screen shot of a program

Description automatically generated

Phương thức hủy xảy ra khi không còn thể hiện nào tham chiếu tới đối tượng

A screen shot of a computer

Description automatically generated

Lưu ý khi ta thức hiện thu hồi bộ nhớ chúng ta sẽ không biết lúc nào .net sẽ thu hồi, .net sẽ thu hồi khi nào nó thấy thiếu bộ nhớ

Chúng ta có thể thử điều này bằng cách thử tạo ra thật nhiều đối tượng từ class ( > 1.000.000)

Tìm hiểu thêm về dispose, lớp Idispose

# 11-Phương thức khởi tạo

## Phương thức khởi tạo, cú pháp khai báo phương thức khởi tạo C#

### **Giới thiệu về khởi tạo (Constructor) lớp trong C#**

Khi một đối tượng của lớp (class) hay cấu trúc (struc) được tạo, thì phương thức khởi tạo của lớp được gọi. Phương thức khởi tạo có tên cùng với tên lớp, có thể có nhiều phương thức khởi tạo nhưng mỗi phương thức có các tham số khác nhau. Phương thức khởi tạo là nơi bạn khởi tạo các giá trị dữ liệu nó giúp cho việc đọc code dễ hơn và sử dụng lớp một cách mềm dẻo hơn.

Nếu bạn không xây dựng phương thức khởi tạo nào cho lớp, C# sẽ tạo ra một phương thức khởi tạo ngầm định, nó thiết lập giá trị các thành viên dữ liệu nhận giá trị mặc định (tùy theo kiểu dữ liệu).

Về phần sử dụng phương thức khởi tạo đã giới thiệu tại: [Phương thức khởi tạo](https://xuanthulab.net/su-dung-lop-thuoc-tinh-trong-c-tao-va-su-dung-doi-tuong-lop.html#Constructors), phần này trình bày hệ thống và kỹ hơn.

### **Cú pháp khai báo phương thức khởi tạo**

Phương thức khởi tạo có tên giống tên lớp, không có kiểu trả về, hãy xem ví dụ sau:

**class** **Product**

{

**private** string name;

**private** decimal price;

// Khai báo phương thức khởi tạo với 2 tham số

**public** **Product**(string nameproduct, decimal priceproduct)

{

name = nameproduct;

price = priceproduct;

}

// Khai báo phương thức khởi tạo không tham số

**public** **Product**()

{

name = "Không tên";

price = 0;

}

// Thuộc tính Name lấy hoặc thiết lập tên sản phẩm

**public** string Name

{

**set** { name = **value**;}

**get** { **return** name;}

}

}

Bạn cũng lưu ý, trong lớp có các phương thức mà thực hiện một biểu thức đơn giản, thì bạn có thể gán ngay biểu thức vào tên phương thức bằng ký hiệu =>, kể cả setter và getter, ví dụ:

**class** **Category**

{

**private** string categoryname;

// Dùng thân biểu thức cho phương thức khởi tạo

**public** **Category**(string nameofCategory) => categoryname = nameofCategory;

**public** String Name

{

// Dùng thân biểu thức cho setter, getter

**set** => categoryname = **value**;

**get** => categoryname;

}

}

Source code: [CS009\_Event (Git)](https://github.com/xuanthulabnet/learn-cs-netcore/tree/master/CS009_Event), hoặc tải [cs009-event](https://github.com/xuanthulabnet/learn-cs-netcore/releases/tag/cs009-event)

## **Sử dụng phương thức khởi tạo trong C#**

Khi tạo đối tượng của lớp bằng toán tử new thì nó sẽ tạo đối tượng và thi hành phương thức khởi tạo tương ứng phù hợp với tham số.

// Tạo đối tượng, không truyền tham số nên

// nó gọi phương thức khởi tạo không tham số

Product product1 = **new** Product();

Console.WriteLine(product1.Name); // Không tên

Product product2 = **new** Product("Laptop", 123);

Console.WriteLine(product2.Name); // Laptop

## **Phương thức khởi tạo của lớp cơ sở C#**

Trong C# các lớp có tính kế thừa, một lớp có thể kế thừa lại các thuộc tính, phương thức từ lớp khác (lớp cơ sở), phần này nói ở [Kế thừa lớp trong C#](https://xuanthulab.net/tinh-ke-thua-trong-lap-trinh-c-c-sharp.html), lưu ý là nếu lớp cha (cơ sở) có xây dựng phương thức khởi tạo, thì cần chỉ định rõ gọi phương thức khởi tạo nào của lớp cơ sở (truyền tham số) khi phương thức khởi tạo lớp con thi hành. Thực hiện điều này bằng cách dùng từ khóa base ở khai báo tên phương thức như ví dụ sau:

**class** **CategoryMobile** : **Category**

{

**private** string description;

// Khi phương thức khởi tạo này được gọi, nó gọi phương thức khởi tạo có

// một tham số của lớp cơ sở (Category) trước, rồi mới thi hành các code

// trong thân của phương thức khởi tạo này

**public** **CategoryMobile**(string nameofCategory, string mota) : **base**(nameofCategory)

{

description = mota;

}

**public** string Description

{

**set** => description = **value**;

**get** => description;

}

}

Khi sử dụng, chay đoạn code

CategoryMobile cat1 = **new** CategoryMobile("Điện thoại", "Danh mục các loại điện thoại");

Console.WriteLine(cat1.Name);

Console.WriteLine(cat1.Description);

// In ra

// Điện thoại

// Danh mục các loại điện thoại

## **Phương thức khởi tạo riêng tư C#**

Khi xây dựng một số lớp đặc biệt, nhất là những lớp tiện ích chỉ chứa thành viên tĩnh, bạn mong muốn chỉ ra một cách tường minh lớp này không được phép tạo đối tượng, lớp này không được phép kế thừa thì hãy cho vào một phương thức khởi tạo không tham số và chỉ ra trạng thái truy cập là private, ví dụ:

