Useful tools: <http://topnguyen.net/blog/useful-extensions-for-visual-studio>

# Trung Tâm Lập Trình TPCenter

# Thông tin về khóa học

# Đối tượng

# Nền tảng cần có

# Định nghĩa

C# là một ngôn ngữ lập trình đơn giản, hiện đại, mục đích tổng quát, hướng đối tượng được phát triển bởi Microsoft và được phê chuẩn bởi European Computer Manufacturers Association (ECMA) và International Standards Organization (ISO).

C# được phát triển bởi Anders Hejlsberg và team của ông trong khi phát triển .Net Framework.

C# được thiết kế cho Common Language Infrastructure (CLI), mà gồm Executable Code và Runtime Environment, cho phép chúng ta sử dụng các ngôn ngữ high-level đa dạng trên các nền tảng và cấu trúc máy tính khác nhau.

Dưới đây là các lý do làm C# là ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi:

* Nó là một ngôn ngữ lập trình hiện đại, mục đích tổng quát.
* Nó là hướng đối tượng.
* Nó dễ dàng để học.
* Nó là một ngôn ngữ được cấu trúc.
* Nó tạo các chương trình hiệu quả.
* Nó có thể được biên dịch trên nhiều nền tảng máy tính khác nhau.
* Nó là một phần của .Net Framework.

## Các đặc điểm lập trình mạnh mẽ của C#

Cấu trúc C# khá gần với các ngôn ngữ high-level truyền thống, C và C++, và là một ngôn ngữ lập trình hướng đối tượng. Nó có sự giống nhau mạnh mẽ với Java, nó có nhiều đặc điểm lập trình mạnh mẽ mà làm cho nó trở nên ưa thích với các lập trình viên trên toàn thế giới.

Dưới đây là các đặc điểm quan trọng của C#:

* Điều kiện Boolean
* Tự động dọn rác bởi Garbage-Collector (GC)
* Thư viện chuẩn (Standard Library)
* Assembly
* Property và sự kiện (Event)
* Delegate Quản lý sự kiện
* Dễ dàng để sử dụng Generic
* Indexer
* Biên dịch có điều kiện (Conditional Compilation)
* Đa luồng dễ dàng (Multithreading)
* LINQ và Lambda Expression
* Tích hợp với Windows

# Cài đặt môi trường

Trong chương này, chúng tôi sẽ giới thiệu các công cụ cần thiết để cài đặt môi trường C#. Chúng tôi đã đề cập rằng C# là một phần của .Net Framework và được sử dụng để viết các ứng dụng .Net. Vì thế, trước khi thảo luận về các công cụ có sẵn để chạy một chương trình C#, bạn nên hiểu cách C# liên quan tới .Net Framework.

## Giới thiệu .Net Framework

.Net Framework là một nền tảng mang tính cách mạng giúp bạn viết các kiểu ứng dụng sau:

* Windows application
* Web application
* Dịch vụ Web

Các ứng dụng .Net Framework là các ứng dụng đa nền tảng. Nền tảng đã được thiết kế theo cách như vậy có thể được sử dụng từ bất kỳ ngôn ngữ lập trình nào: C#, C++, Visual Basic, Jscript, COBOL… Tất cả các ngôn ngữ này có thể truy cập tới nền tảng này cũng như giao tiếp với nhau.

.Net Framework gồm một thư viện code khổng lồ được sử dụng bởi các ngôn ngữ client như C#. Dưới đây là một số thành phần của .Net Framework.

* Common Language Runtime (CLR)
* .Net Framework Class Library
* Common Language Specification
* Common Type System
* Metadata và Assembly
* Windows Form
* ASP.Net và ASP.Net AJAX
* ADO.Net
* Windows Workflow Foundation (WF)
* Windows Presentation Foundation
* Windows Communication Foundation (WCF)
* LINQ

## Integrated Development Environment (IDE) cho C#

Microsoft cung cấp các công cụ phát triển sau cho lập trình C#:

* Visual Studio 2015
* Visual Studio 2013

Hai công cụ sau là có sẵn và miễn phí tại Website chính thức của Microsoft. Sử dụng các tool này, bạn có thể viết tất cả các loại chương trình C# từ các ứng dụng dòng lệnh đơn giản tới các ứng dụng phức tạp hơn. Bạn cũng có thể viết các tệp source code của C# bởi sử dụng một Text Editor cơ bản, như Notepad, và biên dịch code đó vào trong Assembly bởi sử dụng Command-line Compiler, mà là một phần của .Net Framework.

Trong loạt bài này, chúng tôi sử dụng Visual C# 2015 Express và bạn có thể tải nó từ: [**Microsoft Visual Studio**](http://www.microsoft.com/visualstudio/eng/downloads). Nó được cài đặt tự động trên máy tính của bạn.

