

Giới thiệu về khóa học C# căn bản

C# là một ngôn ngữ mềm dẻo và rất phổ biến hiện nay. Nhiều lập trình viên đã lựa chọn ngôn ngữ này cho các ứng dụng của mình bởi sự thân thiện và những tính năng mạnh mà nó hỗ trợ. Bạn có thể yên tâm khi sử dụng C# để viết các phần mềm desktop hay các ứng dụng web. Khóa học C# căn bản của chúng tôi sẽ giúp bạn tiếp cận với ngôn ngữ này.

Các bạn hãy hình dung học một ngôn ngữ lập trình giống như việc học một ngoại ngữ. Bạn phải làm quen với các quy tắc diễn đạt, các cấu trúc ngữ pháp của ngôn ngữ đó. Lúc đầu sẽ có rất nhiều khó khăn nhưng càng tiếp cận bạn sẽ thấy nó rất thân thiện và tự nhiên. Với C# ,đầu tiên các bạn chắc sẽ gặp nhiều bỡ ngỡ, nhưng chỉ sau khóa học này, bạn có thể tự tin sử dụng C# để viết ra những ứng dụng nhỏ, sử dụng ngôn ngữ này để giải quyết nhiều bài toán thực tế. Đây cũng là nền tảng để bạn tiếp tục học và sử dụng những công nghệ cao hơn.

Khóa học C# căn bản nằm trong chương trình học về công nghệ .Net. Sau khóa học này bạn sẽ sử dụng thành thạo ngôn ngữ C#, hiểu rõ hơn các khái niệm trong lập trình hướng đối tượng , về cơ bản nắm bắt được các kiến thức nền tảng của của công nghệ .Net. Từ đó, bạn có thể tự tin tiếp cận với những kiến thức cao hơn trong công nghệ .Net. Chúng tôi sẽ lần lượt gửi tới bạn những kiến thức cơ bản nhất từ quy tắc đặt tên, cách viết câu lệnh, các cấu trúc lệnh trong C# đến các kiến thức về hướng đối tượng trong ngôn ngữ này.

Để chuẩn bị cho khóa học này, bạn cần tìm hiểu một chút về .Net Framework ,bộ Visual Sutdio.Net 2005, cách cài đặt và sử dụng nó.

DANH MỤC BÀI HỌC:

Phần 1: Tìm hiểu ngôn ngữ C#

Bài 1: Giới thiệu về ngôn ngữ C#- Hello C#

Bài 2: Các kiểu dữ liệu trong C#. Biến, hằng và cách sử dụng

Bài 3: Kiểu Enumerator

Bài 4: Kiểu mảng và kiểu chuỗi kí tự

Bài 5: Các cấu trúc lệnh trong C#

Bài 6: Toán tử

Bài 7: Xử lý ngoại lệ: Các lệnh throw ,try- catch, finally.

Phần 2: Hướng đối tượng trong C#

Bài 8: Lớp và đối tượng

Bài 9: Methods và các vấn đề liên quan

Bài 10: Overloading methods

Bài 11: Constructor và Destructor

Bài 12: Iheritance

Bài 13: Overriding Method

Bài 14: Polymorphism

Bài 15: InterFace

Bài 16: Struct

Bài 17: NameSpace

Phần 3: Ôn tập

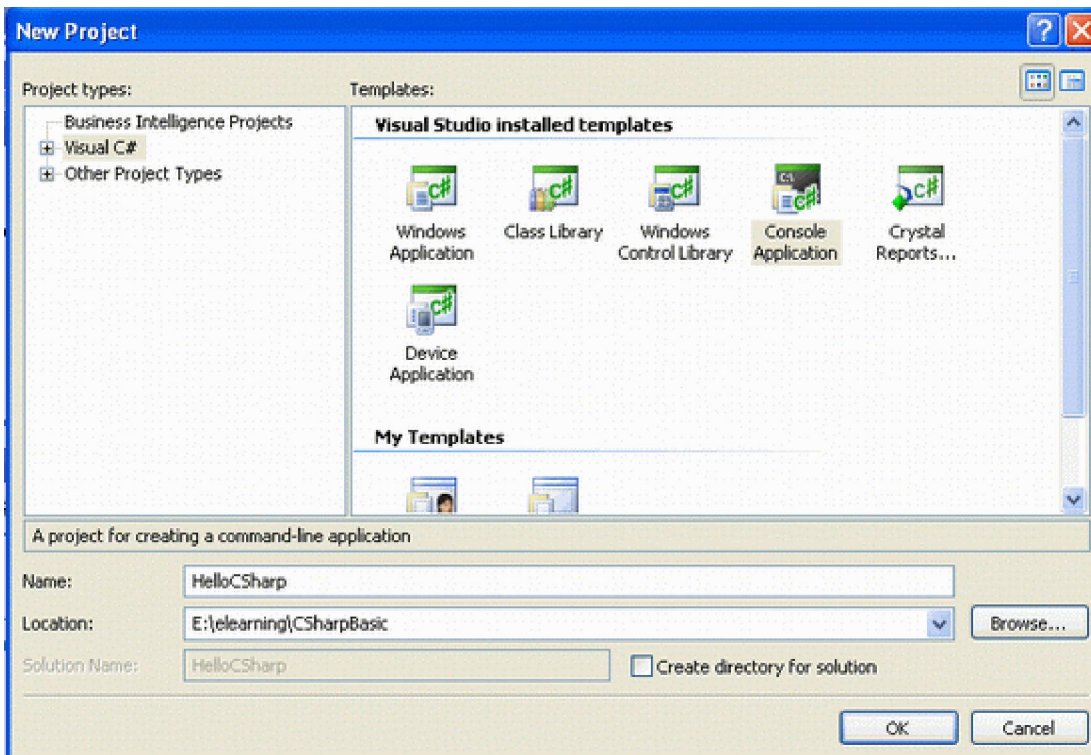
Bài 18: Ôn tập phần ngôn ngữ C#

Bài 19: Ôn tập phần hướng đối tượng trong C#

Bài 1: Giới thiệu về ngôn ngữ C#- Hello C#

Trước tiên chúng ta hãy cùng tìm hiểu một ứng dụng đơn giản nhất của C# thông qua một ứng dụng đơn giản Hello C#.

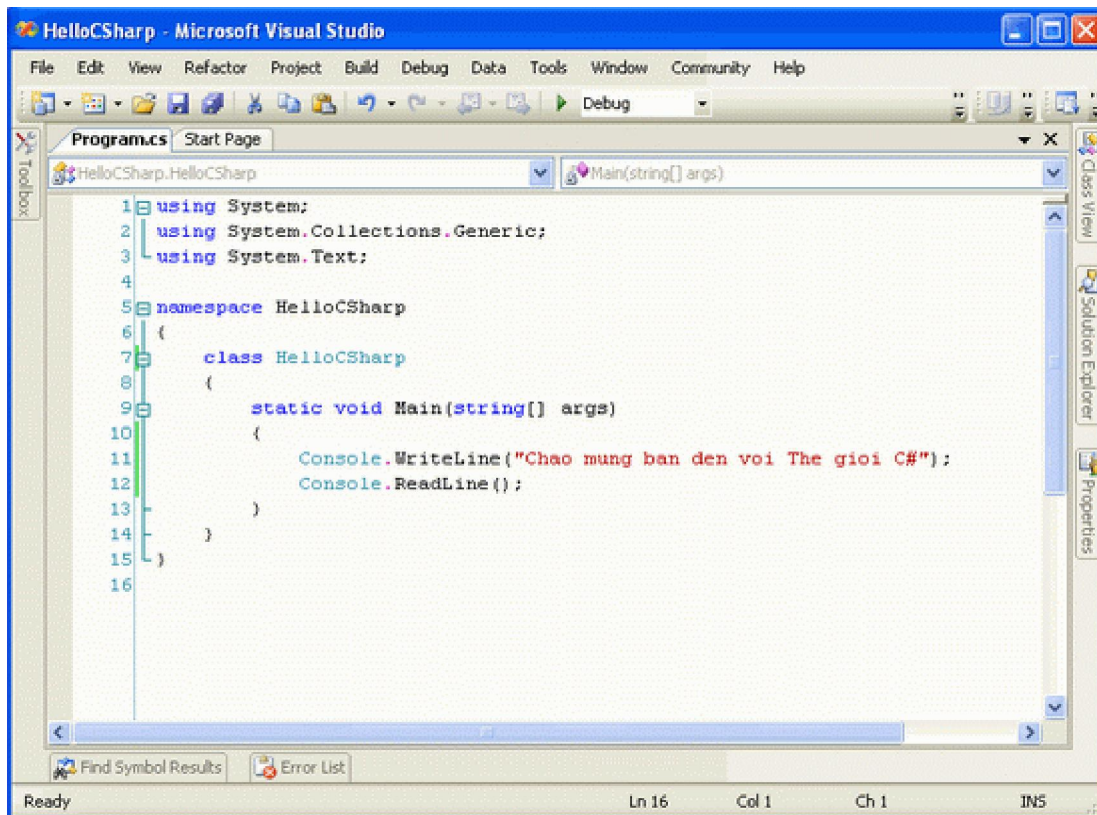
Đầu tiên các bạn mở *Visual Studio.Net 2005* chọn **File-> New-> Project** và chọn ứng dụng **Console Application**



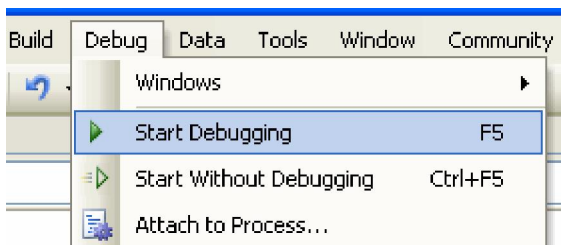
Các bạn gõ tên của ứng dụng vào ô text **Name**.

Chọn nơi lưu trữ ứng dụng bằng cách **Browse** đến thư mục bạn muốn lưu.

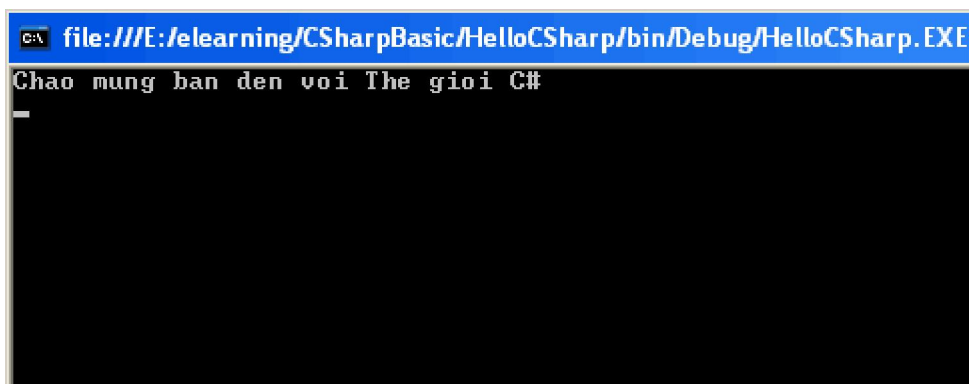
Sau khi nhấn chọn **OK** và cửa sổ soạn thảo ứng dụng xuất hiện, bạn soạn thảo chương trình như sau:



Nhấn **F5** hoặc sử dụng thực đơn (menu) **Debug > Start Debugging** để thực thi chương trình:



Kết quả được hiển thị như sau:



Bạn đã thực hiện hoàn tất một ứng dụng được xây dựng trên nền tảng công nghệ .NET với ngôn ngữ C#. Chúng ta cùng tìm hiểu về nội dung của chương trình.

Khai báo các lớp thư viện cơ sở

```
using  
System;  
using System.Collections.Generic;  
using System.Text;
```

Đây là phần khai báo các lớp thư viện cơ sở của .Net cho chương trình. Các lớp thư viện cơ sở có chứa các hàm mà các bạn có thể sử dụng được ngay. Để khai báo các thư viện này các bạn phải sử dụng từ khóa **using**.

Namespace và Class

Mọi đối tượng của một chương trình C# đều phải đặt trong một class (hay lớp) và các class này sẽ được đặt trong một Namespace (dịch ra tiếng việt là không gian tên). Các Namespace phải có tên khác nhau, các class trong các namespace khác nhau thì có thể trùng tên. Khi đó tên các class sẽ được phân biệt bởi namespace chứa nó.

Một ví dụ mô phỏng về Namespace và Lớp đó là: Hà Nội và Tp.HCM đều có đường mang tên Trần Hưng Đạo, nhưng rõ ràng là con đường Trần Hưng Đạo ở hai nơi hoàn toàn khác nhau mặc dù vẫn có chung tên Trần Hưng Đạo. Vậy ta nói Hà Nội và Tp.HCM là hai namespace chứa hai lớp có cùng tên "Trần Hưng Đạo".

Các bạn sẽ được tìm hiểu kỹ về 2 khái niệm này ở các bài học sau của chúng tôi trong phần Lập trình hướng đối tượng (OOP).

Toán tử "."

Như vậy để gọi một bạn Hoa bạn phải gắn vào lớp học của bạn ấy Hoa lớp tin 1 chẳng hạn. C# cung cấp cho bạn một toán tử dùng để gọi ra các lớp của một namespace đó là toán tử ".". Nếu muốn gọi ra một lớp trong một namespace bạn sử dụng cú pháp sau:

Namespace.Class

Ví dụ HANOI.TranHungDao, viết như thế này để chỉ đến con đường Trần Hưng Đạo tại Hà Nội, hoàn toàn khác với con đường TranHungDao ở TPHCM được viết với dạng TPHCM.TranHungDao.

Sử dụng lớp System.Console để nhập/ xuất dữ liệu.