**class** **MyLib**

{

**public** **static** double PI = 3.14;

**private** **MyLib**()

{

}

}

Lúc này nếu dùng lớp khởi tạo đối tượng thì sẽ lỗi

MyLib my1 = **new** MyLib(); // Lỗi

Kể cả dùng làm lớp cơ sở cũng lỗi

## **Phương thức khởi tạo tĩnh C#**

Bạn có thể xây dựng một phương thức khởi tạo không tham số có chỉ thị truy cập là static, phương thức khởi tạo này dùng để khởi tạo các thành viên dữ liệu tĩnh, nó tự động gọi khi truy cập một thành viên dữ liệu tĩnh lần đầu. Ví dụ:

**class** **MyColorCode**

{

**public** **static** string color\_primary;

**public** **static** string color\_success;

**public** **static** string color\_danger;

**public** **static** string color\_warning;

**public** **static** string color\_info;

// Phương thức khởi tạo tĩnh, được gọi khi lần đầu truy cập một thành viên tĩnh

**static** **MyColorCode**()

{

Console.WriteLine("Static MyColorCode Contructor Call");

color\_danger = "Red";

color\_info = "Cyan";

color\_primary = "Navy";

color\_success = "Green";

color\_warning = "Yellow";

}

}

Đoạn mã sử dụng kiểm tra:

Console.WriteLine(MyColorCode.color\_danger);

Console.WriteLine(MyColorCode.color\_info);

Kết quả là

Static MyColorCode Contructor Call

Red

Cyan

# 12-Kiểu giá trị và kiểu tham chiếu

Trong C# có 2 kiểu dữ liệu là kiểu giá trị và kiểu tham chiếu:

* Khi ta tạo 1 biến kiểu giá trị máy tính sẽ tạo và lưu giá trị này trong một ô nhớ trong vùng nhớ, vì vậy khi ta thao tác với dữ liệu kiểu này, chúng ta sẽ thao tác trực tiếp với ô nhớ này
* Khi ta tạo 1 biến kiểu tham chiếu, máy tính sẽ tạo ra một ô nhớ và lưu lại địa chỉ của nó trong ô nhớ này, vì vậy tất cả các biến khác được tham chiếu đến địa chỉ của ô nhớ này nếu thay đổi sẽ thay đổi toàn bộ dữ liệu của các biến tham chiếu khác
* Kiểu tham trị khi tạo ra được lưu trữ ở bộ nhớ stack
* Kiểm tham chiếu khi được tạo ra sẽ được lưu trữ ở bộ nhớ heap ( đây là một bộ nhớ lớn dùng để lưu trữ những cấu trúc dữ liệu phức tạp và kích thước lớn)

## Kiểu giá trị

* Kiểu giá trị bao gồm:
  + Kiểu số nguyên (int, long, byte, short,…)
  + Kiểu số thực (double, float,…)
  + Kiểu boolean
  + Kiểu char
  + Kiểu cấu trúc: struct
  + Kiểu liệt kê: enum
  + Kiểu Tuple

## Kiểu tham chiếu

* Kiểu tham chiếu bao gồm:
  + Class
  + Delegate
  + Interface
  + Dynamic
  + Object
  + String

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screen shot of a computer

Description automatically generated

# 13-Chuỗi ký tự trong C#

Chuỗi ký tự thực tế ta đã làm việc với nó ở các ví dụ trước, chuỗi là một tập hợp các ký tự sắp xếp có vị trí, nó chỉnh là một mảng các ký tự, kiểu dữ liệu chuỗi đó là string, lớp biểu diễn các chuỗi là System.String

Bạn có thể khai báo, khởi tạo chuỗi với kiểu string, cũng như thực hiện các phép toán trên chuỗi...

string sExample = "Xin chào"; // Khai báo và khởi tạo chuỗi

sExample += " các bạn"; // Nối chuỗi +=, trả về "Xin chào các bạn"

sExample = sExample + "!"; // Nối chuỗi +, trả về "Xin chào các bạn!"

Chuỗi như là mảng mà phần tử mảng là các ký tự, nên có thể truy cập phần tử mảng bằng indexer để đọc ký tự:

char c = sExample[1]; // c= 'i'

Chuỗi nhập vào viết trong cặp nháy kép "", chèn vào chuỗi các ký tự đặc biệt ta dùng ký hiệu \: ví dụ \\ là ký tự \, \" là ký tự ", \n là ký tự xuống dòng, \r là ký tự về đầu dòng ...

### **Viết chuỗi nguyên bản với ký hiệu @ trong C#**

Khi viết chuỗi trong cặp dấu nháy kép "", thì các ký tự đặt biệt được xử lý với ký hiệu \, ví dụ nếu bạn viết:

string s = "C:\\Abc\\xyz";

// Nếu viết string s = "C:\Abc\xyz"; sẽ lỗi

Thì nội dung thực tế của chuỗi là C:\Abc\xyz

Nếu muốn viết chuỗi cố định, nội dung nguyên bản - cho biết sẽ không dùng \ để xử lý ký tự đặc biệt, thì thêm @ vào đầu chuỗi:

string s = @"Ký tự \ được dùng để chèn ký tự đặc biệt như \n, \r";

Bằng ký hiệu @ chuỗi viết thế nào thì nội dung thực tế sẽ như vậy, ngoại trừ hai ký tự "" chuyển thành một ký tự ""

string s = "Anh ấy nói, ""Đây là C#"""; //~ Anh ấy nói "Đây là C#"

Ngoài ra bạn có thể viết chuỗi trên nhiều dòng với ký hiệu @

string s = @"Xin chào các bạn

Tôi đang học C#";

### **Chèn thêm biểu thức vào chuỗi với ký hiệu $ trong C#**

Khi viết chuỗi có ký tự $ phía trước, thì trong chuỗi đó có thể chèn các biểu thức vào chỗ có cặp {} : {biểu-thức}

int a = 10;

int b = 2;

string s = $"Kết quả {a}/{b} là {a/b}"; // "Kết quả 10/2 là 5"