## Viết chương trình C# trên Linux hoặc Mac OS

Mặc dù .Net Framework chạy trên Hệ điều hành Windows, nhưng cũng có một số phiên bản thay thế làm việc trên các Hệ điều hành khác. **Mono** là một phiên bản mã nguồn mở của .Net Framework mà gồm một C# compiler và chạy trên một số Hệ điều hành, gồm Linux và Mac OS. Bạn có thể truy cập: [**Go Mono**](http://www.go-mono.com/mono-downloads/download.html).

Mục đích của Mono không chỉ để chạy trên các ứng dụng .Net trên Microsoft, mà còn đem lại công cụ phát triển cho các lập trình viên Linux. Mono có thể chạy trên nhiều Hệ điều hành khác nhau, như Android, BSD, iOS, Linux, OS X, Windows, Solaris, và UNIX.

# Cú pháp cơ bản

C# là một ngôn ngữ lập trình hướng đối tượng. Trong phương pháp lập trình hướng đối tượng, một chương trình gồm các đối tượng đa dạng mà tương tác với nhau các cách thức của action. Các action mà một đối tượng có thể nhận được gọi là các phương thức. Các đối tượng cùng loại được xem như là có cùng kiểu hoặc, được xem như là trong cùng lớp.

Ví dụ, xét đối tượng Rectangle. Nó có các thuộc tính như length và width. Phụ thuộc vào thiết kế trên. Nó có thể cần những cách để cấp nhận các giá trị của thuộc tính này, tính toán diện tích và hiển thị chi tiết.

Bạn theo dõi ví dụ sau triển khai một lớp Rectangle và cú pháp C# cơ bản của chương trình này. Để minh họa rõ ràng tính hướng đối tượng của C#, mình sẽ tạo hai class trong hai file riêng rẽ nhau.

Lớp *Rectangle*: chứa các thuộc tính, phương thức chính.

using System;

namespace TPCenter

{

class Rectangle

{

// cac bien thanh vien

double length;

double width;

//phuong thuc

public void Acceptdetails()

{

length = 4.5;

width = 3.5;

}

//phuong thuc

public double GetArea()

{

return length \* width;

}

//phuong thuc

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp *ExecuteRectangle*: là lớp chứa phương thức **main()** để tiến hành thao tác trên đối tượng Rectangle của lớp Rectangle. Đến đây, có lẽ bạn chưa hiểu gì, nhưng không sao vì đây chỉ là chương giúp bạn làm quen với C# thôi.

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Chuong trinh minh hoa tinh huong doi tuong ");

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

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//goi cac phuong thuc cua doi tuong nay

r.Acceptdetails();

r.Display();

Console.ReadLine();

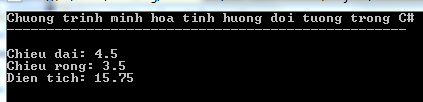
Console.ReadKey();

}

}

}

Nhấn phím **F5** để biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Từ khóa *using*

Lệnh đầu tiên trong bất kỳ chương trình C# nào là:

using System;

Từ khóa *using* được sử dụng để bao namespace trong chương trình. Một chương trình C# có thể bao nhiều lệnh using.

Từ khóa *class*

Từ khóa *class* được sử dụng để khai báo một lớp .

Comments

Comment được sử dụng để khởi tạo code. Compiler bỏ qua các comment. Các comment đa dòng trong các chương trình C# bắt đầu với /\* và kết thúc với \*/ như sau:

/\* dong nay minh hoa comment nhieu dong .

Cu phap co ban C#

Ngon ngu lap trinh C# \*/

Comment đơn dòng được chỉ dẫn bởi ký hiệu '//'. Ví dụ:

// vi du comment don dong

Biến thành viên

Các biến là các thuộc tính hoặc thành viên dữ liệu của một lớp, được sử dụng để lưu giữ dữ liệu. Trong chương trình trước đó, lớp *Rectangle* có hai biến thành viên là *length* và *width*.

Hàm thành viên

Hàm là tập hợp các lệnh mà thực hiện một tác vụ cụ thể. Các hàm thành viên của một lớp được khai báo bên trong lớp đó. Lớp Rectangle chứa 3 hàm thành viên là: *AcceptDetails*, *GetArea* và *Display*.

Thuyết minh một Class

Trong chương trình trên, lớp *ExecuteRectangle* chứa phương thức Main() và khởi tạo lớp *Rectangle*.

Định danh (Identifier)

Một định danh là một tên được sử dụng để nhận diện một lớp, biến, hàm hoặc bất kỳ mục tự định nghĩa (user-defined).

* Một tên phải bắt đầu với một chữ cái mà có thể được theo sau bởi một dãy các chữ cái, chữ số (0-9) hoặc dấu gạch dưới (\_). Ký tự đầu tiên của một định danh không thể là một chữ số.
* Nó phải không chứa bất kỳ khoảng trống hoặc ký tự như ? - + ! @ # % ^ & \* ( ) [ ] { } . ; : " ' / và \. Tuy nhiên, dấu gạch dưới có thể được sử dụng.
* Nó không nên là một từ khóa .