Lớp System.Console đây là lớp thường được sử dụng trong các chương trình Console để đọc và ghi ra màn hình các giá trị text.

Ở chương trình, chúng ta sử dụng:

```
Console.WriteLine("Chào mừng bạn đến với The giới C#"); //ghi dữ liệu ra màn hình
Console.ReadLine(); //đọc dữ liệu
```

Một số hàm thường dùng

```
Console.Read():   Đọc dữ liệu từ bàn phím
Console.ReadLine(): Đọc dữ liệu từ bàn phím và đưa con trỏ xuống dòng dưới.
Console.Write():  Ghi dữ liệu ra màn hình
Console.WriteLine(): Ghi dữ liệu ra màn hình và xuống dòng.
```

Câu lệnh và khối lệnh

Từ chương trình đầu tiên trên, các bạn đã phần nào hình dung được về ngôn ngữ C#.

Các bạn hãy hình dung câu lệnh trong lập trình giống với một câu văn trong văn bản. Nó cũng cần tuân theo những quy tắc nhất định. Nếu ngôn ngữ trong đời sống bạn phải diễn đạt cho mọi người xung quanh hiểu thì trong ngôn ngữ lập trình một câu lệnh được viết với mục đích làm cho trình biên dịch hiểu. Chúng ta tuân thủ một số quy tắc sau khi lập trình với ngôn ngữ C#:

Câu lệnh trong C# luôn được kết thúc bằng dấu ";"

Bạn có thể có một đoạn văn gồm nhiều câu văn diễn tả một nội dung nào đó. Bạn cũng có thể gom nhiều câu lệnh thành một khối lệnh để làm một công việc.

Bạn có thể gom lại thành một khối như sau:

```
{
    Câu lệnh 1;
    Câu lệnh 2;
    Câu lệnh 3;
}
```

Khối lệnh sẽ được đặt trong cặp dấu móc đơn "{}"

Luyện tập:

Tạo một ứng dụng có tên là Hello World lưu trong thư mục là tên của bạn trong ổ cứng. Trong đó in ra 2 dòng:

```
Hello World!
```

Welcome to C# World!

Bài 2: Các kiểu dữ liệu trong C#. Biến, hằng và cách sử dụng

1. Các kiểu dữ liệu trong C#

C# chia thành hai tập hợp kiểu dữ liệu chính: Kiểu xây dựng sẵn (built-in) mà ngôn ngữ cung cấp cho người lập trình và kiểu được người dùng định nghĩa (user-defined) do người lập trình tạo ra.

C# phân tập hợp kiểu dữ liệu này thành hai loại: Kiểu dữ liệu giá trị (value) và kiểu dữ liệu tham chiếu (reference). Bạn có thể chuyển đổi từ kiểu dữ liệu này sang kiểu dữ liệu khác qua việc boxing và unboxing (Tôi sẽ giới thiệu với bạn ở phần sau của bài học này)

Bảng các kiểu dữ liệu xây dựng sẵn

C# Data Type	Mô tả
<code>object</code>	kiểu dữ liệu cơ bản của tất cả các kiểu khác
<code>string</code>	Được sử dụng để lưu trữ những giá trị kiểu chữ cho biến
<code>int</code>	Sử dụng để lưu trữ giá trị kiểu số nguyên
<code>byte</code>	sử dụng để lưu trữ giá byte
<code>float</code>	Sử dụng để lưu trữ giá trị số thực
<code>bool</code>	Cho phép một biến lưu trữ giá trị đúng hoặc sai
<code>char</code>	Cho phép một biến lưu trữ một ký tự

Ghi chú: Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu `struct` đều là kiểu dữ liệu tham chiếu. trong bài học này chúng ta sẽ tìm hiểu các kiểu xây dựng sẵn.

2. Biến và Hằng

a. Biến

Một biến là một vùng lưu trữ với một kiểu dữ liệu. Để tạo một biến chúng ta phải khai báo kiểu của biến và gán cho biến một tên duy nhất. Biến có thể được khởi tạo giá trị ngay khi được khai báo, hay nó cũng có thể được gán một giá trị mới vào bất cứ lúc nào trong chương trình.

Các biến trong C# được khai báo theo công thức như sau:

AccessModifier *DataType* *VariableName*;

Trong đó:

AccessModifier: xác định ưu tiên truy xuất tới biến

Datatype: định nghĩa kiểu lưu trữ dữ liệu của biến

VariableName: là tên biến

Cấp độ truy xuất tới biến được mô tả như bảng dưới đây

Access Modifier	Mô tả
<code>public</code>	Truy cập tại bất kỳ nơi đâu
<code>protected</code>	Cho phép truy xuất bên trong một lớp nơi biến này được định nghĩa, hoặc từ các lớp con của lớp đó.
<code>private</code>	Chỉ truy xuất ở bên trong lớp nơi mà biến được định nghĩa.

Ví dụ bạn khai báo một biến kiểu int

```
int bien1;
```

Bạn có thể khởi gán ngay cho biến đó trong lúc khai báo

```
int bien1 = 9;
```

hoặc có thể gán giá trị sau khi khai báo như sau:

```
int bien1;
bien1 = 9;
```

Cách khai báo biến tương ứng với các kiểu dữ liệu:

C# Data Type	Ví dụ
<code>object</code>	<code>object obj = null;</code>
<code>string</code>	<code>string str = "Welcome";</code>
<code>int</code>	<code>int ival = 12;</code>
<code>byte</code>	<code>byte val = 12;</code>

float	float val = 1.23F;
bool	bool val1 = false; bool val2 = true;
char	char cval = 'a';

Ví dụ sau sẽ minh họa cách sử dụng biến:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SuDungBien
{
    class MinhHoa
    {
        static void Main()
        {
            int bien1 = 9; // khai báo và khởi tạo giá trị cho một biến
            System.Console.WriteLine("Sau khi khai tạo: bien1 ={0}", bien1);
            bien1 = 15; // gán giá trị cho biến
            System.Console.WriteLine("Sau khi gán: bien1 ={0}", bien1);
            Console.ReadLine();
        }
    }
}
```

Các bạn hãy chú ý đến màu sắc của đoạn code trên. Các chữ có màu xanh dương là từ khóa, phần văn bản màu xanh lục sau dấu sổ chéo "//" là các chú thích, phần text nằm trong dấu "" có màu đỏ là các kí tự. Lệnh Write và WriteLine có phân biệt việc in ra màn hình biến và kí tự. Sau đây tôi sẽ lần lượt giải thích các khái niệm trên.

Từ khóa

Trong cuộc sống, mọi ngôn ngữ đều chứa những từ khóa và những từ này hiểu được bởi người nói ra nó. Điều đó cũng đúng với C#. Từ khóa trong C# là những từ đặc biệt và mang nghĩa đặc biệt chỉ dành riêng cho ngôn ngữ này. Trong VS.net những từ khóa của C# sẽ có màu xanh ra trời. trong ví dụ trên các từ khóa là `using`, `namespace`, `int`
Tên và quy tắc đặt tên trong C#

Mọi sự vật hiện tượng trong cuộc sống đều có tên gọi để phân biệt với nhau và điều đó cũng đúng đối với một chương trình máy tính. Mọi đối tượng của chương trình C# đều có tên. Bạn có thể đặt tên cho biến, cho hàm, cho lớp và cho các namespace. Chú ý rằng C# là ngôn ngữ phân biệt chữ hoa chữ thường. Ví dụ bạn khai báo 2 biến kiểu int

`int bien1;`
và `int Bien1;`

Thì 2 biến này là 2 đối tượng khác nhau.

Khi bạn đặt tên cần chú ý đến các nguyên tắc sau:

- Ký tự đầu tiên phải là một chữ cái (có thể là chữ hoa hoặc thường) hoặc là dấu gạch dưới (/)
- Ký tự tiếp theo có thể lấy bất kì.
- Tên không được trùng với từ khóa.
-

Cách viết chú thích

Chú thích trong chương trình C# là những phần text làm rõ hơn cho phần code của lập trình viên. Chú thích không được đọc bởi trình biên dịch, nó không liên quan gì đến chương trình của bạn

Có 2 cách viết chú thích trong C#:

Nếu chú thích trên một dòng bạn đặt phần chú thích sau 2 dấu sổ chéo

```
// chú thích
```

Nếu chú thích trên nhiều dòng bạn đặt phần chú thích trong cặp `/* */` cụ thể

```
/* chú thích*/
```

Cách in ra màn hình

Khi in các ký tự ra màn hình bạn phải đặt chúng trong cặp dấu `""`.

Vậy in ra biến thì sao?

Bạn sẽ làm theo mẫu sau

Ví dụ bạn có 3 biến `bien1`, `bien2`, `bien3` và bạn muốn in chúng ra màn hình. Bạn sẽ dùng câu lệnh:

```
Console.WriteLine("{0} {1} {2}",bien1, bien2, bien3);
```

b. Hằng

Hằng cũng là một biến nhưng giá trị của hằng không thay đổi. Biến là công cụ rất mạnh, tuy nhiên khi làm việc với một giá trị được định nghĩa là không thay đổi, ta phải đảm bảo giá trị của nó không được thay đổi trong suốt chương trình. Ví dụ, khi lập một chương trình thí nghiệm hóa học liên quan đến nhiệt độ sôi, hay nhiệt độ đông của nước, chương trình cần khai báo hai biến là `DoSoi` và `DoDong`, nhưng không cho phép giá trị của hai biến này bị thay đổi hay bị gán. Để ngăn ngừa việc gán giá trị khác, ta phải sử dụng biến kiểu hằng. Hằng được phân thành ba loại: giá trị hằng

(literal), biểu tượng hằng (symbolic constants), kiểu liệt kê (enumerations). Chúng ta sẽ tìm hiểu về kiểu liệt kê ở bài học sau.

Giá trị hằng

Ta có một câu lệnh gán như sau: `x = 100;`

Giá trị 100 là giá trị hằng. Giá trị của 100 luôn là 100. Ta không thể gán giá trị khác cho 100 được.

Biểu tượng hằng

gán một tên cho một giá trị hằng, để tạo một biểu tượng hằng dùng từ khóa `const` và cú pháp sau:

`<const> <type> <tên hằng> = <giá trị>;`

Một biểu tượng hằng phải được khởi tạo khi khai báo, và chỉ khởi tạo duy nhất một lần trong suốt chương trình và không được thay đổi. Ví dụ:

```
const int DoSoi = 100;
```

Trong khai báo trên, 32 là một hằng số và DoSoi là một biểu tượng hằng có kiểu nguyên.

Ví dụ sau sẽ minh họa cách sử dụng biểu tượng hằng

```
class MinhHoaC3
{
    static void Main()
    {
        const int DoSoi = 100; // Độ C
        const int DoDong = 0; // Độ C
        System.Console.WriteLine( "Do dong cua nuoc {0}", DoDong );
    }
    System.Console.WriteLine( "Do soi cua nuoc {0}", DoSoi );
}
```

Kết quả:

Do dong cua nuoc 0

Do soi cua nuoc 100

Các bạn đã hiểu được sự khác nhau giữa biến và hằng- cách sử dụng chúng trong C# . Ngoài ra bạn còn biết thế nào là từ khóa, quy tắc đặt tên trong C#, cách viết chú thích và cách ghi ra màn hình kí tự, biến...

Luyện tập

Câu hỏi:

- 1) Những từ theo sau từ nào là từ khóa trong C#: field, cast, as, object, throw, football, do, get, set, basketball.
- 2) Có bao nhiêu cách khai báo comment trong ngôn ngữ C#, cho biết chi tiết?
- 3) C# chia làm mấy kiểu dữ liệu chính? Nếu ta tạo một lớp tên myClass thì lớp này được xếp vào kiểu dữ liệu nào?

Bài tập:

Bài 1: Tìm lỗi của chương trình sau. Sửa lỗi và biên dịch lại chương trình.

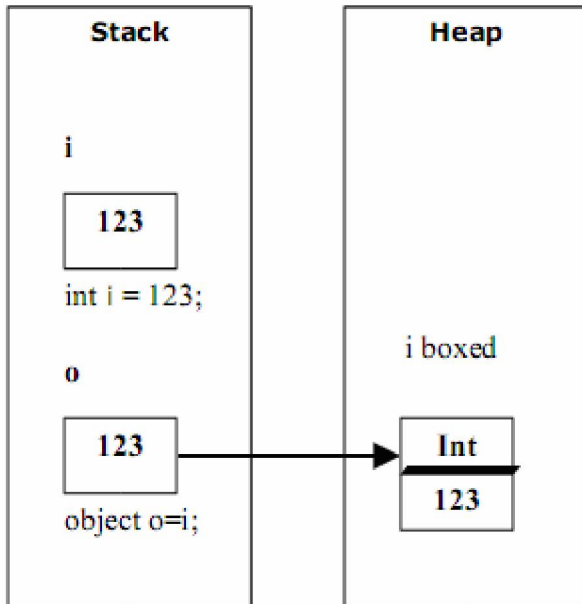
class Bai1