Ngoài ra bạn có thể căn lề, định dạng số, ngày tháng ... tương tự như chuỗi định dạng

Console.WriteLine($"{"VòngLặp",10} {"Chẵn/Lẻ", -5}");

**for** (int i = 8; i < 15; i++)

{

string chanle = (i%2 == 0) ? "Chẵn" : "Lẻ";

Console.WriteLine($"{i,10} {chanle, -5}");

}

Kết quả:

VòngLặp Chẵn/Lẻ

8 Chẵn

9 Lẻ

10 Chẵn

11 Lẻ

12 Chẵn

13 Lẻ

14 Chẵn

Ký hiệu {i,10} là để ra một khoảng trống 10 ký tự, chèn giá trị i vào khoảng trống đó với căn lề bên phải. Ký hiệu {chanle, -5} là đề ra 5 khoảng trống chèn giá trị và căn lề trái.

## **Một số phương thức làm việc với chuỗi C#**

Ví dụ có hai biến chuỗi, thực hiện một số phép toán:

string stringA = "Xin chào,";

string stringB = "các bạn!";

|  |  |
| --- | --- |
| **method** | **Sử dụng** |
| Concat | phương thức tĩnh, nối các chuỗi liệt kê ở tham số lại với nhau  string s = String.Concat(stringA, stringB); // s = "Xin chào,các bạn!" |
| Format | Convert các đối tượng thành chuỗi và chèn chúng vào chuỗi định dạng được chỉ ra, cú pháp cơ bản  String.Format(stringFormat, arg0, arg1 ...)  Trong đó stringFormat là chuỗi định dạng, chứa mẫu và và vị trí mà arg0, arg1 ... sẽ chèn vào.  string s = String.Format("Chào {0}, {0} ơi, hôm nay ngày {1} rồi!", "Nam", DateTime.Now.Day);  // s = "Chào Nam, Nam hôm nay ngày 20 rồi!"  Nhìn vào, chuỗi định dạng chỗ nào có {0} sẽ được chèn bởi arg0("Nam"), chỗ nào có {1} sẽ được chèn bởi arg1 (20)  Tại các vị trí cần chèn {} có thể định dạng dữ liệu chèn vào, chi tiết xem tại [Formatting](https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types?view=netframework-4.8) |
| IndexOf | Tìm vị trí (đầu tiên) của ký tự hoặc chuỗi ký tự trong chuỗi |
| LastIndexOf | Tìm vị trí (cuối) của ký tự hoặc chuỗi ký tự trong chuỗi |
| Insert | Tạo chuỗi = chèn chuỗi này vào trong chuỗi khác, vị trí chèn cần chỉ ra  **var** s = stringA.Insert(8, " tất cả"); // "Xin chào tất cả," |
| PadLeft | Tạo chuỗi mới từ chuỗi cũ, độ dài chuỗi mới chỉ ra - nếu độ dài chuỗi mới lớn hơn chuỗi cũ thì các ký tự phía đầu được chèn khoảng trắng hoặc ký tự chỉ định.  string s1 = "Abc";  string s2 = s1.PadLeft(6); // " Abc"  string s3 = s1.PadLeft(6, '\*'); // "\*\*\*Abc" |
| PadRight | Tương tự PadLeft nhưng chèn khoảng trắng bên phải |
| Replace | Tìm và thay thế trong chuỗi  **var** s = stringA.Replace("chào", "CHÀO"); // "Xin CHÀO," |
| Split | Trả về mảng các chuỗi con được chia từ chuỗi gốc bởi ký tự chia chỉ định, chuỗi chia chỉ định  **var** s = "Nguyễn Văn A".Split(' '); // s tương đương mảng {"Nguyễn","Văn", "A"} |
| ToLower | Sinh chuỗi mới bằng cách chuyển các ký tự thành chữ thường |
| ToUpper | Sinh chuỗi mới bằng cách chuyển các ký tự thành chữ in |
| Trim | Sinh chuỗi mới bằng cách loại bỏ khoảng trắng (hoặc chỉ định) ở đầu và cuối |
| Substring | Lấy ra chuỗi con từ chuỗi chính - chuỗi con lấy từ vị trí chỉ ra đến cuối hoặc theo độ dài  string s = stringA.Substring(4); // s = "chào,"  string x = stringA.Substring(0, 3); // s = "Xin" (dài 3) |

## **Sử dụng StringBuilder**

Khi sử dụng biến kiểu string để thực hiện các thao tác nhằm mục đích cuối cùng thu được chuỗi theo yêu cầu, trong quá trình đó bạn có thể sử dụng nhiều biến kiểu string, để phục vụ các phép toán như nối chuỗi, tìm kiếm, thay thế ... Mỗi khi khởi tạo một biến kiểu string, bạn đã cấp phát một lượng bộ nhớ để lưu trữ chuỗi - thường thì bộ nhớ này nhiều hơn những gì bạn cần.

Để thi hành tối ưu hơn về tốc độ, về sử dụng bộ nhớ có thể dùng tới đối tượng StringBuilder ở namespace System.Text (thêm vào đầu file using System.Text;)

StringBuilder stringBuilder = **new** StringBuilder();

stringBuilder.Append("Xin chào các bạn - xuanthulab.net");

stringBuilder.Replace("Xin chào", "XIN CHÀO");

Console.WriteLine(stringBuilder); // Out: XIN CHÀO các bạn - xuanthulab.net

Một số phương thức khi sử dụng StringBuilder

|  |  |
| --- | --- |
| **method** | **Sử dụng** |
| Append | Nối một đối tượng (chuỗi, số ...) vào StringBuilder |
| AppendFormat | Nối vào một chuỗi sinh ra theo StringFormat |
| Insert | Chèn vào StringBuilder một chuỗi ở vị trí chỉ ra |
| Remove | Loại bỏ một lượng ký tự, bắt đầu từ vị trí do chỉ định |
| Replace | Tìm và thay thế |
| ToString | Trả về chuỗi (String) |