Từ khóa

Từ khóa là các từ dành riêng (*Reserved Keyword*) được định nghĩa trước cho C# compiler. Những từ khóa này không thể được sử dụng làm định danh. Tuy nhiên, nếu bạn muốn sử dụng các từ khóa này để làm định danh, bạn có thể đặt ký tự @ ở trước chúng.

, một số định danh có ý nghĩa đặc biệt trong ngữ cảnh của code, ví dụ như get và set được gọi là các contextual keyword (từ khóa thuộc ngữ cảnh).

Bảng sau liệt kê các từ khóa dành riêng và các từ khóa thuộc ngữ cảnh :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Reserved Keyword** | | | | | | |
| abstract | as | base | bool | break | byte | case |
| catch | char | checked | class | const | continue | decimal |
| default | delegate | do | double | else | enum | event |
| explicit | extern | false | finally | fixed | float | for |
| foreach | goto | if | implicit | in | in (generic modifier) | int |
| interface | internal | is | lock | long | namespace | new |
| null | object | operator | out | out (generic modifier) | override | params |
| private | protected | public | readonly | ref | return | sbyte |
| sealed | short | sizeof | stackalloc | static | string | struct |
| switch | this | throw | true | try | typeof | uint |
| ulong | unchecked | unsafe | ushort | using | virtual | void |
| volatile | while |  |  |  |  |  |
| **Contextual Keyword** | | | | | | |
| add | alias | ascending | descending | dynamic | from | get |
| global | group | into | join | let | orderby | partial (type) |
| partial (method) | remove | select | set |  |  |  |

# Kiểu dữ liệu (Data Types)

Các biến được phân chia thành các kiểu sau:

* Kiểu giá trị (Value type)
* Kiểu tham chiếu (Reference type)
* Kiểu con trỏ (Pointer type)

## Kiểu giá trị

Các biến kiểu giá trị có thể được gán một giá trị một cách trực tiếp. Chúng được kế thừa từ lớp **System.ValueType**.

Kiểu giá trị trực tiếp chứa dữ liệu. Một số ví dụ là **int, char**, và **float**, tương ứng giữ số nguyên, chữ cái, và số thực. Khi bạn khai báo một kiểu **int**, hệ thống cấp phát bộ nhớ để lưu giá trị đó.

Bảng sau liệt kê các kiểu giá trị có sẵn 2015:

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu** | **Biểu diễn** | **Dãy giá trị** | **Giá trị mặc định** |
| bool | Giá trị Boolean | True hoặc False | False |
| byte | Kiểu unsigned integer (8 bit) | 0 tới 255 | 0 |
| char | Kiểu Unicode character (16 bit) | U +0000 tới U +ffff | '\0' |
| decimal | Kiểu thập phân (128 bit) | (-7.9 x 1028 tới 7.9 x 1028) / 100 to 28 | 0.0M |
| double | Kiểu double (64 bit) | (+/-)5.0 x 10-324 tới (+/-)1.7 x 10308 | 0.0D |
| float | Kiểu float (32 bit) | -3.4 x 1038 tới + 3.4 x 1038 | 0.0F |
| int | Kiểu integer (32 bit) | -2,147,483,648 tới 2,147,483,647 | 0 |
| long | Kiểu signed integer (64 bit) | -9,223,372,036,854,775,808 tới 9,223,372,036,854,775,807 | 0L |
| sbyte | Kiểu signed integer (8 bit) | -128 tới 127 | 0 |
| short | Kiểu signed integer (16 bit) | -32,768 tới 32,767 | 0 |
| uint | Kiểu unsigned integer (32 bit) | 0 tới 4,294,967,295 | 0 |
| ulong | Kiểu unsigned integer (64 bit) | 0 tới 18,446,744,073,709,551,615 | 0 |
| ushort | Kiểu unsigned integer (16 bit) | 0 tới 65,535 | 0 |

## Từ khóa sizeof

Để lấy kích cỡ chính xác của một kiểu hoặc một biến trên một nền tảng cụ thể, bạn có thể sử dụng phương thức **sizeof**. Biểu thức *sizeof(type)* hiển thị kích cỡ của đối tượng hoặc kiểu bằng giá trị byte. Ví dụ dưới đây để lấy kích cỡ của kiểu **int** trên bất kỳ máy tính:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Tu khoa sizeof ");

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

Console.WriteLine("Kich co cua kieu du lieu int la: {0}", sizeof(int));

Console.WriteLine("Kich co cua kieu du lieu float la: {0}", sizeof(float));

Console.WriteLine("Kich co cua kieu du lieu double la: {0}", sizeof(double));

Console.WriteLine("Kich co cua kieu du lieu char la: {0}", sizeof(char));

Console.ReadLine();

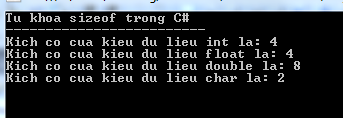
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Kiểu tham chiếu

Kiểu tham chiếu không chứa dữ liệu thực sự được lưu giữ trong một biến, nhưng chúng chứa một tham chiếu tới các biến.