```
{  
    public static void Main()  
    {  
        double myDouble;  
        decimal myDecimal;  
        myDouble = 3.14;  
        myDecimal = 3.14;  
        Console.WriteLine("My Double: {0}", myDouble);  
        Console.WriteLine("My Decimal: {0}", myDecimal);  
    }  
}
```

Bài 2(Tiếp): Boxing và Unboxing

1. Boxing

Bạn có thể dễ dàng hình dung quá trình này thông qua tên gọi của nó, nghĩa là một giá trị được đưa vào bên trong một đối tượng. Nói cách khác, boxing là những xử lý cho phép kiểu dữ liệu giá trị như (int, uint, long...) được đối xử như kiểu tham chiếu (các đối tượng). Và quá trình boxing được thực hiện ngầm định. Bạn hãy xem hình dưới minh họa về quá trình boxing một số nguyên:



Đây là chương trình minh họa quá trình trên.

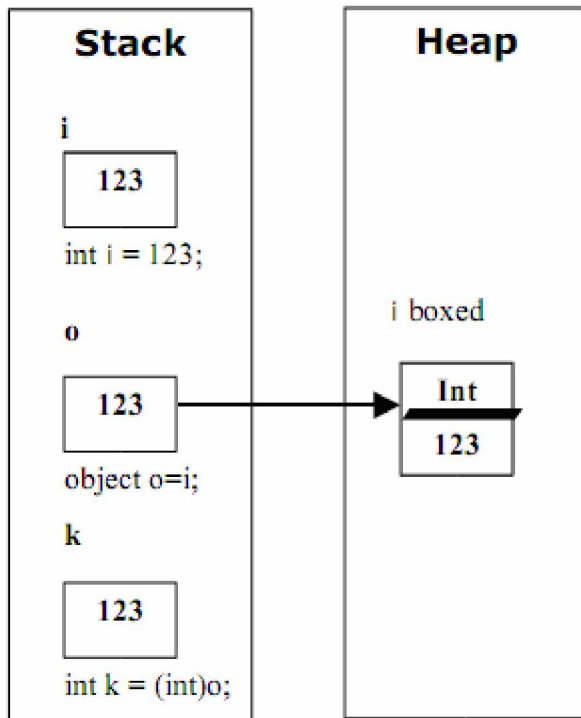
```
using System;
class Boxing
{
    public static void Main()
    {
        int i = 123;
        object o = i;

        Console.WriteLine("The object value = {0}", o);
        Console.ReadLine();
    }
}
```

2.Unboxing

Unboxing là quá trình ngược lại với boxing, tức là đưa từ một đối tượng ra một giá trị . Quá trình này sẽ được thực hiện một cách tường minh. Và để thực hiện được điều này bạn cần chắc chắn rằng đối tượng đã được boxing đúng kiểu giá trị đưa ra và sao chép giá trị từ thể hiện hay đối tượng vào

biến kiểu giá trị. Hình dưới đây mô tả quá trình unboxing. Như bạn thấy nó ngược lại với quá trình boxing ở trên



Unboxing sau khi thực hiện Boxing.

Đây là chương trình minh họa cả quá trình boxing và unboxing:

```
using System;
public class Unboxing
{
    public static void Main()
    {
        int i = 123;
        // Boxing
        object o = i;
        // Unboxing phải được thực hiện tường minh tường minh
        int k = (int) o;
```

```
        Console.WriteLine("k: {0}", k);  
    }  
}
```

Nếu một đối tượng được Unboxing là null hay là tham chiếu đến một đối tượng có kiểu dữ liệu khác, một `InvalidOperationException` (Ngoại lệ) sẽ được phát sinh. Các bạn sẽ được học về cách xử lý ngoại lệ ở bài 7 của khóa học.

Bài 3: Kiểu liệt kê (Enumerator)

1. Định nghĩa

Kiểu liệt kê đơn giản là tập hợp các tên hằng có giá trị không thay đổi (thường được gọi là danh sách liệt kê).

2. Cách khai báo và sử dụng

Các bạn hãy xem lại ví dụ ở bài học số 2 về cách sử dụng biểu tượng hằng, chúng ta có hai biểu tượng hằng có quan hệ với nhau:

```
const int DoDong = 0;  
const int DoSoi = 100;
```

Do mục đích mở rộng ta mong muốn thêm một số hằng số khác vào danh sách trên, như các hằng sau:

```
const int DoNong = 60;  
const int DoAm = 40;  
const int DoNguoi = 20;
```

Các biểu tượng hằng trên đều có ý nghĩa quan hệ với nhau, cùng nói về nhiệt độ của nước, khi khai báo từng hằng trên có vẻ cồng kềnh và không được liên kết chặt chẽ cho lắm. Thay vào đó C# cung cấp kiểu liệt kê để giải quyết vấn đề trên:

```
enum NhetDoNuoc  
{  
    DoDong = 0,  
    DoNguoi = 20,  
    DoAm = 40,  
    DoNong = 60,  
}
```

```
DoSoi = 100,  
}
```

Mỗi kiểu liệt kê có một kiểu dữ liệu cơ sở, kiểu dữ liệu có thể là bất cứ kiểu dữ liệu nguyên nào như int, short, long... tuy nhiên kiểu dữ liệu của liệt kê không chấp nhận kiểu ký tự. Để khai báo một kiểu liệt kê ta thực hiện theo cú pháp sau:

[thuộc tính] [bổ sung] enum <tên liệt kê> [:kiểu cơ sở]

```
{  
    danh sách các thành phần  
    liệt kê
```

```
}
```

Thành phần thuộc tính và bổ sung là tự chọn có thể có hoặc không.

Một kiểu liệt kê bắt đầu với từ khóa enum, tiếp sau là một định danh cho kiểu liệt kê:

enum **NhietDoNuoc**

Thành phần kiểu cơ sở chính là kiểu khai báo cho các mục trong kiểu liệt kê. Nếu bỏ qua thành phần này thì trình biên dịch sẽ gán giá trị mặc định là kiểu nguyên int, tuy nhiên chúng ta có thể sử dụng bất cứ kiểu nguyên nào như ushort hay long,...ngoại trừ kiểu ký tự. Đoạn ví dụ sau khai báo một kiểu liệt kê sử dụng kiểu cơ sở là số nguyên không dấu uint:

enum **KichThuoc :uint**

```
{  
    Nho = 1,  
    Vua = 2,  
    Lon = 3,  
}
```

Lưu ý là khai báo một kiểu liệt kê phải kết thúc bằng một danh sách liệt kê, danh sách liệt kê này phải có các hằng được gán, và mỗi thành phần phải phân cách nhau dấu phẩy.
Ví dụ sau minh họa về cách sử dụng kiểu liệt kê

```
using System;  
using System.Collections.Generic;  
using System.Text;
```

```
namespace Bien  
{  
    class KieuEnum  
    {  
        enum NhietDoNuoc: int  
        {
```



```
DoDong = 0,
DoNguoi = 20,
DoAm = 40,
DoNong = 60,
DoSoi = 100,
}
static void Main()
{
    System.Console.WriteLine( "Nhiệt độ dòng: {0}",(int)NhiệtDoNước.DoDong);
    System.Console.WriteLine("Nhiệt độ người: {0}", (int)NhiệtDoNước.DoNguoi);
    System.Console.WriteLine("Nhiệt độ am: {0}", (int)NhiệtDoNước.DoAm);
    System.Console.WriteLine("Nhiệt độ nóng: {0}", (int)NhiệtDoNước.DoNong);
    System.Console.WriteLine("Nhiệt độ soi: {0}",
        (int)NhiệtDoNước.DoSoi);
    Console.ReadLine();
}
}
```

Kết quả:

Nhiệt độ dòng: 0
Nhiệt độ người: 20
Nhiệt độ am: 40
Nhiệt độ nóng: 60
Nhiệt độ soi: 100

Chú ý:

Mỗi thành phần trong kiểu liệt kê tương ứng với một giá trị số, trong trường hợp này là một số nguyên. Nếu chúng ta không khởi tạo cho các thành phần này thì chúng sẽ nhận các giá trị tiếp theo với thành phần đầu tiên là 0. Ta xem thử khai báo sau:

enum Thutu

```
{
    ThuNhat,
    ThuHai,
    ThuBa = 10,
    ThuTu
}
```

Khi đó giá trị của ThuNhat là 0, giá trị của ThuHai là 1, giá trị của ThuBa là 10 và giá trị của ThuTu là 11.

Chú ý:

Kiểu liệt kê là một kiểu hình thức do đó bắt buộc phải thực hiện phép chuyển đổi tường minh với các kiểu giá trị nguyên:

```
int x = (int) ThuTu.ThuNhat;
```

Ở bài sau bạn sẽ được học về kiểu string và kiểu mảng.

Bài 4: Mảng (Array) và kiểu chuỗi kí tự (string)

1. Dữ liệu kiểu mảng

a. Định nghĩa :

Mảng là một nhóm những biến có cùng một kiểu dữ liệu. Những biến này được lưu trữ trong bộ nhớ vùng bộ nhớ kế tiếp do đó mảng cho phép truy xuất và thực thi đến từng phần tử trong mảng.

b. Công thức khai báo một mảng

```
Datatype [] variableName = new Datatype [number of elements];
```

Trong đó:

number of elements: là số phần tử của mảng

Datatype: kiểu dữ liệu mà mảng lưu trữ

variableName: là tên mảng.

Ví dụ:

```
// mảng kiểu int
int[] iarray = new int[5];
// mảng kiểu string
string[] sarray = new string[6];
```

Ví dụ: cách khai báo khác

```
string[] sarray2 = { "Welcome", "to", "C# Array" };
```

Khi lập trình, tùy theo điều kiện chương trình mà bạn có thể chọn lựa một trong hai cách trên.

c. Cách truy xuất đến các phần tử trong mảng.

Để truy xuất đến một phần tử trong một mảng chúng ta sử dụng chỉ số của phần tử trong mảng, ví dụ với mảng `iarray` ở trên, chúng ta sẽ lấy được giá trị của của phần tử thứ 3 trong mảng như sau:

```
// Truy xuất đến phần tử thứ 3 trong mảng
int iValue = iarray[2];
// Gặp lỗi nếu truy xuất đến phần tử không nằm trong mảng
int iValue = iarray[5];
```

Để truy xuất đến phần tử thứ 3, chúng ta dùng chỉ số 2, như thế, chỉ số để đánh dấu các phần tử trong mảng xuất phát từ 0.

Chúng ta cũng dễ dàng nhận thấy khi thực thi, chương trình báo lỗi ở dòng `int iValue = iarray[5]`, do phần tử thứ 6 không tồn tại trong mảng.

2. Dữ liệu kiểu chuỗi

a. Định nghĩa

kiểu dữ liệu chuỗi lưu giữ một mảng những ký tự.

b. Khai báo và sử dụng

Kiểu dữ liệu chuỗi khá thân thiện với người lập trình trong bất cứ ngôn ngữ lập trình nào, kiểu dữ liệu chuỗi lưu giữ một mảng những ký tự (charater). Để khai báo một chuỗi chúng ta sử dụng từ khoá `string` tương tự như cách tạo một thể hiện của bất cứ đối tượng nào:

```
string chuoi;
chuoi = "Learning C#";
```

chúng ta cũng có thể gán giá trị cho chuỗi ngay khi khởi tạo như sau:

```
string chuoi = "Learning C#";
```

Bạn có thể tham khảo thêm về kiểu `string` ở các bài viết này:

<http://www.itgatevn.com.vn/index.aspx?u=iex&su=d&cid=44&id=20788>

<http://www.itgatevn.com.vn/index.aspx?u=iex&su=d&cid=44&id=20757>

<http://www.itgatevn.com.vn/index.aspx?u=iex&su=d&cid=44&id=20784>

Bài 5: Các cấu trúc lệnh trong C#

1. Các cấu trúc điều khiển

C# cung cấp hai cấu trúc điều khiển thực hiện việc lựa chọn điều kiện thực thi chương trình đó là cấu trúc if và switch...case

Cấu trúc if

Cấu trúc if trong C# được mô tả như sau:

```
if (biểu thức điều kiện)
{
    // câu lệnh thực thi nếu biểu thức điều kiện đúng
}
[else
{
    // câu lệnh thực thi nếu biểu thức điều kiện sai
}]
```

Ví dụ:

```
if (20 % 4 > 0)
{
    Console.WriteLine("Số 20 không chia hết cho 4");
}
else
{
    Console.WriteLine("Số 20 chia hết cho số 4");
}
```

Cấu trúc switch ... case

Cấu trúc switch....case có cấu trúc như sau:

```
// switch ... case
switch (Biến điều kiện)
{
case giá trị 1:
    Câu lệnh thực thi
    break;
case giá trị 2:
    Câu lệnh thực thi
    break;
case giá trị 3:
    Câu lệnh thực thi
    break;
default:
    Câu lệnh thực thi
    break;
}
```

Ví dụ:

```
int x = 20 % 4;
switch (x)
{
    case 1:
        Console.WriteLine("20 chia cho 4 được số dư là 1");
        break;
    case 0:
        Console.WriteLine("20 chia hết cho 4");
        break;
    default:
        Console.WriteLine("Không thuộc tất cả các trường hợp
trên");
        break;
}
```

2. Cấu trúc vòng lặp trong lập trình C#

C# cung cấp các cấu trúc vòng lặp chương trình

- While
- Do... while
- For
- Foreach

Sau đây, tôi xin giới thiệu công thức và ví dụ sử dụng các vòng lặp trên

Vòng lặp *While*

Cấu trúc vòng lặp while

while (biểu thức điều kiện)

```
{  
    // câu lệnh  
}
```

Thực thi câu lệnh hoặc một loạt những câu lệnh đến khi điều kiện không được thỏa mãn.

Ví dụ:

```
using System;  
class WhileTest  
{  
    public static void Main()  
    {  
        int n = 1;  
  
        while (n < 6)  
        {  
            Console.WriteLine("Current value of n is {0}", n);  
            n++;  
        }  
    }  
}
```

Vòng lặp do

Cấu trúc vòng lặp while

do

```
{  
    // câu lệnh  
}
```

While (biểu thức điều kiện)

Thực thi câu lệnh **ít nhất một lần** đến khi điều kiện không được thỏa mãn.

Ví dụ:

```
using System;  
public class TestDoWhile  
{  
    public static void Main ()  
    {  
        int x;  
        int y = 0;  
  
        do  
        {  
            x = y++;  
            Console.WriteLine(x);  
        }  
  
        while(y < 5);  
    }  
}
```

Vòng lặp **for**

Cấu trúc vòng lặp for

for ([phần khởi tạo]; [biểu thức điều kiện]; [bước lặp])


```
{  
    // thực thi câu lệnh  
}
```

Ví dụ:

```
using System;  
public class ForLoopTest  
{  
    public static void Main()  
    {  
        for (int i = 1; i <= 5; i++)  
            Console.WriteLine(i);  
    }  
}
```

Vòng lặp *foreach*

Câu lệnh lặp `foreach` khá mới với những người đã học ngôn ngữ C, từ khóa này được sử dụng trong ngôn ngữ Visual Basic. Câu lệnh `foreach` cho phép chúng ta lặp qua tất cả các mục trong một mảng hay trong một tập hợp. Cú pháp sử dụng lệnh lặp `foreach` như sau:

```
foreach (<kiểu dữ liệu thành phần> <tên truy cập> in <mảng/tập hợp> )  
{  
    // thực hiện thông qua <tên truy cập> tương ứng với  
    // từng mục trong mảng hay tập hợp  
}
```

Dữ liệu kiểu tập hợp chưa được đề cập tới trong các bài học trước nên bạn chỉ cần quan tâm đến vòng lặp foreach sử dụng với mảng. Bạn hãy xem ví dụ sau để hiểu cách sử dụng của vòng lặp foreach truy cập đến từng phần tử của mảng.