## **Dùng biểu thức chính quy với chuỗi trong C#**

Khi làm việc với chuỗi, những tác vụ phức tạp muốn nhanh chóng có thể áp dụng kỹ thuật [biểu thức chính quy](https://xuanthulab.net/bieu-thuc-chinh-quy-regexp.html) vì C# hỗ trợ các lớp để làm việc với RegExp ([regularexpressions](https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=netframework-4.8)), phần này trình bày ở một bài riêng

Ví dụ xây dựng lại hàm tìm kiếm địa chỉ email có trong một văn bản, ví dụ này đã từng xây dựng trong PHP: [RegEx tìm email](https://xuanthulab.net/ham-preg-match-preg-match-all-trong-php.html#example)

String text = @"Đây là địa chỉ

email userabcguest@xuanthulab.net.vn và

xyz@gmail.com cần trích xuất";

String pattern = @"(([^\s.]+)@((.[^\s]+)(\..[^\s]+)))";

Regex rx = **new** Regex(pattern);

// Tìm kiếm.

MatchCollection matches = rx.Matches(text);

// In thông báo tìm kiếm.

Console.WriteLine("Tìm thấy {0} email trong:\n\n {1}\n\n",

matches.Count,

text);

// Xuất kết quả email tìm được

**foreach** (Match match **in** matches)

{

GroupCollection groups = match.Groups;

Console.WriteLine("{0}", groups[0].Value);

}

Kết quả chạy

Tìm thấ**y** 2 email trong:

Đâ**y** là địa chỉ

email userabcguest@xuanthulab.net.vn và

xyz@gmail.com cần **tr**ích xuất

userabcguest@xuanthulab.net.vn

xyz@gmail.com

Mã nguồn [CS012\_String(Git)](https://github.com/xuanthulabnet/learn-cs-netcore/tree/master/CS012_String), hoặc tải [ex012-1](https://github.com/xuanthulabnet/learn-cs-netcore/releases/tag/ex012-1)

# 14-Struc và Enum

## Struct

* Struct là kiểu dữ liệu cấu trúc
* Struct gần giống như class (Có trường dữ liệu, phương thức, constructor…)
* Struct là kiểu dữ liệu giá trị

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program code

Description automatically generated

A black background with white text

Description automatically generated

* Nếu struct có constuctor thì khi khai báo cần có từ khóa new
* Bắt buộc phải khởi tạo giá trị cho các trường dữ liệu khi khởi tạoA computer screen shot of a program code

  Description automatically generated
* A black background with white text

  Description automatically generated
* A close-up of white text

  Description automatically generated

## Enum

* Enum là kiểu dữ liệu liệt kê

Giả sử ta có bài toán sau:

A screenshot of a computer

Description automatically generated

Để giải quyết bài toán này ta có thể dùng switch

A computer screen shot of code

Description automatically generated

Kết quả:



Xử lý bằng Enum

A screenshot of a computer

Description automatically generated

A computer screen shot of a code

Description automatically generated

KẾt quả:



# 15-Kế thừa

* Là việc một class kế thừa lại các thuộc tính và phương thức của 1 class khác
* Ví dụ class B kế thừa từ class A
  + Class A được gọi là lớp cha hay lớp cơ sở
  + Class B được gọi là lớp con hay lớp kế thừa

Lớp cha:

internal class PhuongTien

{

// field

string \_name;

int \_vantoc;

// constructor

public PhuongTien()

{

Console.WriteLine("Phuong tien duoc tao");

}

public PhuongTien(string name, int van\_toc)

{

\_name=name;

\_vantoc=van\_toc;

}

// property

public string Name

{

get { return \_name; }

set { \_name = value; }

}

public int VanToc

{

get => \_vantoc;

set => \_vantoc = value;

}

// method

public void dichuyen()

{

Console.WriteLine($"Phuong tien {\_name} dang di chuyen voi van toc la {\_vantoc} km/h");

}

}

Lớp con:

class XeMay : PhuongTien

{

int \_sobanh;

string \_diahinh;

public XeMay()

{

Console.WriteLine("Xe may duoc tao");

}

public XeMay(string name, int van\_toc) :base(name, van\_toc)

{

\_diahinh = "Duong bo";

\_sobanh = 2;

}

}

Lúc này ta cần lưu ý 1 số điểm:

* Đối với phương thức khởi tạo không có tham số, lớp kế thừa chỉ cần viết lại constructor 1 lần nữa:

A computer screen shot of a program code

Description automatically generated A computer screen with text

Description automatically generated

* Đối với phương thức khởi tạo có tham số ta cần sử dụng từ khóa **base**

A computer code with white text

Description automatically generated A black screen with white text

Description automatically generated

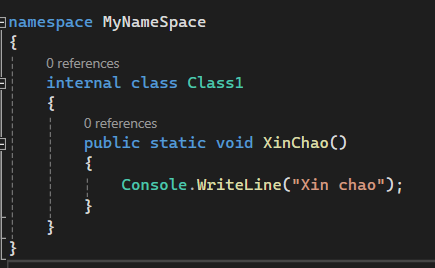
# 16-NameSpace

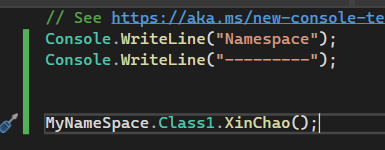
Namespace là cách gộp những thành phần như class, struc, enum, interface,… vào 1 khu vực để có thể dex sử dụng, quản lý và theo dõi

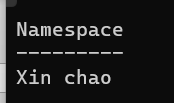
Khai báo 1 namespace:

**Namespace <tên namspace> {}**

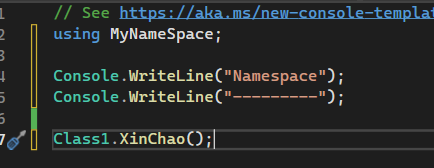
Bên trong 1 namespace chúng ta cũng có thể khai báo các namespace con khác

Ví dụ ta có 1 namesapce như sau: 

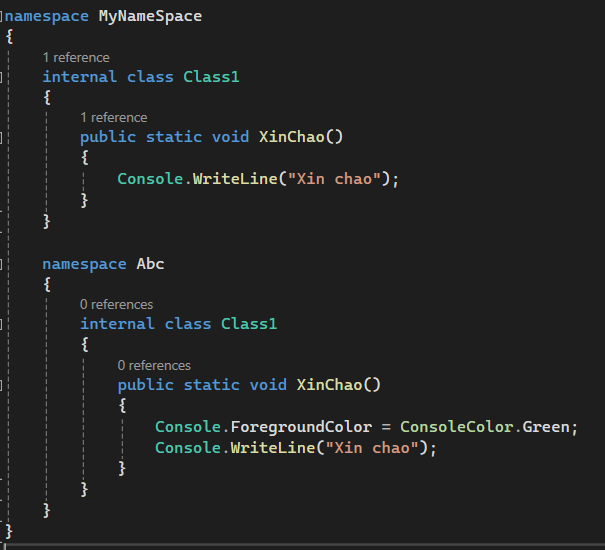
Ta có thể gọi đến phương thức XinChao thông qua namespace này:



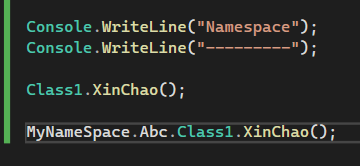
Để không cần gọi qua namespace ta có thể nạp namespace vào bằng cách sử dụng từ khóa using



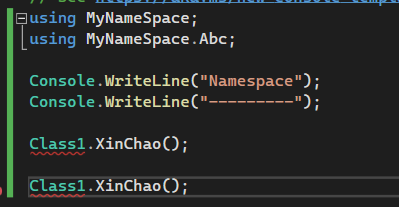
## Namespace con, namespace lồng nhau

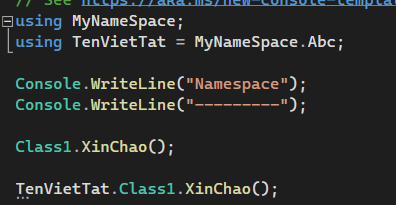


Lúc này để sử dụng phương thức XinChao của namespace Abc ta có thể làm như sau:



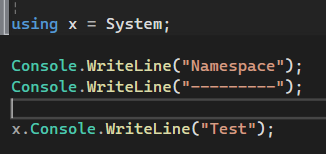
Hoặc

Trong trường hợp này cả 2 namespace Abc và MyNameSpace đều có Class1 vì vậy sẽ bị xung đột code vì hệ thống không biết đnag gọi đến Class1 của namespace nào vì vậy chúng ta cần phải đặt tên cho Name space đó bằng 1 tên viết tắt: trong ví dụ là TenVietTat

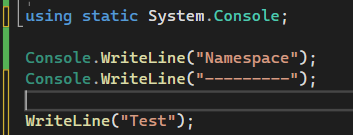


## Sử dụng using static

Hoặc ví dụ ta có thể thay vì gọi Console.WriteLine() trong namespace System ta có thể làm như sau:



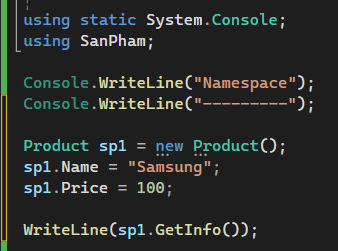
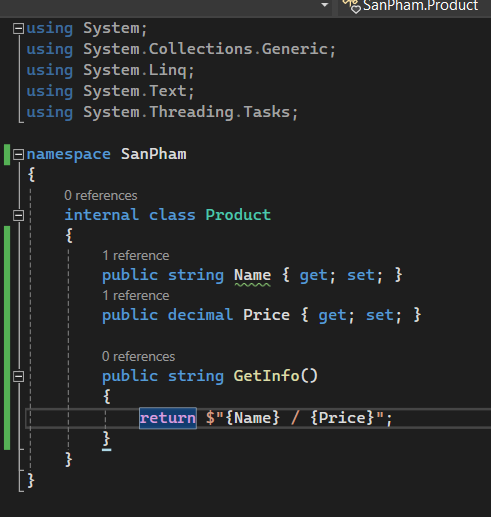
Trong class Console có các phương thức tĩnh như WriteLine,.. ta cũng có thể nạp các phương thức tĩnh như sau:

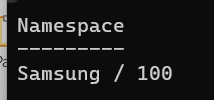


Khi ta nạp các phương thức tĩnh của class ta có thể gọi thẳng đến phương thức mà không cần thông qua class và namespace

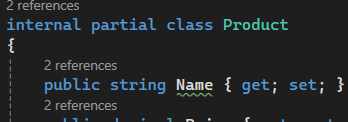
## Sử dụng partial để chia nhỏ mã nguồn

Giả sử ta có 1 class product nằm trong file Product.cs như sau:



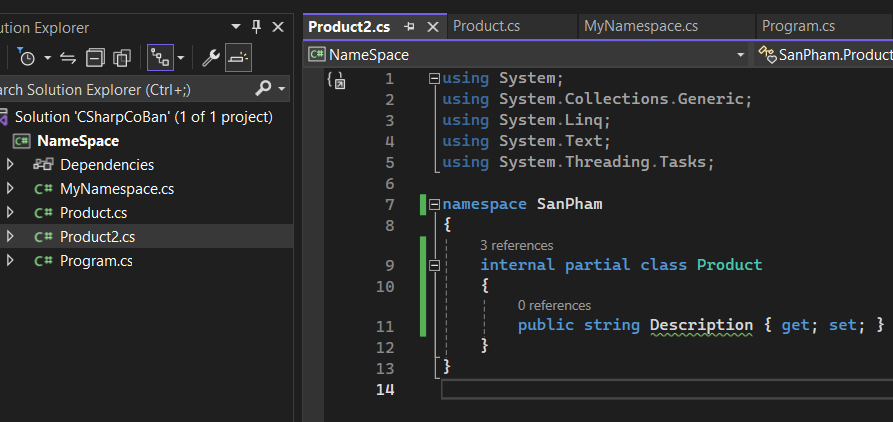


Giờ ta muốn thêm các phương thức hay trường dữ liệu cho class Product thì chúng ta có thể tiếp tục khai báo trong class Procduct nhưng như vậy nếu có quá nhiều class Product sẽ rất dài, chúng ta có thể chia nhỏ file như sau:

Thêm từ khóa partial vào class 

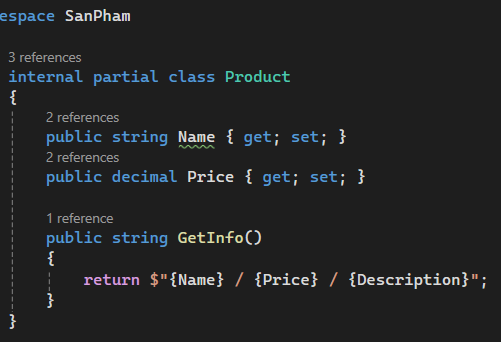
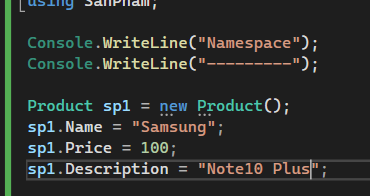
Khi đó ta có thể chia nhỏ class này ra nhiều file code khác

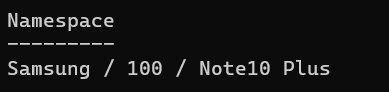
Ta tạo 1 file Product2.cs

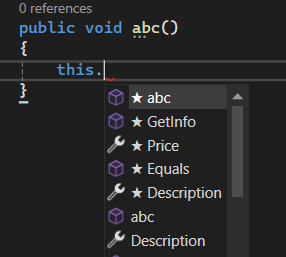


Trong file này ta khai báo giống hệt file product 1 về namespace và class và chúng ta thêm các định nghĩa mới

Khi đó product.cs sẽ có thể sử dụng những định nghĩa mới trong product2.cs

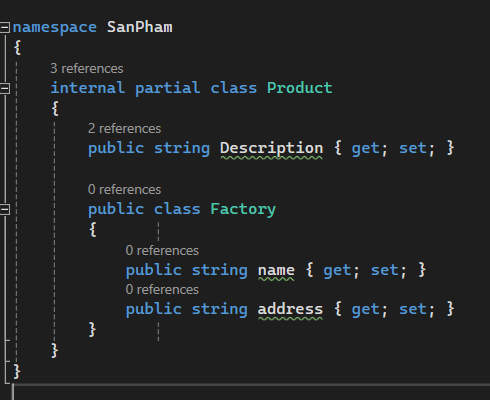
 



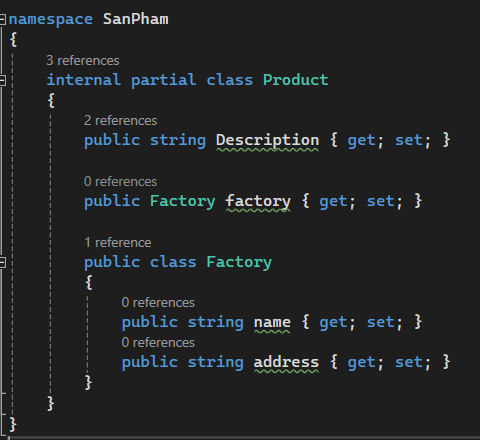
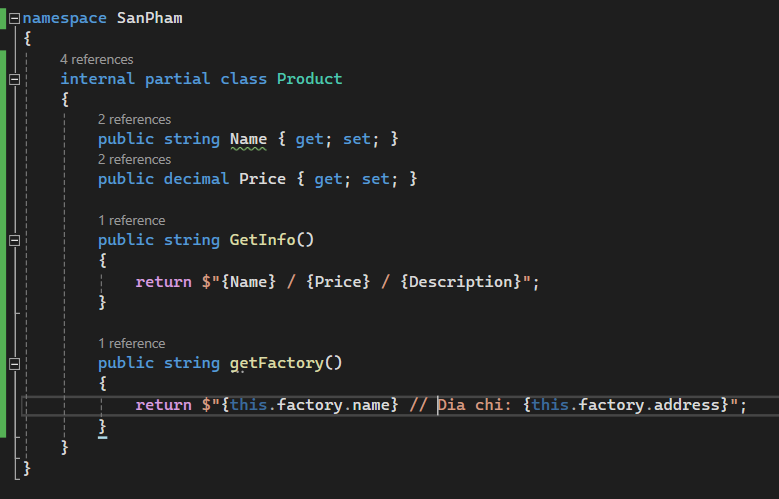
Khi ta thao tác trên các partial ta có thể truy cập được vào các thành phần trong các partial còn lại:

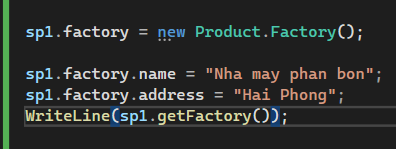
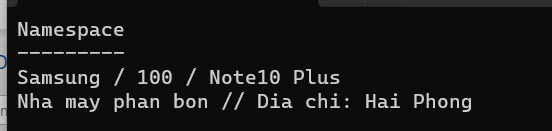
## Khai báo class bên trong class

Để khai báo 1 class là con của 1 class khác ta sẽ khai báo class đó trong phần định nghĩa của class cha