Nói cách khác, chúng tham chiếu tới một vị trí bộ nhớ. Việc sử dụng nhiều biến, thì kiểu tham chiếu có thể tham chiếu tới tới một vị trí bộ nhớ. Nếu dữ liệu trong vị trí bộ nhớ bị thay đổi bởi một trong số các biến, thì biến khác tự động phản ánh sự thay đổi về giá trị này. Ví dụ các kiểu tham chiếu **có sẵn**  là: **object**, **dynamic,** và **string**.

### **Kiểu object**

Kiểu **object** là lớp cơ sở cơ bản cho tất cả kiểu dữ liệu Common Type System (CTS). Object là một alias cho lớp System.Object. Các kiểu object có thể được gán giá trị của bất kỳ kiểu, kiểu giá trị, kiểu tham chiếu, kiểu tự định nghĩa (user-defined) khác. Tuy nhiên, trước khi gán các giá trị, nó cần một sự chuyển đổi kiểu.

Khi một kiểu giá trị được chuyển đổi thành kiểu object, nó được gọi là **boxing** và ngược lại, khi một kiểu object được chuyển đổi thành kiểu giá trị, nó được gọi là **unboxing**.

object obj;

obj = 100; // day la vi du boxing

### **Kiểu Dynamic**

Bạn có thể lưu giữ bất kỳ kiểu giá trị nào trong biến kiểu dữ liệu dynamic. Việc kiểm tra các kiểu biến này diễn ra tại run time.

Cú pháp để khai báo một kiểu dynamic là:

dynamic <tên\_biến> = giá\_trị;

Ví dụ

dynamic d = 20;

Kiểu dynamic là tương tự với kiểu object, ngoại trừ việc kiểm tra cho các biến kiểu object diễn ra tại compile time, trong khi việc kiểm tra các biến kiểu dynamic diễn ra tại run time.

### **Kiểu string**

Kiểu **string**  cho phép bạn gán bất kỳ giá trị chuỗi nào cho một biến. Kiểu string là một *alias* cho lớp **System.String**. Nó kế thừa từ kiểu object. Giá trị cho một kiểu string có thể được gán bởi sử dụng các hằng chuỗi trong hai mẫu: quoted và @quoted.

Ví dụ:

String str = "Hoc C# co ban va nang cao tai TPCenter";

Và một hằng chuỗi @quoted trông như sau:

@"TPCenter Team";

Các kiểu tự định nghĩa (user-defined) là: Class, Interface, hoặc Delegate. Chúng ta sẽ bàn về các kiểu này trong các chương sau.

## Kiểu con trỏ

Các biến kiểu con trỏ lưu giữ địa chỉ bộ nhớ của kiểu khác. Các con trỏ có khả năng như con trỏ trong C hoặc C++.

Cú pháp để khai báo một kiểu con trỏ là:

type\* identifier;

Ví dụ:

char\* cptr;

int\* iptr;

# Chuyển đổi kiểu dữ liệu (Type Conversion)

Chuyển đổi kiểu dữ liệu là biến đổi một kiểu dữ liệu này thành kiểu dữ liệu khác. Nó còn được gọi là ép kiểu, ép kiểu có hai mẫu sau:

* **Chuyển đổi kiểu ngầm định (implicit)** - Việc chuyển đổi này được thực hiện bởi C# theo một phương thức an toàn kiểu (type-safe). Ví dụ: việc chuyển đổi từ các lớp kế thừa thành các lớp cơ sở.
* **Chuyển đổi kiểu tường minh (explicit)** - Việc chuyển đổi này được thực hiện một cách rõ ràng bởi người dùng bằng việc sử dụng các hàm được định nghĩa trước. Các chuyển đổi kiểu tường minh cần một toán tử cast.

Ví dụ sau minh họa một sự chuyển đổi kiểu tường minh :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Chuyen doi kieu du lieu ");

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

double d = 5678.74;

int i;

// cast kieu du lieu double thanh kieu du lieu int.

i = (int)d;

Console.WriteLine("Gia tri cua i = " +i);

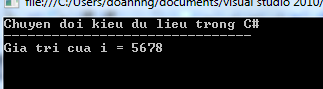
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Phương thức Chuyển đổi kiểu dữ liệu