```
using System;

public class UsingForeach
{
    public static int Main()
    {
        int[] intArray = {1,2,3,4,5,6,7,8,9,10};
        foreach( int item in intArray)
        {
            Console.Write("{0} ", item);
        }
        Console.ReadLine();
        return 0;
    }
}
```

Kết quả:

0 1 2 3 4 5 6 7 8 9 10

3. Các lệnh break, goto và continue

Câu lệnh nhảy goto:

Lệnh nhảy goto là một lệnh nhảy đơn giản, cho phép chương trình nhảy vô điều kiện tới một vị trí trong chương trình thông qua tên nhãn. Goto giúp chương trình của bạn được linh hoạt hơn nhưng trong nhiều trường hợp nó sẽ làm mất đi cấu trúc thuật toán và gây rối chương trình.

Cách sử dụng lệnh goto:

Tạo một nhãn

goto đến nhãn

Nhãn là một định danh theo sau bởi dấu hai chấm (:). Thường thường một lệnh goto gắn với một điều kiện nào đó.

Ví dụ:

```
public class UsingGoto
{
    public static void Main()
    {
        int i = 0;
        lap: // nhãn
            Console.WriteLine("i:{0}",i);
        i++;
        if ( i < 10 )
            goto lap; // nhảy về nhãn lap
        Console.ReadLine();
    }
}
```

Tương đương với vòng lặp for sau:

```
for (int i = 0; i < 10;i++)
```

```
Console.WriteLine("i:{0}", i);
```

Câu lệnh nhảy break và continue

Khi đang thực hiện các lệnh trong vòng lặp, có yêu cầu như sau: không thực hiện các lệnh còn lại nữa mà thoát khỏi vòng lặp, hay không thực hiện các công việc còn lại của vòng lặp hiện tại mà nhảy qua vòng lặp tiếp theo. Để đáp ứng yêu cầu trên C# cung cấp hai lệnh nhảy là break và continue để thoát khỏi vòng lặp.

Break khi được sử dụng sẽ đưa chương trình thoát khỏi vòng lặp và tiếp tục thực hiện các lệnh tiếp ngay sau vòng lặp.

Continue ngừng thực hiện các công việc còn lại của vòng lặp hiện thời và quay về đầu vòng lặp để thực hiện bước lặp tiếp theo.

Ví dụ:

```
public class UsingBreak_Continue
{
    public static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            if (i % 2 == 0) continue;
            Console.WriteLine("i:{0}", i);
            if (i==7) break;
        }
        Console.ReadLine();
    }
}
```

Nếu không có lệnh **break** và **continue** vòng lặp sẽ lần lượt in ra các số từ 0 đến 9 nhưng khi gặp **I** chẵn ($i \% 2 == 0$) thì nó sẽ **continue** – tức là không thực hiện các lệnh tiếp theo mà quay trở lại đầu vòng lặp với giá trị của **I** được tăng lên 1. Lệnh **break** được thực hiện khi ($i == 7$) nó sẽ thoát khỏi vòng lặp ngay lập tức và cũng kết thúc chương trình và kết quả là chương trình trên chỉ in ra các số lẻ từ 1 đến 7

Bài 6: Toán tử

1. Định nghĩa toán tử

Toán tử được kí hiệu bằng một biểu tượng dùng để thực hiện một hành động. Các kiểu dữ liệu cơ bản của C# như kiểu nguyên hỗ trợ rất nhiều các toán tử như toán tử gán, toán tử toán học, logic...

2. Các loại toán tử

a. Toán tử gán

Đến lúc này toán tử gán khá quen thuộc với chúng ta, hầu hết các chương trình minh họa từ đầu đều đã sử dụng phép gán. Toán tử gán hay phép gán làm cho toán hạng bên trái thay đổi giá trị bằng với giá trị của toán hạng bên phải. Toán tử gán là toán tử hai ngôi. Đây là toán tử đơn giản nhất thông dụng nhất và cũng dễ sử dụng nhất. ví dụ $a = b$

b. Toán tử toán học

Ngôn ngữ C# cung cấp năm toán tử toán học, bao gồm bốn toán tử đầu các phép toán cơ bản. Toán tử cuối cùng là toán tử chia nguyên lấy phần dư.

Các phép toán số học cơ bản (+, -, *, /)

Các phép toán này không thể thiếu trong bất cứ ngôn ngữ lập trình nào, C# cũng không ngoại lệ, các phép toán số học đơn giản nhưng rất cần thiết bao gồm: phép cộng (+), phép trừ (-), phép nhân (*), phép chia (/) nguyên và không nguyên %. Khi chia hai số nguyên, thì C# sẽ bỏ phần phân số, hay bỏ phần dư, tức là nếu ta chia $8/3$ thì sẽ được kết quả là 2 và sẽ bỏ phần dư là 2. Tuy nhiên, khi chia cho số thực có kiểu như float, double, hay decimal thì kết quả chia được trả về là một số thực.

Phép toán chia lấy dư

Để tìm phần dư của phép chia nguyên, chúng ta sử dụng toán tử chia lấy dư (%). Ví dụ, câu lệnh sau $8 \% 3$ thì kết quả trả về là 2 (đây là phần dư còn lại của phép chia nguyên). Thật sự phép toán chia lấy dư rất hữu dụng cho người lập trình. Khi chúng ta thực hiện một phép chia dư n cho một số khác, nếu số này là bội số của n thì kết quả của phép chia dư là 0.

Ví dụ $20 \% 5 = 0$ vì 20 là một bội số của 5. Điều này cho phép chúng ta ứng dụng trong vòng lặp, khi muốn thực hiện một công việc nào đó cách khoảng n lần, ta chỉ cần kiểm tra phép chia dư n, nếu kết quả bằng 0 thì thực hiện công việc. Cách sử dụng này đã áp dụng trong ví dụ minh họa sử dụng vòng lặp for bên trên.

Ví dụ: Phép chia và phép chia lấy dư.

```
using System;
class Tester
{
    public static void Main()
    {
        int i1, i2;
        float f1, f2;
        double d1, d2;
        decimal dec1, dec2;
        i1 = 17;
        i2 = 4;
        f1 = 17f;
        f2 = 4f;
        d1 = 17;
        d2 = 4;
        dec1 = 17;
        dec2 = 4;
        Console.WriteLine("Integer: \t{0}", i1/i2);
        Console.WriteLine("Float: \t{0}", f1/f2);
        Console.WriteLine("Double: \t{0}", d1/d2);
        Console.WriteLine("Decimal: \t{0}", dec1/dec2);
        Console.WriteLine("\nModulus: : \t{0}", i1%i2);
        Console.ReadLine();
    }
}
```

Kết quả:

Integer: 4

float: 4.25

double: 4.25

decimal: 4.25

Modulus: 1

c.Toán tử tăng và giảm

Khi sử dụng các biến số ta thường có thao tác là cộng một giá trị vào biến, trừ đi một giá trị từ biến đó, hay thực hiện các tính toán thay đổi giá trị của biến sau đó gán giá trị mới vừa tính toán cho

chính biến đó.

d. Tính toán và gán trở lại

Giả sử chúng ta có một biến tên `Luong` lưu giá trị lương của một người, biến `Luong` này có giá trị hiện thời là 1.500.000, sau đó để tăng thêm 200.000 ta có thể viết như sau:

```
Luong = Luong + 200.000;
```

Trong câu lệnh trên phép cộng được thực hiện trước, khi đó kết quả của vế phải là 1.700.000 và kết quả này sẽ được gán lại cho biến `Luong`, cuối cùng `Luong` có giá trị là 1.700.000. Chúng ta có thể thực hiện việc thay đổi giá trị rồi gán lại cho biến với bất kỳ phép toán số học nào:

```
Luong = Luong * 2;
```

```
Luong = Luong - 100.000;
```

...

Do việc tăng hay giảm giá trị của một biến rất thường xảy ra trong khi tính toán nên C# cung cấp các phép toán tự gán (self-assignment). Bảng sau liệt kê các phép toán tự gán.

Toán tử	Ý nghĩa
<code>+=</code>	Cộng thêm giá trị toán hạng bên phải vào giá trị toán hạng bên trái
<code>-=</code>	Toán hạng bên trái được trừ bớt đi một lượng bằng giá trị của toán hạng bên phải
<code>*=</code>	Toán hạng bên trái được nhân với một lượng bằng giá trị của toán hạng bên phải.
<code>/=</code>	Toán hạng bên trái được chia với một lượng bằng giá trị của toán hạng bên phải.
	Toán hạng bên trái được chia lấy dư với

%=	một lượng bằng giá trị của toán hạng bên phải.
----	--

Bảng mô tả các phép toán tự gán.

Dựa trên các phép toán tự gán trong bảng ta có thể thay thế các lệnh tăng giảm lương như sau:

```
Luong += 200.000;
```

```
Luong *= 2;
```

```
Luong -= 100.000;
```

Kết quả của lệnh thứ nhất là giá trị của Luong sẽ tăng thêm 200.000, lệnh thứ hai sẽ làm cho giá trị Luong nhân đôi tức là tăng gấp 2 lần, và lệnh cuối cùng sẽ trừ bớt 100.000 của Luong. Do việc tăng hay giảm 1 rất phổ biến trong lập trình nên C# cung cấp hai toán tử đặc biệt là tăng một (++) hay giảm một (--).

Khi đó muốn tăng đi một giá trị của biến đếm trong vòng lặp ta có thể viết như sau:

```
bienDem++;
```

e. Toán tử tăng giảm tiền tố và tăng giảm hậu tố

Giả sử muốn kết hợp các phép toán như gia tăng giá trị của một biến và gán giá trị của biến cho biến thứ hai, ta viết như sau:

```
var1 = var2++;
```

Câu hỏi được đặt ra là gán giá trị trước khi cộng hay gán giá trị sau khi đã cộng. Hay nói cách khác giá trị ban đầu của biến var2 là 10, sau khi thực hiện ta muốn giá trị của var1 là 10, var2 là 11, hay var1 là 11, var2 cũng 11?

Để giải quyết yêu cầu trên C# cung cấp thứ tự thực hiện phép toán tăng/giảm với phép toán gán, thứ tự này được gọi là tiền tố (prefix) hay hậu tố (postfix). Do đó ta có thể viết: `var1 = var2++; //`
Hậu tố

Khi lệnh này được thực hiện thì phép gán sẽ được thực hiện trước tiên, sau đó mới đến phép toán tăng. Kết quả là var1 = 10 và var2 = 11. Còn đối với trường hợp tiền tố: var1 = ++var2;

Khi đó phép tăng sẽ được thực hiện trước tức là giá trị của biến var2 sẽ là 11 và cuối cùng phép gán được thực hiện. Kết quả cả hai biến var1 và var2 đều có giá trị là 11.

Minh họa sử dụng toán tử tăng trước và tăng sau khi gán.