Khi 1 class là con của 1 class khác ta có thể sử dụng tất cả các access modifier giống như 1 thành phần của class

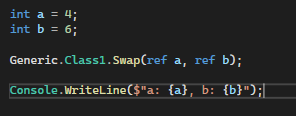
 

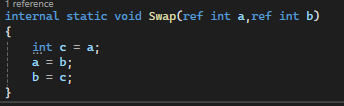
# 17-Generic

* Generic hay còn gọi là kiểu đại diện
* Trong giải thuật việc sử dụng generic trong các lớp giúp sử dụng code 1 cách tối đa mà không phải viết lại code nhiều lần trong trường hợp code có logic giống nhau

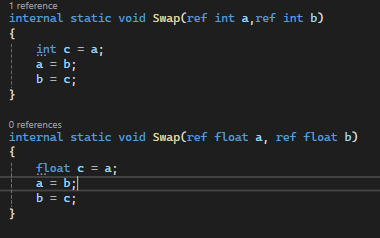
## Vấn đề

Ta có bài toán sau:

* Xây dưng hàm Swap dùng để đổi chỗ giả trị 2 biến bất kỳ





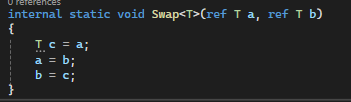
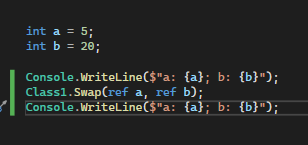
* Ta nhận thấy rằng với mỗi kiểu dữ liệu của a và b ta lại phải overload lại hàm Swap
* 
* Ta nhận thấy rằng về giải thuật giống nhau trên những kiểu dữ liệu khác nhau, lúc này ta có thể sử dụng generic để giải quyết

## Generic trong phương thức

Cú pháp

<access modifier> <static nếu có> <void hoặc return type> <method name><datatype 1, datatype 2>(<data type 1> <param1>,…)

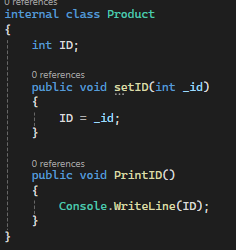
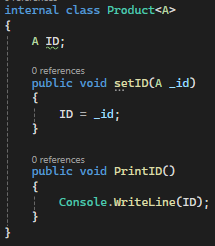
Tên kiểu dữ liệu ta tự định nghĩa, ví dụ dưới là kiểu dữ liệu T và kiểu này cụ thể như nào lúc chạy sẽ biết



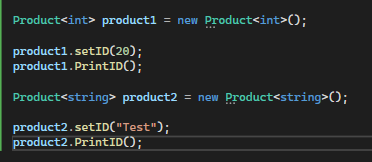
Khi sử dụng ta sẽ khai báo xem T là kiểu dữ liệu gì

## Generic trong class

Ta cos 1 class product như sau:

Bài toàn giờ ở đây là ID có thể là string hoặc các kiểu dữ liệu khác vì vậy chúng ta sẽ làm như sau

Kết quả:



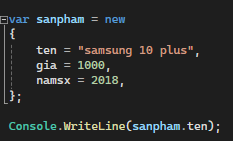
# 18-Kiểu vô danh và Dynamic

## Anonymous

Trong C# hỗ trợ tạo ra những object chứa các property, field mà chỉ có thể đọc

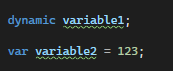
new {property1 = value1, property2 = value2,…}

Thông thường những obj anonymous được khai báo vào các biến ngầm định (var)

Chúng ta chỉ có thể đọc dữ liệu từ kiểu vô danh mà không thể ghi

## Dyanamic

Dynamic gần giống với khai báo ngầm định về hình thức nhưng không nhất thiết phải khởi tạo ngay giá trị cho dữ biến



Biến dynamic có thể được gán bởi bất kỳ đối tượng nào và kiểu của dynamic được xác định ở thời điểm thực thi

Var cần được xác định kiểu dữ liệu ở thời điểm biên dịch

Dynamic xác định kiểu dữ liệu ở thời điểm thực thi

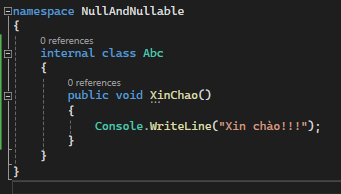
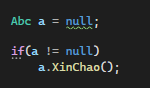
# 19-Null và nullable

## Null

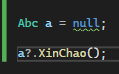
Là từ khóa cho biết không tham chiếu đến một đối tượng nào cả

Được sử dụng cho biến kiểu tham chiếu

Có class sau:



Hoặc có thể viết:

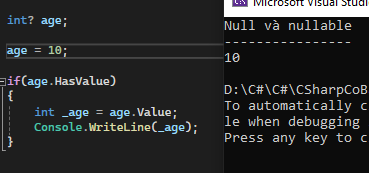


## Nullable

Bình thường biến kiểu tham trị không nhận giá trị null

Nhưng nếu vẫn muốn biến kiểu tham trị có giá trị null ta làm như sau:

Để kiểm tra một biến nullable có giá trị hay không ta sử dụng thuộc tính **HasValue** (trả về true nếu có lưu giá trị, trả về false khi không lưu giá trị gì)

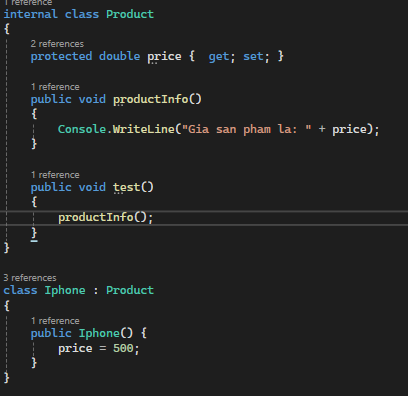
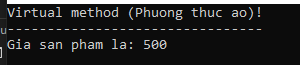