C# cung cấp các phương thức chuyển đổi kiểu có sẵn được liệt kê trong bảng sau:

|  |  |
| --- | --- |
| **STT** | **Phương thức & Miêu tả** |
| 1 | **ToBoolean**  Chuyển đổi một kiểu thành một giá trị Boolean, nếu có thể |
| 2 | **ToByte**  Chuyển đổi một kiểu thành một byte |
| 3 | **ToChar**  Chuyển đổi một kiểu thành một Unicode character, nếu có thể |
| 4 | **ToDateTime**  Chuyển đổi một kiểu (kiểu integer hoặc string) thành các cấu trúc date-time |
| 5 | **ToDecimal**  Chuyển đổi một kiểu số thực hoặc số nguyên thành một kiểu thập phân |
| 6 | **ToDouble**  Chuyển đổi một kiểu thành một kiểu double |
| 7 | **ToInt16**  Chuyển đổi một kiểu thành một 16-bit integer |
| 8 | **ToInt32**  Chuyển đổi một kiểu thành một 32-bit integer |
| 9 | **ToInt64**  Chuyển đổi một kiểu thành một 64-bit integer |
| 10 | **ToSbyte**  Chuyển đổi một kiểu thành một kiểu signed byte |
| 11 | **ToSingle**  Chuyển đổi một kiểu thành một số small floating point |
| 12 | **ToString**  Chuyển đổi một kiểu thành một string |
| 13 | **ToType**  Chuyển đổi một kiểu thành một kiểu đã xác định |
| 14 | **ToUInt16**  Chuyển đổi một kiểu thành một kiểu unsigned int |
| 15 | **ToUInt32**  Chuyển đổi một kiểu thành một kiểu unsigned long |
| 16 | **ToUInt64**  Chuyển đổi một kiểu thành một unsigned big integer |

Ví dụ sau minh họa cách chuyển đổi các kiểu dữ liệu đa dạng thành kiểu dữ liệu string :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Chuyen doi kieu du lieu ");

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

int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;

//su dung phuong thuc ToString()

Console.WriteLine(i.ToString());

Console.WriteLine(f.ToString());

Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

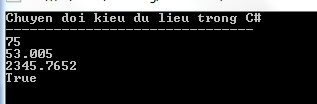
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Biến (Variables)

Một biến là không gì khác ngoài một tên được cung cấp cho khu vực lưu giữ mà chương trình có thể thao tác. Mỗi biến có một kiểu cụ thể, mà quyết định kích cỡ và cách bố trí bộ nhớ của biến đó, cách dãy giá trị có thể được lưu giữ trong bộ nhớ đó, và cách tập hợp các hoạt động có thể được áp dụng tới biến đó.

Các kiểu giá trị cơ bản có thể được phân chia thành:

|  |  |
| --- | --- |
| **Kiểu dữ liệu** | **Ví dụ** |
| Kiểu số nguyên | sbyte, byte, short, ushort, int, uint, long, ulong, và char |
| Kiểu số thực dấu chấm động | float và double |
| Kiểu thập phân | decimal |
| Kiểu Boolean | true hoặc false |
| Kiểu Nullable | Kiểu dữ liệu Nullable |

C# cũng cho phép định nghĩa các kiểu giá trị khác của biến như kiểu **enum** và các kiểu tham chiếu của biến như **class**, sẽ được bàn luận trong các chương tới.

Định nghĩa biến

Cú pháp để định nghĩa biến là:

<kiểu\_dữ\_liệu> <danh\_sách\_biến>;

Ở đây, **kiểu\_dữ\_liệu** phải là một kiểu dữ liệu hợp lệ , gồm: char, int, double hoặc bất kỳ kiểu dữ liệu tự định nghĩa (user-defined) nào và **danh\_sách\_biến** có thể chứa một hoặc nhiều tên định danh được phân biệt bởi dấu phảy.

Dưới đây là một số định nghĩa biến hợp lệ :

int i, j, k;

char c, ch;

float f, salary;

double d;

Bạn có thể khởi tạo một biến tại thời điểm định nghĩa, như sau:

int i = 100;

Khởi tạo biến

Biến được khởi tạo (được gán một giá trị) với một dấu bằng được theo sau bởi một biểu thức hằng. Form chung cho khởi tạo biến là:

tên\_biến = giá\_trị;

Các biến có thể được khởi tạo trong khai báo của chúng. Phần khởi tạo gồm một dấu bằng được theo sau bởi một biểu thức hằng, như sau:

<kiểu\_dữ\_liệu> <tên\_biến> = giá\_trị;

Một số ví dụ về khởi tạo biến là:

int d = 3, f = 5; /\* khai báo và khởi tạo biến d và f. \*/

byte z = 22; /\* khai báo và khởi tạo biến z. \*/

double pi = 3.14159; /\* khai báo và khởi tạo biến pi \*/

char x = 'x'; /\* khai báo và khởi tạo biến ký tự x. \*/

Việc khởi tạo biến một cách chính xác là một bài thực hành tốt cho bạn, nếu không thì chương trình có thể tạo ra kết quả không mong đợi.

Ví dụ sau sử dụng các kiểu biến đa dạng :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Khai bao va khoi tao bien ");

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

//khai bao bien

short a;

int b;

double c;

/\* khoi tao bien \*/

a = 10;

b = 20;

c = a + b;

Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);

Console.ReadLine();

Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:

Biến trong C#

Nhận giá trị từ người dùng

Lớp **Console** trong **System** namespace cung cấp hàm **ReadLine()** để nhận đầu vào từ người dùng (chẳng hạn nhập từ bàn phím) và lưu nó vào trong một biến.