```
using System;
class Tester
{
    static int Main()
    {
        int valueOne = 10;
        int valueTwo;
        valueTwo = valueOne++;
        Console.WriteLine("Thực hiện tăng sau: {0}, {1}",
            valueOne, valueTwo);
        valueOne = 20;
        valueTwo = ++valueOne;
        Console.WriteLine("Thực hiện tăng trước: {0}, {1}",
            valueOne, valueTwo);
        Console.ReadLine();
        return 0;
    }
}
```

Kết quả:

Thực hiện tăng sau: 11, 10

Thực hiện tăng trước: 21, 21

f. Toán tử quan hệ

Những toán tử quan hệ được dùng để so sánh giữa hai giá trị, và sau đó trả về kết quả là một giá trị logic kiểu bool (true hay false). Ví dụ toán tử so sánh lớn hơn (>) trả về giá trị là true nếu giá trị bên trái của toán tử lớn hơn giá trị bên phải của toán tử. Do vậy 5 > 2 trả về một giá trị là true, trong khi 2 > 5 trả về giá trị false. Các toán tử quan hệ trong ngôn ngữ C# được trình bày ở bảng 3.4 bên dưới. Các toán tử trong bảng được minh họa với hai biến là value1 và value2, trong đó value1 có giá trị là 100 và value2 có giá trị là 50.

Tên toán tử	Kí hiệu	Biểu thức so sánh	Kết quả
So sánh bằng	==	Value1==100	True
		Value1==50	False

Không bằng	!=	Value2 !=100 Value2 !=50	False True
Lớn hơn	>	Value1> value2 Value2> value1	True False
Lớn hơn hoặc bằng	>=	Value2 >= 50	True
Nhỏ hơn	<	Value1<value2 Value2<value1	False True
Nhỏ hơn hoặc bằng	<=	Value1<=value2	False

Các toán tử so sánh (giả sử value1 = 100, và value2 = 50).

Như trong bảng 3.4 trên ta lưu ý toán tử so sánh bằng (==), toán tử này được ký hiệu bởi hai dấu bằng (=) liền nhau và cùng trên một hàng, không có bất kỳ khoảng trống nào xuất hiện giữa chúng. Trình biên dịch C# xem hai dấu này như một toán tử.

g. Toán tử logic

Trong câu lệnh if mà chúng ta đã tìm hiểu trong phần trước, thì khi điều kiện là true thì biểu thức bên trong if mới được thực hiện. Đôi khi chúng ta muốn kết hợp nhiều điều kiện với nhau như: bắt buộc cả hai hay nhiều điều kiện phải đúng hoặc chỉ cần một trong các điều kiện đúng là đủ hoặc không có điều kiện nào đúng...C# cung cấp một tập hợp các toán tử logic để phục vụ cho người lập trình.

Bảng sau liệt kê ba phép toán logic, bảng này cũng sử dụng hai biến minh họa là x, và y trong đó x có giá trị là 5 còn y có giá trị là 7

Tên toán tử	Kí hiệu	Biểu thức logic	Giá trị	Logic
And	&&	(x==3)&&(y==7)	False	Cả hai điều kiện phải đúng
Or		(x==3) (y==7)	True	Chỉ cần một điều kiện đúng
Not	!	! (x==3)	True	Biểu thức trong ngoặc

				phải sai
--	--	--	--	----------

h.Toán tử ba ngôi

Hầu hết các toán tử đòi hỏi có một toán hạng như toán tử (++ , --) hay hai toán hạng như (+, -, *, /, ...). Tuy nhiên, C# còn cung cấp thêm một toán tử có ba toán hạng (? :). Toán tử này có cú pháp sử dụng như sau:

<Biểu thức điều kiện > ? <Biểu thức thứ 1> : <Biểu thức thứ 2>

Toán tử này sẽ xác định giá trị của một biểu thức điều kiện, và biểu thức điều kiện này phải trả về một giá trị kiểu bool. Khi điều kiện đúng thì <biểu thức thứ 1> sẽ được thực hiện, còn ngược lại điều kiện sai thì <biểu thức thứ 2> sẽ được thực hiện. Có thể diễn giải theo ngôn ngữ tự nhiên thì toán tử này có ý nghĩa : "Nếu điều kiện đúng thì làm công việc thứ nhất, còn ngược lại điều kiện sai thì làm công việc thứ hai". Cách sử dụng toán tử ba ngôi này được minh họa trong ví dụ sau.

Sử dụng toán tử ba ngôi.

```
using System;
class Tester
{
    public static int Main()
    {
        int value1;
        int value2;
        int maxvalue;
        value1 = 10;
        value2 = 20;
        maxvalue = value1 > value2 ? value1 : value2;
        Console.WriteLine("Giá trị thu nhất {0}, giá trị thu hai {1}, giá trị lớn nhất {2}", value1,
value2, maxvalue);
        Console.ReadLine();
        return 0;
    }
}
```

Trong ví dụ minh họa trên toán tử ba ngôi được sử dụng để kiểm tra xem giá trị của value1 có lớn hơn giá trị của value2, nếu đúng thì trả về giá trị của value1, tức là gán giá trị value1 cho biến maxvalue, còn ngược lại thì gán giá trị value2 cho biến maxvalue.

Bài 7: Xử lý ngoại lệ, các lệnh throw, catch, finally.

1. Định nghĩa ngoại lệ và trình xử lý ngoại lệ

Ngoại lệ: Là một *đối tượng* đóng gói những thông tin về sự cố của một chương trình không bình thường.

Một trình xử lý ngoại lệ:

Là một khối lệnh chương trình được thiết kế xử lý các ngoại lệ mà chương trình phát sinh.

Xử lý ngoại lệ được thực thi trong câu lệnh catch. Một cách lý tưởng thì nếu một ngoại lệ được bắt và được xử lý, thì chương trình có thể sửa chữa được vấn đề và tiếp tục thực hiện hoạt động. Thậm chí nếu chương trình không tiếp tục, bằng việc bắt giữ ngoại lệ chúng ta có cơ hội để in ra những thông điệp có ý nghĩa và kết thúc chương trình một cách rõ ràng. Nếu đoạn chương trình của chúng ta thực hiện mà không quan tâm đến bất cứ ngoại lệ nào mà chúng ta có thể gặp (như khi giải phóng tài nguyên mà chương trình được cấp phát), chúng ta có thể đặt đoạn mã này trong khối finally, khi đó nó sẽ chắc chắn sẽ được thực hiện thậm chí ngay cả khi có một ngoại lệ xuất hiện.

Phát sinh và bắt giữ ngoại lệ

Trong ngôn ngữ C#, chúng ta chỉ có thể phát sinh (throw) những đối tượng các kiểu dữ liệu là System.Exception, hay những đối tượng được dẫn xuất từ kiểu dữ liệu này.

Namespace System của CLR chứa một số các kiểu dữ liệu xử lý ngoại lệ mà chúng ta có thể sử dụng trong chương trình. Những kiểu dữ liệu ngoại lệ này bao gồm ArgumentNullException, InvalidCastException, và OverflowException, cũng như nhiều lớp khác nữa.

2. Lệnh Throw

Cú pháp: throw new System.Exception();

Khi phát sinh ngoại lệ thì ngay tức khắc sẽ làm ngừng việc thực thi trong khi CLR sẽ tìm

kiểm một trình xử lý ngoại lệ. Nếu một trình xử lý ngoại lệ không được tìm thấy trong phương thức hiện thời, thì CLR tiếp tục tìm trong phương thức gọi cho đến khi nào tìm thấy. Nếu CLR trả về lớp Main() mà không tìm thấy bất cứ trình xử lý ngoại lệ nào, thì nó sẽ kết thúc chương trình.

Ví dụ:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Programming_CSharp
{
    public class Test
    {
        public static void Main()
        {
            Console.WriteLine("hàm Main....");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Kết thúc hàm Main...");
        }
        public void Func1()
        {
            Console.WriteLine("Bắt đầu hàm Func1...");
            Func2();
            Console.WriteLine("Kết thúc hàm Func1...");
        }
        public void Func2()
        {
            Console.WriteLine("Bắt đầu hàm Func2...");
            throw new System.Exception();
            Console.WriteLine("Kết thúc hàm Func2...");
        }
    }
}
```

Giải thích ví dụ trên như sau: Hàm Main() gọi hàm Func1(). Hàm Func1() thực hiện lệnh in ra màn hình dòng "bắt đầu hàm Func1" sau đó nó gọi tới hàm Func2(). Hàm Func2() lại in ra dòng "bắt đầu hàm Func2" sau đó nó sẽ phát sinh ra một ngoại lệ dùng câu lệnh throw new

System.Exception(). Tại đây chương trình bị ngừng thực thi, CLR sẽ tìm kiếm trình xử lý ngoại lệ cho ngoại lệ hàm Func2() phát sinh. CLR sẽ lần lượt tìm kiếm trong stack, ở hàm Func1() nhưng không có trình xử lý ngoại lệ nào, nó sẽ tiếp tục tìm đến hàm main nhưng ở hàm này cũng không có nên CLR sẽ gọi trình xử lý ngoại lệ mặc định, nó sẽ xuất ra một thông điệp lỗi như các bạn thấy khi thực thi chương trình.

3. Lệnh Try Catch

Trong C#, một trình xử lý ngoại lệ hay một đoạn chương trình xử lý các ngoại lệ được gọi là một khối catch và được tạo ra với từ khóa catch.

Chúng ta sẽ viết lại ví dụ trên nhưng đặt throw vào trong khối try và một khối catch sẽ dùng để xử lý ngoại lệ do lệnh throw phát sinh. Khối catch sẽ đưa ra thông báo là đã có một lỗi được xử lý.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Programming_CSharp
{
    public class Test
    {
        public static void Main()
        {
            Console.WriteLine("hàm Main....");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Kết thúc hàm Main...");
            Console.ReadLine();
        }
        public void Func1()
        {
            Console.WriteLine("Bắt đầu hàm Func1...");
            Func2();
            Console.WriteLine("Kết thúc hàm Func1...");
        }
        public void Func2()
        {

```

```

Console.WriteLine("Bắt đầu hàm Func2...");
try
{
    Console.WriteLine("Bắt đầu Khối try");
    throw new System.Exception();
    Console.WriteLine("Kết thúc khối try");
}
catch
{
    Console.WriteLine("Ngoại lệ đã được xử lý");
}
Console.WriteLine("Kết thúc hàm Func2...");
}
}
}

```

Tương tự như ví dụ tôi đã vừa trình bày, cho đến khi chương trình thực hiện hàm Func2() khi lệnh throw phát sinh ra ngoại lệ, chương trình sẽ bị ngừng thực hiện và CLR sẽ tìm phần xử lý ngoại lệ trong stack, đầu tiên nó sẽ gọi đến hàm Func1() tại đây hàm Func2() được gọi và nó sẽ tìm thấy phần xử lý ngoại lệ trong khối catch, nó sẽ in ra dòng "Ngoại lệ đã được xử lý". Đó cũng là lý do mà chương trình sẽ không bao giờ in ra dòng "Kết thúc khối try".

4. Lệnh Finally

Trong một số tình huống chúng ta cần phải thực hiện bất cứ khi nào một ngoại lệ được phát sinh ra, ví dụ như việc đóng một tập tin. Để làm việc này chúng ta có thể đặt câu lệnh trong cả hai khối try và catch. Tuy nhiên có một cách giải quyết tốt hơn, đó là sử dụng câu lệnh Finally. Các hành động đặt trong khối finally sẽ luôn được thực hiện mà không cần quan tâm tới việc có hay không một ngoại lệ phát sinh trong chương trình.

Chúng ta cùng xét ví dụ sau:

```

using System;
namespace Programming_CSharp
{
    public class Test
    {
        public static void Main()
        {
            Test t = new Test();
        }
    }
}

```

```

        t.TestFunc();
        Console.ReadLine();
    }
    // chia hai số và xử lý ngoại lệ nếu có
    public void TestFunc()
    {
        try
        {
            Console.WriteLine("mở file");
            double a = 5;
            double b = 0;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
            Console.WriteLine("dòng này có thể xuất hiện hoặc không");
        }
        catch (System.DivideByZeroException)
        {
            Console.WriteLine("lỗi chia cho 0!");
        }
        catch
        {
            Console.WriteLine("không có ngoại lệ");
        }
        finally
        {
            Console.WriteLine("Đóng tệp.");
        }
    }
    // thực hiện chia nếu hợp lệ
    public double DoDivide(double a, double b)
    {
        if ( b == 0)
        {
            throw new System.DivideByZeroException();
        }
        if ( a == 0)
        {
            throw new System.ArithmeticException();
        }
        return a/b;
    }
}

```

Đầu tiên hãy gán a= 5 và b=0 chạy chương trình Bạn sẽ thấy lệnh `Console.WriteLine("dòng này có thể xuất hiện hoặc không");`;

Sẽ không được thực hiện do xuất hiện một ngoại lệ là lỗi chia cho 0 và chương trình sẽ tìm tới phần xử lý ngoại lệ này mà bỏ qua phần lệnh tiếp theo.

Sau đó bạn thay đổi giá trị $b=12$ và chạy chương trình thì lệnh

`Console.WriteLine("dòng này có thể xuất hiện hoặc không");` được thực hiện.

Tuy nhiên ở cả 2 trường hợp bạn đều thấy thực hiện lệnh `Console.WriteLine("Đóng tệp.");`

Đó là vì lệnh này đã được đặt trong khối Finally.

Nắm được cách xử lý ngoại lệ qua việc sử dụng các câu lệnh throw, catch và finally sẽ giúp bạn lập trình có hiệu quả hơn.

Bài 8:Lớp và đối tượng

1. Lớp và đối tượng

a.

Lớp (class)

Một lớp là một khái niệm mô tả cho những thực thể có chung tính chất và hành vi. Lớp định nghĩa những thuộc tính và hành vi được dùng cho những đối tượng của lớp đó. Do đó có thể nói lớp là một khuôn mẫu cho các đối tượng.

Công thức để tạo một class

```
AccessModifier class className
{
    // thân class
}
```

Ví dụ tôi định nghĩa một class là color, class này được truy cập public

```
public class Color
{
    // Nội dung class
}
```

b. Đối tượng

Đối tượng là một đại diện, hay có thể nói là một sản phẩm của một class. Tất cả các đối tượng đều có chung những thuộc tính và hành vi mà class định nghĩa. Cách tạo đối tượng giống như cách tạo một biến có kiểu dữ liệu là Class.

Ví dụ tôi muốn khai báo một đối tượng c là thể hiện của class color nói trên. Tôi làm như sau:

```
Color c = new Color();
```

Sau đó từ đối tượng c này tôi sẽ truy cập đến các thành phần của class thông qua toán tử "." (Mà tôi đã giới thiệu ở bài học số 2). Tuy nhiên việc truy cập các thành phần của class còn tùy thuộc vào thành phần đó là **instance** hay **static**. Bạn sẽ được tìm hiểu khái niệm này ở phần tiếp theo của bài học.

c. Ưu điểm khi sử dụng lớp và đối tượng trong lập trình

Có một số những ưu điểm của việc sử dụng Class và đối tượng trong phát triển phần mềm. Những ưu điểm nổi bật nhất được liệt kê như sau:

- Duy trì code bằng việc mô hình hóa
- Đóng gói những sự phức tạp trong mã lệnh từ người dùng
- Khả năng sử dụng lại
- Cung cấp đơn kế thừa để thực thi nhiều phương thức.

Các bạn sẽ hiểu hơn về điều này ở các bài học sau của chúng tôi về inheritance (sự thừa kế) hay polymorphism (Tính đa hình).

2. Khái niệm thành viên thể hiện (instance) và thành viên tĩnh(static)

Trong một lớp, các thành viên có thể là instance (thành viên thể hiện) hoặc static (thành viên tĩnh). Một thành viên instance có nghĩa là thành viên đó liên quan đến thể hiện của một kiểu dữ liệu. Thành viên static được xem như một phần của lớp. Chúng ta truy cập đến thành viên tĩnh của một lớp thông qua tên lớp đã được khai báo. Còn các thành viên instance thì phải thông qua thể hiện của lớp.

Giả sử chúng ta có một lớp là class A và có 2 thể hiện là t1, t2 và một phương thức tĩnh là a(). Để truy cập phương thức này ta viết A.a(). Chứ không thể viết t1. a() hoặc t2.a() vì trong C# không cho phép truy cập đến phương thức cũng như các biến thành viên tĩnh thông qua một thể hiện.

Một số ngôn ngữ thì có sự phân chia giữa phương thức của lớp và các phương thức khác (toàn cục) tồn tại bên ngoài không phụ thuộc bất cứ một lớp nào. Tuy nhiên, điều này không cho phép trong C#, ngôn ngữ C# không cho phép tạo các phương thức bên ngoài của lớp, nhưng ta có thể tạo được các phương thức giống như vậy bằng cách tạo các phương thức tĩnh bên trong một lớp. Phương thức tĩnh ít nhiều giống phương thức toàn cục bởi ta có thể truy cập đến nó không phải thông qua bất cứ một thể hiện nào của lớp chứa nó. Tuy nhiên phương thức tĩnh sẽ hoạt động tốt hơn phương

thức toàn cục vì nó luôn được đặt trong phạm vi một lớp do đó chúng ta sẽ tránh được tình trạng lộn xộn do bị trùng tên giữa các phương thức đặt trong namespace.

3. Các thành phần của lớp

a. Fields

Field là một phần tử dùng để thể hiện các biến trong lớp – các biến hoặc các thể hiện của một lớp dự:

```
class Color
{
    internal ushort redPart;
    internal ushort bluePart;
    internal ushort greenPart;
    public Color(ushort red, ushort blue, ushort green)
    {
        redPart = red;
        bluePart = blue;
        greenPart = green;
    }
}
```

Lớp color chứa các instance fields như là redPart, bluePart, và greenPart.

Fields có thể là static như ví dụ dưới đây:

```
Class Color
{
    public static Color Red = new Color(0xFF, 0, 0);
    public static Color Blue = new Color(0, 0xFF, 0);
    public static Color Green = new Color(0, 0, 0xFF);
    public static Color White = new Color(0xFF, 0xFF, 0xFF);
    ...
}
```

Các phần tử Red, Blue, Green, white đều là các phần tử static.

b. Properties

Property (thuộc tính, đặc tính) là một phần tử dùng để truy cập đến đặc điểm của một đối tượng hoặc một class (lớp). Ví dụ như là độ dài một chuỗi, kích cỡ của font chữ, độ rộng của một cửa sổ, tên của một customer ...Property là phần mở rộng của fields. Cả 2 đều được gọi tên với các kiểu kết hợp, và cách truy cập đến fields và properties là như nhau. Tuy nhiên khác với fields,

properties không chỉ rõ nơi lưu trữ nó. Thay vào đó, properties có cách truy nhập là dùng câu lệnh để thi hành việc đọc và ghi giá trị.

Property được định nghĩa bằng 2 phần. Phần thứ nhất giống như cách định nghĩa Fields. Phần thứ 2 chứa phần tử truy cập get và set. Các bạn hãy xem ví dụ dưới đây :

```
public class Button
{
    private string caption;
    public string Caption
    {
        get
        {
            return caption;
        }
        set
        {
            caption = value;
            Repaint();
        }
    }
}
```

c. **Methods**

Bạn sẽ được học về methods ở bài sau

Bài 9: Methods (Phương thức) và các vấn đề liên quan

1. Khái niệm về method

Method (Tiếng việt gọi là hàm hoặc phương thức) là một thành phần của class dùng để thực thi một công việc nào đó của một đối tượng hoặc một class. Một method sẽ chứa một danh sách các đối số(hoặc có thể không có đối số), một giá trị trả về (trừ void method). Method có thể là static (tĩnh) hoặc non- static (hay là instance method- phương thức thể hiện). Đây là ví dụ về method:

```
public class Stack
{
    public static Stack Clone(Stack s) {...}
    public static Stack Flip(Stack s) {...}
    public object Pop() {...}
    public void Push(object o) {...}
    public override string ToString() {...}
    ...
}
```

```

}
class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 1; i < 10; i++)
            s.Push(i);
        Stack flipped = Stack.Flip(s);
        Stack cloned = Stack.Clone(s);
        Console.WriteLine("Original stack: " + s.ToString());
        Console.WriteLine("Flipped stack: " + flipped.ToString());
        Console.WriteLine("Cloned stack: " + cloned.ToString());
    }
}

```

Class Stack có 2 phương thức tĩnh (static method) là Clone và Flip và các phương thức thể hiện là Push, Push, Pop, and ToString. Một lần nữa bạn thấy các phương thức tĩnh được gọi trực tiếp từ tên của lớp còn phương thức thể hiện phải được gọi qua thể hiện của lớp. Phương thức có thể được overloaded . Có nghĩa là nhiều methods có tên giống nhau. Bạn sẽ được học về overloading methods trong bài tới.

2. Truyền tham số cho method

Việc truyền tham số vào cho phương thức chỉ được thực hiện đối với các kiểu dữ liệu giá trị. Vậy cách thức truyền tham số sẽ được thực hiện như thế nào? Trong C# các tham số được truyền vào hàm thông qua 2 cách là truyền theo giá trị (truyền tham trị) và truyền theo địa chỉ (truyền tham chiếu). Chúng ta sẽ lần lượt tìm hiểu hai các truyền này.

a. Truyền tham trị

Khi một đối tượng có kiểu giá trị được truyền giá trị vào cho một phương thức thì có một bản sao chép đối tượng đó được tạo ra bên trong phương thức. Khi phương thức thực hiện xong thì đối tượng sao chép này sẽ được hủy.

Dưới đây là ví dụ về cách truyền tham trị:

```

using System;
public class Time
{
    public void DisplayCurrentTime()
    {
        Console.WriteLine("{0}/{1}/{2}/ {3}:{4}:{5}", Date,
            Month, Year, Hour, Minute, Second);
    }
    public int GetHour()

```

```
{
    return Hour;
}
public void GetTime(int h, int m, int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
public Time( System.DateTime dt)
{
    Year = dt.Year;
    Month = dt.Month;
    Date = dt.Day;
    Hour = dt.Hour;
    Minute = dt.Minute;
    Second = dt.Second;
}
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
}
public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time( currentTime);
        t.DisplayCurrentTime();
        int theHour = 0;
        int theMinute = 0;
        int theSecond = 0;
        t.GetTime( theHour, theMinute, theSecond);
        System.Console.WriteLine("Current time: {0}:{1}:{2}",
            theHour, theMinute, theSecond);
    }
}
```

Kết quả:

8/6/2002 14:15:20

Current time: 0:0:0

Như các bạn thấy kết quả in ra Current time: 0:0:0 tức là các biến theHour, theMinute, theSecond vẫn giữ nguyên giá trị của nó sau hàm GetTime.

b. Truyền tham chiếu

Như tôi đã nói trên, **mỗi phương thức sẽ có một giá trị duy nhất được trả về**, mặc dù giá trị này có thể là một tập hợp các giá trị. Đôi khi chúng ta muốn phương thức trả về nhiều hơn một giá trị. Cách thực hiện là tạo ra cách tham số dưới hình thức tham chiếu. Khi bạn truyền tham chiếu, trong phương thức bạn sẽ xử lý và gán các giá trị mới cho các tham chiếu này và kết quả là sau khi phương thức thực hiện xong ta dùng các tham số truyền vào như là các kết quả trả về. Để làm việc này bạn phải thêm từ khóa ref (viết tắt của reference) vào trước các tham số trong phần khai báo phương thức và lời gọi phương thức.

Ví dụ để các biến theHour, theMinute, theSecond trong ví dụ trên sau khi được xử lý trong GetTime sẽ có giá trị như chúng vừa được gán, chúng ta sẽ phải làm như sau:

Đầu tiên thêm khai báo ref vào trước các tham số trong phương thức GetTime():

```
public void GetTime(ref int h, ref int m, ref int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
```

Tiếp theo là sẽ gọi phương thức GetTime dưới dạng truyền tham chiếu như sau:

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
```

Tóm lại cơ chế truyền tham số dạng tham chiếu sẽ thực hiện trên chính đối tượng được đưa vào. Còn cơ chế truyền tham số giá trị thì sẽ tạo ra các bản sao các đối tượng được truyền vào, do đó mọi thay đổi bên trong phương thức không làm ảnh hưởng đến các đối tượng được truyền vào dưới dạng giá trị.

Bài 10: Overloading Method

Method Overloading xuất hiện khi trong một class có từ hai hàm có cùng tên. Có hai kiểu Method Overloading:

- Function Overloading dựa trên kiểu giá trị tham số truyền vào.
- Function Overloading dựa trên số lượng tham số truyền vào.

Ví dụ

```
class Library
{
    // Function Overloading
    public void insertbooks(int id)
    {
        //
    }
    public void insertbooks(int id, int type)
    {
        //
    }
    public void insertbooks(string id, int type)
    {
        //
    }
}
```

Ba hàm insertbooks ở trên là một ví dụ về function overloading trong lập trình C#. Trong khi hàm thứ nhất và thứ 2 là overloading theo số lượng tham số, và hàm thứ 3 với hàm thứ 2 là overloading theo kiểu tham số truyền vào.

Bài 11: constructor & Destructor

Nội dung bài học

- *Khái niệm constructor và destructor*
- *Instance constructor và static constructor*

1. Khái niệm về constructor và destructor

a.

Constructor

Constructors (Tạm dịch là phương thức khởi tạo) là những hàm đặc biệt cho phép thực thi, điều khiển chương trình ngay khi khởi tạo đối tượng. Trong C#, Constructors có tên giống như tên của Class và không có giá trị trả về.

Ví dụ:

```
class Library
```



```

{
    private int ibooktypes;
    //Constructor
    public Library()
    {
        ibooktypes = 7;
    }
    public Library(int value)
    {
        ibooktypes = value;
    }
}

```

b. Destructor

Destructor (tạm dịch là phương thức hủy) là một hàm dùng để hủy đi một thể hiện của một class. Destructor không có đối số, không có từ chỉ thuộc tính truy nhập, và không được gọi tường minh. Destructor của một thể hiện sẽ được gọi tự động khi một thể hiện kết thúc "vòng đời" của nó thông qua bộ thu dọn rác tự động (Garbage Collection). Destructor cũng có tên trùng với tên class. Để khai báo một destructor chúng ta đặt dấu "~" vào trước destructor

Ví dụ

```

class Library
{
    private int ibooktypes;
    //Constructor
    public Library()
    {
        ibooktypes = 7;
    }
    public Library(int value)
    {
        ibooktypes = value;
    }
    //Destructor
    ~ Library()
    {
        //thực thi câu lệnh
    }
}

```

~Library() là destructor của lớp Library

2. Instance constructor và Static constructor

Như bài học trước tôi đã giới thiệu, Các thành phần trong class có thể là instance và static. Constructors cũng là một thành phần của class, vậy Instance constructors và Static constructors có gì khác nhau? Chúng ta sẽ cùng tìm hiểu.

a. Instance constructor

Một instance constructor(tạm dịch là một bộ khởi dựng thể hiện) sẽ khởi tạo một số giá trị khi một thể hiện của một lớp được tạo ra.

Ví dụ:

```
class Point
{
    public double x, y;
    public Point()
    {
        this.x = 0;
        this.y = 0;
    }
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public static double Distance(Point a, Point b)
    {
        double xdiff = a.x - b.x;
        double ydiff = a.y - b.y;
        return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
    }
    public override string ToString()
    {
        return string.Format("{0}, {1}", x, y);
    }
}
class Test
{
    static void Main()
    {
        Point a = new Point();
        Point b = new Point(3, 4);
        double d = Point.Distance(a, b);
    }
}
```

```

        Console.WriteLine("Distance from {0} to {1} is {2}", a, b, d);
    }
}

```

Trong lớp Point có 2 instance constructors. Một không có đối số truyền vào và 2 constructor còn lại có 2 tham số kiểu double. Nếu class không có instance constructor nào thì constructor không có đối số sẽ được gọi tự động.

b. Static constructor

Nếu một lớp khai báo một phương thức khởi tạo tĩnh (static constructor), thì được đảm bảo rằng phương thức này sẽ được thực hiện trước bất cứ thể hiện nào của lớp được tạo ra. Static Constructor hữu dụng khi chúng ta cần cài đặt một số công việc mà không thể thực hiện được thông qua chức năng khởi dựng và công việc cài đặt này chỉ được thực duy nhất một lần. Static constructor không có thuộc tính truy cập, không có đối số và không được gọi tường minh mà sẽ được gọi tự động.