# 20-Vitual method (Phương thức ảo), Abstract (Lớp trừu tượng), Interface

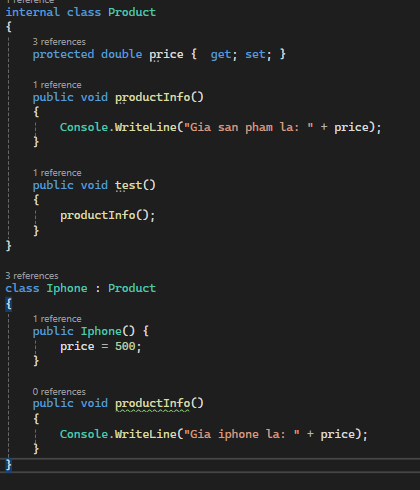
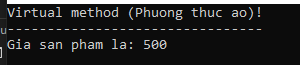
## Tính đa hình trong C#

## Phương thức ảo

* Là phương thức được định nghĩa trong lớp cơ sở (lớp cha), phương thức này có thể overide ở lớp kế thừa

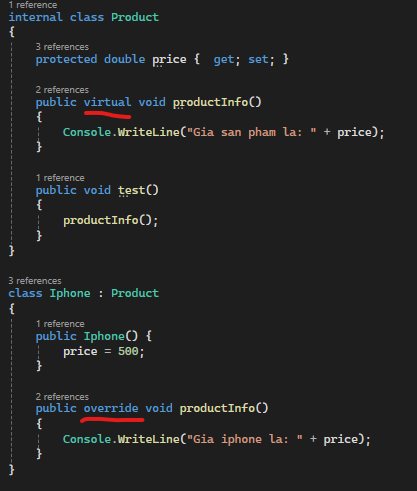


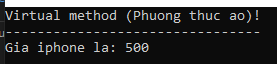
Trường hợp ta muốn nạp chồng phương thức:



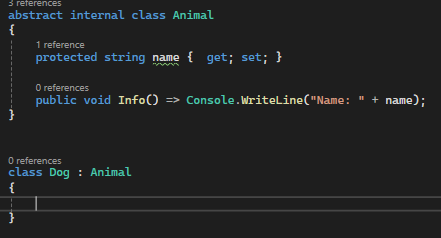
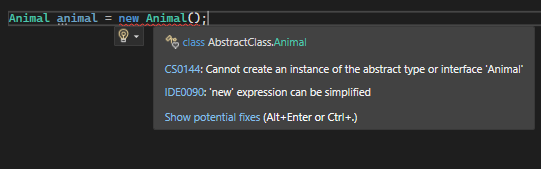
Kết quả phương thức productInfo ở lớp kế thừa không được thự thi

Để thực thi được ta làm như sau:



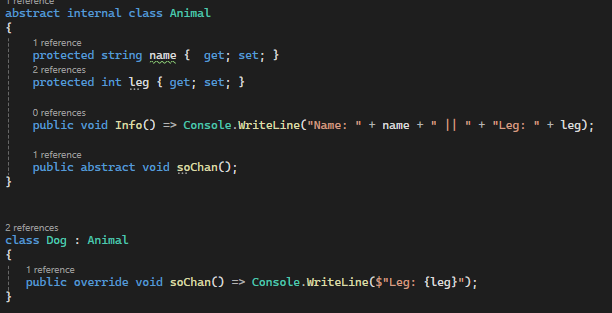


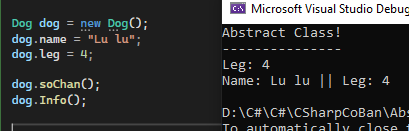
## Lớp trừu tượng (abstract)

Lớp abstract là các lớp được dùng làm cơ sở cho các lớp kế thừa mà không thể được khởi tạo



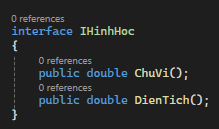
Trong lớp trừu tượng ta cũng có thể khai báo ra một phương thức trừu tượng bằng cách thêm từ khóa abstract vào sau access modifier

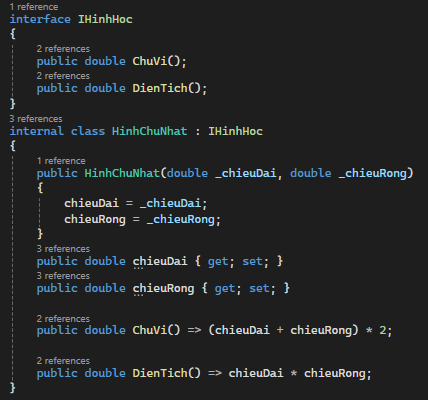
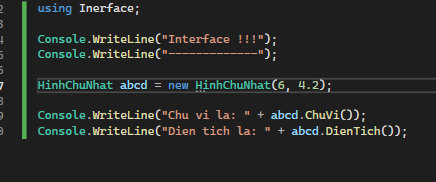
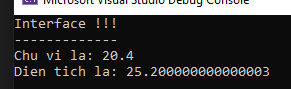
Phương thức trừu tượng chỉ có phần đầu không có phần thân, và bắt buộc lớp kế thừa phải nạp trồng lại nó



## Interface

* Interface gần giống như lớp abstract tứ là khai báo ra cấu trúc là 1 class nhưng không được dùng để khởi tạo đối tượng mà nó chỉ được dùng để làm cơ sở cho các lớp kế thừa
* Khác biệt với abstract ở chỗ là trong interface cỉ chứa các phương thức, các phương thức khai báo trong interface mặc định chỉ cần khai báo tên và nó là phương thức trừu tượng, lớp kế thừa bắt buộc phải định nghĩa lại phương thức đó

Lưu ý đặt tên quy ước có chữ cái I viêt hoa đứng đầu để xác định đó là interface



Nếu 1 lớp được implement từ nhiều interface thì chỉ cần viết các interface cách nhau bởi dấu phẩy