Ví dụ:

int num;

num = Convert.ToInt32(Console.ReadLine());

Hàm **Convert.ToInt32()** chuyển đổi dữ liệu đã nhập bởi người dùng thành kiểu dữ liệu int, bởi vì **Console.ReadLine()** chấp nhận dữ liệu trong định dạng chuỗi.

Biểu thức lvalue và rvalue

Có hai loại biểu thức :

* **lvalue:** Một biểu thức mà là một lvalue có thể xuất hiện hoặc bên trái hoặc bên phải của phép gán .
* **rvalue:** Một biểu thức mà là một rvalue có thể xuất hiện bên phải nhưng không thể ở bên trái của một phép gán .

Các biến là lvalue và vì thế chúng có thể xuất hiện ở bên trái của một phép gán. Các hằng số là rvalue và vì thế chúng không thể được gán và không thể xuất hiện ở bên trái của một phép gán. Sau đây là một lệnh hợp lệ :

int g = 20;

Lệnh sau là không hợp lệ và sẽ tạo một Compile-time error :

10 = 20;

# Hằng (Constant/Literal)

Constant liên quan tới các giá trị cố định mà chương trình không thể thay đổi trong khi thực thi. Những giá trị cố định này cũng được gọi là literal.

Constant là một kiểu dữ liệu thay thế cho Literal, còn Literal thể hiện chính nó. Trong ví dụ: const PI = 3.14 thì Constant ở đây là PI, còn Literal là 3.14.

Constant có thể là bất kỳ kiểu dữ liệu cơ bản nào, và có thể được phân chia thành giá trị hằng số nguyên, hằng số thực, hằng ký tự, hằng chuỗi và Boolean literal (tạm dịch: hằng logic). Ngoài ra, C# cũng có các kiểu hằng liệt kê.

Ngoài ra, constant được đối xử giống như các biến thông thường, ngoại trừ việc giá trị của chúng là không thể thay đổi sau khi định nghĩa.

## Hằng số nguyên

**Hằng số nguyên** có thể là decimal (cơ số 10), octal (cơ số 8) hay hexadecimal (cơ số 16). Giá trị có tiền tố (prefix) là 0 cho octal, là 0x hay 0X cho hexadecimal và không có gì cho decimal.

Một hằng số nguyên cũng có các hậu tố (suffix) U hay L thể hiện kiểu unsigned hay long. Hậu tố có thể là chữ hoa hoặc chữ thường và có thể trong bất kỳ thứ tự nào.

Ví dụ về các hằng số nguyên:

212 /\* hợp lệ \*/

215u /\* hợp lệ \*/

0xFeeL /\* hợp lệ \*/

078 /\* không hợp lệ: 8 không phải là chữ số trong hệ cơ số 8 \*/

032UU /\* không hợp lệ: không thể lặp lại hậu tố \*/

Dưới đây là các kiểu hằng số nguyên đa dạng:

85 /\* thập phân \*/

0213 /\* hệ cơ số 8 \*/

0x4b /\* hệ cơ số 16 \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

## Hằng số thực

Một **hằng số thực** bao gồm phần nguyên (integer part), dấu chấm thập phân (decimal point), phần lẻ (fraction part) và phần mũ (exponent part). Chúng ta có thể biểu diễn hằng số thực theo dạng decimal hay dạng mũ.

Xét các ví dụ sau:

3.14159 /\* hợp lệ \*/

314159E-5L /\* hợp lệ \*/

510E /\* không hợp lệ: phần exponent chưa hoàn thành \*/

210f /\* không hợp lệ: thiếu phần decimal hoặc exponent \*/

.e55 /\* không hợp lệ: thiếu phần integer hoặc fraction \*/

Khi thể hiện dạng decimal phải bao gồm dấu chấm thập phân, phần mũ hoặc cả hai. Khi thể hiện dạng mũ phải gồm phần nguyên, phần lẻ hoặc cả hai. Dạng mũ đi kèm với kí tự E hoặc e.

## Hằng ký tự

Các **hằng kí tự**  mở đầu và kết thúc bởi dấu nháy đơn. Nếu hằng ký tự bắt đầu với L (ví dụ L'x') thì nó là kiểu wchar\_t. Nếu không thì, nó là hằng ký tự kiểu char, ví dụ như 'x'.

Hằng kí tự có thể là một kí tự (như 'X'), một escape sequence (như '\t') hay một kí tự mở rộng (như '\u02c0′).