Ví dụ:

```

using System;
using Personnel.Data;
class Employee
{
    private static DataSet ds;
    static Employee() {
        ds = new DataSet(...);
    }
    public string Name;
    public decimal Salary;
    ...
}

```

Khi đó đối tượng ds sẽ được tạo ra khi trước khi ta tạo một thể hiện lớp employee.

Bài 12: Inheritance - Sự kế thừa

Có 2 kiểu kế thừa trong lập trình hướng đối tượng là đơn kế thừa (kế thừa từ nhiều lớp) và đa kế thừa (kế thừa từ nhiều lớp). C# chỉ cung cấp mô hình đơn kế thừa. Ví dụ về kế thừa trong C#.

```

/* Ví dụ về thừa kế trong lập trình C# */
using System;
using System.Collections.Generic;
using System.Text;

namespace __OOP_Inheritance

```

```
{
class Program
{
    static void Main(string[] args)
    {
        Dog objDog = new Dog(4);
        objDog.displayProperties();
        Chicken objChicken = new Chicken(2);
        objChicken.displayProperties();
        Console.Read();
    }
}
class Animal
{
    protected int ifoots;
    protected string sName;

    protected void setFoot(int ival)
    {
        ifoots = ival;
    }
    protected void setName(string sVal)
    {
        sName = sVal;
    }
    public void displayProperties()
    {
        Console.WriteLine(sName + " have " + ifoots.ToString()+ " foots");
    }
}
class Dog : Animal
{
    public Dog(int ival)
    {
        setName("Dog");
        ifoots = ival;
    }
}
class Chicken : Animal
{
    public Chicken(int ival)
    {
        setName("Chicken");
        setFoot(ival);
    }
}
```

}

Kết quả khi thực thi chương trình

```
Dog have 4 foots
Chicken have 2 foots
```

Ở ví dụ trên, Dog và Chicken là hai lớp kế thừa từ lớp Animal, do đó các thuộc tính như số chân, ifoots và tên sName đương nhiên xuất hiện trong hai lớp này và cho phép sử dụng. Tương tự, các hàm như *setName()*, *setFoot()*, *displayProperties()* tại lớp Animal cũng được kế thừa xuống hai lớp Dog và Chicken. Do đó ta có thể gọi những hàm này, và kết quả hiển thị khi gọi hàm *displayProperties()* theo đối tượng objDog và objChicken khác nhau như hình trên.

Một lưu ý trong thừa kế đây là Overriding method. Nếu một hàm được định nghĩa trong lớp con có cùng tên, kiểu với hàm trong lớp cha, khi ấy hàm trong lớp con sẽ overrides (làm ẩn) hàm trong lớp cha. Đó được gọi là overriding.

Ví dụ về Overriding:

/ Ví dụ về thừa kế,overrrding trong lập trình C# */*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace __OOP_Inheritance
{
    class Program
    {
        static void Main(string[] args)
        {
            Dog objDog = new Dog(4);
            objDog.displayProperties();
            Chicken objChicken = new Chicken(2);
            objChicken.displayProperties();
            Tiger objTiger = new Tiger(4);
            objTiger.displayProperties();
            Console.Read();
        }
    }
    class Animal
    {
```

```
protected int ifoots;
protected string sName;

protected void setFoot(int ival)
{
    ifoots = ival;
}
protected void setName(string sVal)
{
    sName = sVal;
}
public virtual void displayProperties() // chú ý hàm này
{
    Console.WriteLine(sName + " has " + ifoots.ToString()+ " foots");
}
}
class Dog : Animal
{
    public Dog(int ival)
    {
        setName("Dog");
        ifoots = ival;
    }
}
class Chicken : Animal
{
    public Chicken(int ival)
    {
        setName("Chicken");
        setFoot(ival);
    }
    public void displayProperties()
    {
        base.displayProperties();
        Console.WriteLine(sName + " have " + ifoots.ToString() + " foots (from
Chicken class)");
    }
}

class Tiger : Animal
{
    public Tiger(int ival)
    {
        setFoot(ival);
    }
    public override void displayProperties() // chú ý hàm này
```

```
    {  
        Console.WriteLine("Tiger has " + ifoots.ToString() + " foots");  
    }  
}
```

Kết quả thực hiện chương trình



```
Dog has 4 foots  
Chicken has 2 foots  
Chicken have 2 foots <from Chicken class>  
Tiger has 4 foots  
-
```

Hàm displayProperties() trong lớp Tiger overrides hàm displayProperties() trong lớp Animal.

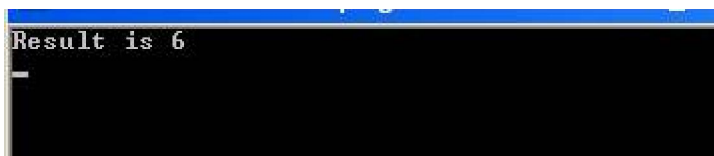
Bài 13: Polymorphism

Ví dụ về polymorphism:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace __OOP_polymorphism  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Child objchild = new Child();  
            Console.WriteLine("Result is " + objchild.methodA().ToString());  
            Console.Read();  
        }  
    }  
    class Parent  
    {  
        public int methodA()  
        {  
            return methodB() * methodC();  
        }  
        public virtual int methodB()
```

```
{
    return 1;
}
public int methodC()
{
    return 2;
}
}
class Child : Parent
{
    public override int methodB()
    {
        return 3;
    }
}
}
```

Kết quả chạy chương trình

A screenshot of a terminal window with a black background and white text. The text "Result is 6" is displayed on the first line. There is a small white cursor or highlight on the line below.

Như bình thường của mô hình kế thừa, kết quả trả về khi gọi hàm methodA() từ đối tượng của lớp Child phải là "Result is 2". Nhưng trong kết quả trên, kết quả là "Result is 6". Kết quả này do hàm methodB() tại lớp Child đã override hàm methodB() tại lớp Parent.

Vậy ta có thể khái quát Polymorphism như sau:

- Polymorphism không chỉ đơn giản là overriding, mà nó là overriding thông minh.
- Khác biệt giữa Overriding và Polymorphism đó là trong Polymorphism, sự quyết định gọi hàm được thực hiện khi chương trình chạy.

Bài 14: Abstract Class và Sealed class

Abstract Class là lớp dùng để định nghĩa những thuộc tính và hành vi chung của những lớp khác. Một Abstract class được dùng như một lớp cha của các lớp khác. Từ khóa **abstract** được dùng để định nghĩa một abstract class. Những lớp được định nghĩa bằng cách dùng từ khóa **abstract** thì không cho phép khởi tạo đối tượng của lớp ấy.


```
abstract class Shape
{
    public abstract float calculateArea();
    public void displaySomething()
    {
        Console.WriteLine("Something is displayed");
    }
}
class Circle:Shape
{
    float radius;
    public override float calculateArea()
    {
        return radius * 22 / 7;
    }
}
```

Khi thực thi chương trình, bạn không thể tạo đối tượng cho lớp Shape, vì nó là abstract class.

Bài 15: Interface

1. Định nghĩa interface (giao diện)

Giao diện là ràng buộc, giao ước đảm bảo cho các lớp hay các cấu trúc sẽ thực hiện một điều gì đó. Khi một lớp thực thi một giao diện, thì lớp này báo cho các thành phần client biết rằng lớp này có hỗ trợ các phương thức, thuộc tính, sự kiện và các chỉ mục khai báo trong giao diện.

Một giao diện thì giống như một lớp chỉ chứa các phương thức trừu tượng. Một lớp trừu tượng được dùng làm lớp cơ sở cho một họ các lớp dẫn xuất từ nó. Trong khi giao diện là sự trộn lẫn với các cây kế thừa khác.

Tuy nhiên bạn phải hiểu là giao diện không phải là lớp.

Sau đây tôi sẽ giới thiệu với bạn cách định nghĩa và thực thi một giao diện

Cú pháp để định nghĩa một giao diện như sau:

[thuộc tính] [phạm vi truy cập] interface <tên giao diện> [: danh sách cơ sở]

```
{  
<phần thân giao diện>  
}
```

Phần thuộc tính chúng ta chưa đề cập tới các bạn hãy lưu ý đến phần phạm vi truy cập bao gồm public, private, protected, internal, và protected internal đã được nói đến. Theo sau từ khóa interface là tên của giao diện. Thông thường tên của giao diện được bắt đầu với từ I hoa (điều này không bắt buộc nhưng việc đặt tên như vậy rất rõ ràng và dễ hiểu, tránh nhầm lẫn với các thành phần khác).

Danh sách cơ sở là danh sách các giao diện mà giao diện này mở rộng, phần này sẽ được trình bày trong phần thực thi nhiều giao diện. Phần thân của giao diện chính là phần thực thi giao diện.

2. Thực thi giao diện

Giả sử chúng ta muốn tạo một giao diện nhằm mô tả những phương thức và thuộc tính của một lớp cần thiết để lưu trữ và truy cập từ một cơ sở dữ liệu hay các thành phần lưu trữ dữ liệu khác như là một tập tin. Chúng ta quyết định gọi giao diện này là IStorage. Trong giao diện này chúng ta xác nhận hai phương thức: Read() và Write(), khai báo này sẽ được xuất hiện trong phần thân của giao diện như sau:

```
interface IStorable  
{  
void Read();  
void Write(object);  
}
```

Mục đích của một giao diện là để định nghĩa những khả năng mà chúng ta muốn có trong một lớp. Ví dụ, chúng ta có thể tạo một lớp tên là Document, lớp này lưu trữ các dữ liệu trong cơ sở dữ liệu, do đó chúng ta quyết định lớp này thực thi giao diện IStorable. Để làm được điều

này, chúng ta sử dụng cú pháp giống như việc tạo một lớp mới Document được thừa kế từ IStorable bằng dùng dấu hai chấm (:) và theo sau là tên giao diện:

```
public class Document : IStorable
{
    public void Read()
    {
        //...
    }
    public void Write()
    {
        //...
    }
}
```

Dưới đây là code minh họa việc thực thi giao diện:

```
using System;
// khai báo giao diện
interface IStorable
{
    // giao diện không khai báo bổ sung truy cập
    // phương thức là public và không thực thi
    void Read();
    void Write(object obj);
    int Status
    {
        get;
        set;
    }
}
```

```
}  
// tạo một lớp thực thi giao diện IStorable  
public class Document : IStorable  
{  
    public Document( string s)  
    {  
        Console.WriteLine("Creating document with: {0}", s);  
    }  
    // thực thi phương thức Read()  
    public void Read()  
    {  
        Console.WriteLine("Implement the Read Method for IStorable");  
    }  
    // thực thi phương thức Write  
    public void Write( object o)  
    {  
        Console.WriteLine("Impleting the Write Method for IStorable");  
    }  
    // thực thi thuộc tính  
    public int Status  
    {  
        get  
        {  
            return status;  
        }  
        set  
        {  
            status = value;  
        }  
    }  
}  
// lưu trữ giá trị thuộc tính
```

```
private int status = 0;
}
public class Tester
{
    static void Main()
    {
        // truy cập phương thức trong đối tượng Document
        Document doc = new Document("Test Document");
        doc.Status = -1;
        doc.Read();
        Console.WriteLine("Document Status: {0}", doc.Status);
        // gán cho một giao diện và sử dụng giao diện
        IStorable isDoc = (IStorable) doc;
        isDoc.Status = 0;
        isDoc.Read();
        Console.WriteLine("IStorable Status: {0}", isDoc.Status);
        Console.ReadLine();
    }
}
```

3. Thực thi nhiều giao diện

Trong ngôn ngữ C# cho phép chúng ta thực thi nhiều hơn một giao diện. Ví dụ, nếu lớp Document có thể được lưu trữ và dữ liệu cũng được nén. Chúng ta có thể chọn thực thi cả hai giao diện IStorable và ICompressible. Như vậy chúng ta phải thay đổi phần khai báo trong danh sách cơ sở để chỉ ra rằng cả hai giao diện đều được thực thi, sử dụng dấu phẩy (,) để phân cách giữa hai giao diện:

```
public class Document : IStorable, ICompressible
```

Khi đó lớp Document phải thực hiện đầy đủ các method được xác nhận trong ICompressible.

4. Mở rộng giao diện

C# cung cấp chức năng cho chúng ta mở rộng một giao diện đã có bằng cách thêm các phương thức và các thành viên hay bổ sung cách làm việc cho các thành viên. Ví dụ, chúng ta có thể mở rộng giao diện ICompressible với một giao diện mới là ILoggedCompressible. Giao diện mới này mở rộng giao diện cũ bằng cách thêm phương thức ghi log các dữ liệu đã lưu:

```
interface ILoggedCompressible : ICompressible
{
    void LogSavedBytes();
}
```

5. Kết hợp giao diện

Một cách tương tự, chúng ta có thể tạo giao diện mới bằng cách kết hợp các giao diện cũ và ta có thể thêm các phương thức hay các thuộc tính cho giao diện mới. Ví dụ, chúng ta quyết định tạo một giao diện IStorableCompressible. Giao diện mới này sẽ kết hợp những phương thức của cả hai giao diện và cũng thêm vào một phương thức mới để lưu trữ kích thước nguyên thủy của các dữ liệu trước khi nén:

```
interface IStorableCompressible : IStorable, ILoggedCompressible
{
    void LogOriginalSize();
}
```

Các bạn đã được làm quen với một khái niệm nữa trong C# đó là giao diện (interface). Cách định nghĩa và thực thi một interface, các thực thi nhiều interface, mở rộng và kết hợp các interface.

Bài 16: Kiểu Struct

Struct (kiểu cấu trúc) là kiểu dữ liệu đơn giản do người dùng định nghĩa, kích thước nhỏ có thể dùng để thay thế cho lớp. Struct cũng tương tự như lớp, cũng chứa các phương thức (methods), những thuộc tính(properties), các trường(fields), các toán tử(operators), các kiểu dữ liệu lồng bên trong và bộ chỉ mục (indexer). Có một số sự khác nhau quan trọng giữa những lớp và cấu trúc. Ví dụ, cấu trúc thì không hỗ trợ kế thừa (Inheritance)và bộ hủy (Destructor)giống như kiểu lớp. Một điều quan trọng nhất là trong khi lớp là kiểu dữ liệu tham chiếu, thì cấu trúc là kiểu dữ liệu giá trị . Do đó cấu trúc thường dùng để thể hiện các đối tượng không đòi hỏi một ngữ nghĩa tham chiếu, hay một lớp nhỏ mà khi đặt vào trong stack thì có lợi hơn là đặt trong bộ nhớ heap. Như vậy,chúng ta chỉ nên sử dụng những cấu trúc chỉ với những kiểu dữ liệu nhỏ, và những hành vi hay thuộc tính của nó giống như các kiểu dữ liệu được xây dựng sẵn.

Trong bài này chúng ta sẽ tìm hiểu cách định nghĩa và làm việc với kiểu cấu trúc và cách sử dụng Constructor để khởi tạo những giá trị của cấu trúc.

Định nghĩa một struct

Cú pháp để khai báo một struct cũng tương tự như cách khai báo một lớp:

```
[thuộc tính] [bổ sung truy cập] struct <tên cấu trúc> [: danh sách giao diện]
{
    [thành viên của cấu trúc]
}
```

Ví dụ sau minh họa cách tạo một cấu trúc.

```
using System;
public struct Location
{
    public Location( int xCoordinate, int yCoordinate)
```

```
{
    xVal = xCoordinate;
    yVal = yCoordinate;
}

public int x
{
    Get
    {
        return xVal;
    }
    set
    {
        xVal = value;
    }
}

public int y
{
    get
    {
        return yVal;
    }
    set
    {
        yVal = value;
    }
}

public override string ToString()
{
    return (String.Format("{0}, {1}", xVal, yVal));
}

// thuộc tính private lưu tọa độ x, y
```



```
private int xVal;
private int yVal;
}
public class Tester
{
    public void myFunc( Location loc)
    {
        loc.x = 50;
        loc.y = 100;
        Console.WriteLine("Loc1 location: {0}", loc);
    }
    static void Main()
    {
        Location loc1 = new Location( 200, 300);
        Console.WriteLine("Loc1 location: {0}", loc1);
        Tester t = new Tester();
        t.myFunc( loc1 );
        Console.WriteLine("Loc1 location: {0}", loc1);
    }
}
```

Những điểm khác nhau giữa Class và Struct

Struct không hỗ trợ thừa kế. Struct được thừa kế từ lớp object nhưng không thể thừa kế từ các lớp khác hay các struct khác. Struct luôn được ngầm định là sealed, nghĩa là không có lớp hay struct nào có thể kế thừa nó. Tuy nhiên struct có thể thực thi nhiều giao diện như class.

Struct không có constructor và destructor mặc định.

Không cho phép khởi tạo các trường thể hiện (instance fields) trong struct vì thế đoạn mã sau sẽ không hợp lệ:

```
private int xVal = 20;
private int yVal = 50;
```

Tạo struct:

Chúng ta tạo một thể hiện của struct bằng cách sử dụng từ khóa new trong câu lệnh gán, như khi chúng ta tạo một đối tượng của lớp. Như trong ví dụ, lớp Tester tạo một thể hiện của Location như sau:

```
Location loc1 = new Location( 200, 300);
```

Ở đây một thể hiện mới tên là loc1 và nó được truyền hai giá trị là 200 và 300.

Struct là một kiểu giá trị và được lưu trữ trên stack. Chúng ta cũng có thể truyền struct vào hàm như các tham số khác.

Bài 17: Namespace trong C#

Ta có thể hiểu Namespace là một gói những thực thể có thuộc tính và hành vi độc lập với bên ngoài. Những ưu điểm của namespace được liệt kê như sau:

- Tránh được sự trùng lặp tên giữa các class.
- Cho phép tổ chức mã nguồn một cách có khoa học và hợp lý.

Khai báo một Namespace

```
namespace NamespaceName
{
    // nơi chứa đựng tất cả các class
}
Trong đó,
```

Namespace: là từ khóa khai báo một Namespace
NamespaceName: là tên của một Namespace

Ví dụ

```
namespace CSharpProgram
{
    class Basic
```

```

{
}
class Advance
{
}
}

```

Bài 18: Luyện tập (Phần ngôn ngữ C#)

Hãy nhớ lại bài học đầu tiên, bạn được học cách mở một ứng dụng console trong Visual Studio.net và gõ những dòng code đầu tiên "Hello C#". Giờ đây bạn đã hoàn toàn làm chủ được ngôn ngữ này rồi. Chúng ta hãy cùng điểm lại những kiến thức trọng tâm của toàn khóa học qua bài ôn tập này.

I. Nhắc lại những đặc điểm của ngôn ngữ C#

C# là một sản phẩm của microsoft, là một ngôn ngữ hướng đối tượng khá thân thiện và mềm dẻo mà bạn có thể sử dụng để xây dựng các ứng dụng desktop hay web.

C# có sẵn các thư viện với các hàm hỗ trợ mạnh cho việc lập trình.

Bạn cần phải nhớ một số nguyên tắc sau:

- C# là ngôn ngữ phân biệt hoa thường
- Quy tắc đặt tên trong C#
- Quy tắc viết chú thích.
- Cách khai báo các thư viện dùng trong chương trình

II. Các kiểu dữ liệu trong c#

C# có 2 loại dữ liệu là dữ liệu kiểu value và dữ liệu kiểu reference. Kiểu value hầu hết là những kiểu có sẵn còn kiểu reference hầu hết là những kiểu do người dùng định nghĩa. Bảng dưới đây tổng hợp các kiểu dữ liệu được xây dựng sẵn:

Kiểu	Khai báo
object	object o = null;
string	string s = "hello";
sbyte	sbyte val = 12;
short	short val = 12;
int	int val = 12;
long	long val1 = 12; long val2 = 34L;
byte	byte val1 = 12;

ushort	ushort val1 = 12;
uint	uint val1 = 12; uint val2 = 34U;
ulong	ulong val1 = 12; ulong val2 = 34U; ulong val3 = 56L; ulong val4 = 78UL;
float	float val = 1.23F;
double	double val1 = 1.23; double val2 = 4.56D;
bool	bool val1 = true; bool val2 = false;
char	char val = 'h';
decimal	decimal val = 1.23M;

III. Các loại lệnh của C#

Bảng sau sẽ tổng hợp lại toàn bộ các loại lệnh của C#

Câu lệnh	Ví dụ minh họa
Câu lệnh đơn và khối lệnh	<pre>static void Main() { F(); G(); { H(); I(); } }</pre>
Khai báo nhãn và lệnh goto	<pre>static void Main(string[] args) { if (args.Length == 0) goto done; Console.WriteLine(args.Length); done: Console.WriteLine("Done"); }</pre>
Khai báo các hằng khu vực (Local constan)	<pre>static void Main() { const float pi = 3.14f; const int r = 123; Console.WriteLine(pi * r * r); }</pre>
khai báo các biến khu vực (local variable)	<pre>static void Main() { int a; int b = 2, c = 3; a = 1;</pre>

	<pre> Console.WriteLine(a + b + c); } </pre>
Các biểu thức lệnh (Expression Statement)	<pre> static int F(int a, int b) { return a + b; } static void Main() { F(1, 2); // Expression statement } </pre>
Lệnh If	<pre> static void Main(string[] args) { if (args.Length == 0) Console.WriteLine("No args"); else Console.WriteLine("Args"); } </pre>
Lệnh Switch	<pre> static void Main(string[] args) { switch (args.Length) { case 0: Console.WriteLine("No args"); break; case 1: Console.WriteLine("One arg "); break; default: int n = args.Length; Console.WriteLine("{0} args", n); break; } } </pre>
Lệnh While	<pre> static void Main(string[] args) { int i = 0; while (i < args.Length) { Console.WriteLine(args[i]); i++; } } </pre>
Lệnh Do...While	<pre> static void Main() { string s; do { s = Console.ReadLine(); } while (s != "Exit"); } </pre>
Lệnh For	<pre> static void Main(string[] args) { for (int i = 0; i < args.length; i++) Console.WriteLine(args[i]); } </pre>
Lệnh foreach	<pre> static void Main(string[] args) { foreach (string s in args) </pre>

	<pre> Console.WriteLine(s); } </pre>
Lệnh break	<pre> static void Main(string[] args) { int i = 0; while (true) { if (i == args.Length) break; Console.WriteLine(args[i++]); } } </pre>
Lệnh continue	<pre> static void Main(string[] args) { int i = 0; while (true) { Console.WriteLine(args[i++]); if (i < args.Length) continue; break; } } </pre>
Lệnh return	<pre> static int F(int a, int b) { return a + b; } static void Main() { Console.WriteLine(F(1, 2)); return; } </pre>
Lệnh Throw và try... catch	<pre> static int F(int a, int b) { if (b == 0) throw new Exception("Divide by zero"); return a / b; } static void Main() { try { Console.WriteLine(F(5, 0)); } catch (Exception e) { Console.WriteLine("Error"); } } </pre>

Bài học sau sẽ tổng hợp các kiến thức về hướng đối tượng trong C#.

Bài 19: Luyện tập- phần hướng đối tượng trong C#

1. Lớp (Class) và đối tượng (Object)

Khái niệm về lớp và đối tượng

Lớp là một khái niệm mô tả cho những thực thể có chung tính chất và hành vi có thể nói lớp là một khuôn mẫu cho các đối tượng. Còn đối tượng là những đại diện cho lớp, mọi đối tượng đều có chung tính chất và hành vi mà lớp định nghĩa.

Các thành phần của lớp

Các thành phần của lớp gồm: Fields, properties và Methods và các thành phần này được phân làm 2 loại là static và instance.

Trong đó :

Fields là các phần tử dùng để thể hiện các biến trong lớp.

Properties là phần tử dùng để truy cập đến đặc điểm của một đối tượng hoặc một class. Properties được định nghĩa bằng 2 phần, phần thứ nhất giống như định nghĩa Fields, phần thứ 2 có thêm 2 phần tử get và set.

Methods hay phương thức chính là các "hành vi" được định nghĩa trong class. Nó dùng để thực hiện một công việc nào đó của một đối tượng hay một class. Khi học về method bạn cần đặc biệt lưu ý đến các vấn đề về Methods overloading, Constructor, Destructor và cách truyền tham số cho method.

Lớp Abstract class và Sealed class

Là 2 lớp đặc biệt của C# trong đó:

Abstract class là lớp chứa phương thức Abstract hay phương thức ảo- tức là các phương thức chỉ được khai báo chứ không thực thi hành động nào. Abstract class chỉ được dùng làm lớp cha cho các lớp kế thừa.

Sealed class là lớp không bao giờ được kế thừa. Nếu bạn khai báo một lớp dẫn xuất từ một lớp Sealed class thì chương trình sẽ báo lỗi.

2. Struct.

Khái niệm về Struct

Struct là một kiểu dữ liệu đơn giản do người dùng định nghĩa, có kích thước nhỏ và có thể được dùng thay cho lớp. Struct cũng chứa những thành phần tương tự như lớp.

Cách khai báo và sử dụng

```
[thuộc tính] [bổ sung truy cập] struct <tên cấu trúc> [: danh sách giao diện]
{
[thành viên của cấu trúc]
}
```

Phân biệt Struct với Class

Struct là kiểu dữ liệu giá trị còn class là kiểu tham chiếu. Struct luôn được mặc định là sealed không hỗ trợ inheritance, constructor và destructor. Nhưng struct cũng có thể thực thi nhiều giao diện như class.

3. Inheritance

Khái niệm về Inheritance

Inheritance là việc một class có thể kế thừa (sử dụng lại) các thuộc tính và các phương thức được định nghĩa từ một class khác. Khi đó class kế thừa được gọi là lớp dẫn xuất hay lớp con còn lớp được kế thừa là lớp cơ sở hay lớp cha.

Khai báo và sử dụng Inheritance.

Khai báo lớp B kế thừa lớp A:

```
class A {}
class B: A {}
```

4. Overriding method và Polymorphism

Phân biệt giữa Overriding method và Polymorphism

Overriding method là một hàm cùng tên cùng kiểu được khai báo trong lớp con và sẽ override hàm trong lớp cha. Polymorphism không chỉ override hàm trong lớp cha mà nó còn override thông minh. Sự khác biệt quan trọng giữa Overriding method và Polymorphism là trong Polymorphism việc quyết định gọi hàm được thực hiện khi chương trình chạy.

5. Interface

Giao diện là ràng buộc, giao ước đảm bảo cho các lớp hay các cấu trúc sẽ thực hiện một điều gì đó. Khi một lớp thực thi một giao diện, thì lớp này báo cho các thành phần client biết rằng lớp này có hỗ trợ các phương thức, thuộc tính, sự kiện và các chỉ mục khai báo trong giao diện.

Khai báo một giao diện:


```
[thuộc tính] [phạm vi truy cập] interface <tên giao diện> [: danh sách cơ sở]
{
<phần thân giao diện>
}
```

6. Namespace

Ta có thể hiểu Namespace là một gói những thực thể có thuộc tính và hành vi độc lập với bên ngoài.

Khai báo một Namespace

```
namespace NamespaceName
{
    // nơi chứa đựng tất cả các class
}
Trong đó,
```

Namespace: là từ khóa khai báo một NameSpace.
NamespaceName: là tên của một Namespace.

Hết

Chúc các bạn học tốt ^^!