Một số kí tự khi được đứng trước bởi dấu \ thì chúng sẽ mang một ý nghĩa đặc biệt như bắt đầu dòng mới '\n' hay tạo một tab '\t'. Chúng được biết như là escape sequence (dãy thoát). Bảng dưới đây thể hiện một số mã escape sequence phổ biến:

|  |  |
| --- | --- |
| **Dãy thoát** | **Ý nghĩa** |
| \\ | Ký tự \ |
| \' | Ký tự ' |
| \" | Ký tự " |
| \? | Ký tự ? |
| \a | Tiếng chuông |
| \b | Backspace |
| \f | Form feed |
| \n | Dòng mới |
| \r | Carriage return |
| \t | tab ngang |
| \v | tab dọc |
| \ooo | Số trong cơ số 8 của 1 đến 3 chữ số |
| \xhh . . . | Số thập lục phân của một hoặc nhiều chữ số |

Sau đây là ví dụ để chỉ một số ký tự dãy thoát:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du ky tu tab va newline (dòng mới) ");

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

Console.WriteLine("Hello\tWorld\n\n");

Console.ReadLine();

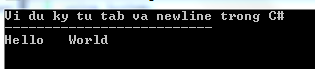
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Hằng chuỗi

**Hằng chuỗi** chứa trong dấu nháy kép, ví dụ "abc". Một chuỗi sẽ chứa các kí tự tương tự hằng kí tự, gồm các ký tự thuần, escape sequence, và các ký tự mở rộng.

Có thể ngắt một dòng dài thành nhiều dòng bởi sử dụng hằng chuỗi và phân biệt chúng bởi sử dụng khoảng trắng (whitespace).

Xét ví dụ một hằng chuỗi thể hiện theo 3 cách khác nhau:

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

@"hello dear"

## Định nghĩa hằng

Hằng được định nghĩa bởi sử dụng từ khóa **const**. Cú pháp để định nghĩa một hằng là:

const <kiểu\_dữ\_liệu> <tên\_hằng> = giá\_trị;

Chương trình sau minh họa cách định nghĩa và sử dụng một hằng trong chương trình C# của bạn:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Hang so ");

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

//khai bao hang so pi

const double pi = 3.14159;

double r;

Console.Write("Nhap ban kinh: ");

r = Convert.ToDouble(Console.ReadLine());

double dien\_tich = pi \* r \* r;

Console.WriteLine("\nBan kinh: {0}, Dien tich: {1}", r, dien\_tich);

Console.ReadLine();

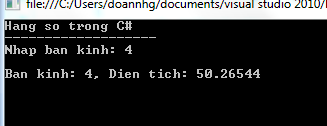
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Toán tử (Operators)

Một toán tử là một biểu tượng, mà nói cho compiler thực hiện các thao tác toán học và logic cụ thể. C# cung cấp nhiều toán tử có sẵn, đó là:

* Toán tử số học
* Toán tử quan hệ
* Toán tử logic
* Toán tử so sánh bit
* Toán tử gán
* Toán tử hỗn hợp

## Toán tử số học

Bảng dưới liệt kê các toán tử số học được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A giữ giá trị 10, biến B giữ 20 thì:

**Ví dụ**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| + | Thêm hai toán hạng | A + B sẽ cho kết quả là 30 |
| - | Trừ giá trị toán hạng hai từ toán hạng đầu | A - B sẽ cho kết quả là -10 |
| \* | Nhân hai toán hạng | A \* B sẽ cho kết quả là 200 |
| / | Chia lấy phần nguyên hai toán hạng | B / A sẽ cho kết quả là 2 |
| % | Chia lấy phần dư | B % A sẽ cho kết quả là 0 |
| ++ | Lượng gia giá trị toán hạng thêm 1 đơn vị | A++ sẽ cho kết quả là 11 |
| -- | Lượng giảm giá trị toán hạng một đơn vị | A-- sẽ cho kết quả là 9 |

## Toán tử quan hệ

Bảng dưới đây liệt kê các toán tử quan hệ được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A giữ giá trị 10, biến B giữ 20 thì:

**Ví dụ**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| == | Kiểm tra nếu 2 toán hạng bằng nhau hay không. Nếu bằng thì điều kiện là true. | (A == B) là không đúng. |
| != | Kiểm tra 2 toán hạng có giá trị khác nhau hay không. Nếu không bằng thì điều kiện là true. | (A != B) là true. |
| > | Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn toán hạng bên phải hay không. Nếu lớn hơn thì điều kiện là true. | (A > B) là không đúng. |
| < | Kiểm tra nếu toán hạng bên trái nhỏ hơn toán hạng bên phải hay không. Nếu nhỏ hơn thì là true. | (A < B) là true. |
| >= | Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn hoặc bằng giá trị của toán hạng bên phải hay không. Nếu đúng là true. | (A >= B) là không đúng. |
| <= | Kiểm tra nếu toán hạng bên trái có giá trị nhỏ hơn hoặc bằng toán hạng bên phải hay không. Nếu đúng là true. | (A <= B) là true. |

## Toán tử logic

Bảng dưới đây chỉ rõ tất cả các toán tử logic được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A có giá trị 1 và biến B có giá trị 0:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_logic_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| && | Được gọi là toán tử logic AND (và). Nếu cả hai toán tử đều có giá trị khác 0 thì điều kiện trở lên true. | (A && B) là false. |
| || | Được gọi là toán tử logic OR (hoặc). Nếu một trong hai toán tử khác 0, thì điều kiện là true. | (A || B) là true. |
| ! | Được gọi là toán tử NOT (phủ định). Sử dụng để đảo ngược lại trạng thái logic của toán hạng đó. Nếu điều kiện toán hạng là true thì phủ định nó sẽ là false. | !(A && B) là true. |

## Toán tử so sánh bit

Toán tử so sánh bit làm việc trên đơn vị bit, tính toán biểu thức so sánh từng bit. Bảng dưới đây về &, |, và ^ như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Giả sử nếu A = 60; và B = 13; thì bây giờ trong định dạng nhị phân chúng sẽ là như sau:

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

Các toán tử so sánh bit được hỗ trợ bởi ngôn ngữ C# được liệt kê trong bảng dưới đây. Giá sử ta có biến A có giá tri 60 và biến B có giá trị 13, ta có:

**Ví dụ**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| & | Toán tử AND (và) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong cả hai toán hạng. | (A & B) sẽ cho kết quả là 12, tức là 0000 1100 |
| | | Toán tử OR (hoặc) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong một hoặc hai toán hạng. | (A | B) sẽ cho kết quả là 61, tức là 0011 1101 |
| ^ | Toán tử XOR nhị phân sao chép bit mà nó chỉ tồn tại trong một toán hạng mà không phải cả hai. | (A ^ B) sẽ cho kết quả là 49, tức là 0011 0001 |
| ~ | Toán tử đảo bit (đảo bit 1 thành bit 0 và ngược lại). | (~A ) sẽ cho kết quả là -61, tức là 1100 0011. |
| << | Toán tử dịch trái. Giá trị toán hạng trái được dịch chuyển sang trái bởi số các bit được xác định bởi toán hạng bên phải. | A << 2 sẽ cho kết quả 240, tức là 1111 0000 (dịch sang trái hai bit) |
| >> | Toán tử dịch phải. Giá trị toán hạng trái được dịch chuyển sang phải bởi số các bit được xác định bởi toán hạng bên phải. | A >> 2 sẽ cho kết quả là 15, tức là 0000 1111 (dịch sang phải hai bit) |

## Toán tử gán

Đây là những toán tử gán được hỗ trợ bởi ngôn ngữ C#:

**Ví dụ**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| = | Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái. | C = A + B sẽ gán giá trị của A + B vào trong C |
| += | Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái. | C += A tương đương với C = C + A |
| -= | Trừ đi giá trị toán hạng phải từ toán hạng trái và gán giá trị này cho toán hạng trái. | C -= A tương đương với C = C - A |
| \*= | Nhân giá trị toán hạng phải với toán hạng trái và gán giá trị này cho toán hạng trái. | C \*= A tương đương với C = C \* A |
| /= | Chia toán hạng trái cho toán hạng phải và gán giá trị này cho toán hạng trái. | C /= A tương đương với C = C / A |
| %= | Lấy phần dư của phép chia toán hạng trái cho toán hạng phải và gán cho toán hạng trái. | C %= A tương đương với C = C % A |
| <<= | Dịch trái toán hạng trái sang số vị trí là giá trị toán hạng phải. | C <<= 2 tương đương với C = C << 2 |
| >>= | Dịch phải toán hạng trái sang số vị trí là giá trị toán hạng phải. | C >>= 2 tương đương với C = C >> 2 |
| &= | Phép AND bit | C &= 2 tương đương với C = C & 2 |
| ^= | Phép OR loại trừ bit | C ^= 2 tương đương với C = C ^ 2 |
| |= | Phép OR bit. | C |= 2 tương đương với C = C | 2 |

## Các toán tử hỗn hợp

Dưới đây là một số toán tử hỗn hợp quan trọng gồm **sizeof, typeof** và **? :** được hỗ trợ bởi ngôn ngữ C#.

**Ví dụ**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| sizeof() | Trả về kích cỡ của một kiểu dữ liệu | sizeof(int), trả về 4 |
| typeof() | Trả về kiểu của một lớp | typeof(StreamReader); |
| & | Trả về địa chỉ của một biến | &a; trả về địa chỉ thực sự của biến |
| \* | Trỏ tới một biến | \*a; tạo con trỏ với tên là a tới một biến |
| ? : | Biểu thức điều kiện (Conditional Expression) | Nếu Condition là true ? Thì giá trị X : Nếu không thì Y |
| is | Xác định đối tượng là một kiểu cụ thể hay không | If( Ford is Car) // Kiểm tra nếu Ford là một đối tượng của lớp Car |
| as | Ép kiểu mà không tạo một exception nếu việc ép kiểu thất bại | Object obj = new StringReader("Hello");  StringReader r = obj as StringReader; |

## Thứ tự ưu tiên toán tử

Thứ tự ưu tiên toán tử xác định cách biểu thức được tính toán. Ví dụ, toán tử nhân có quyền ưu tiên hơn toán tử cộng, và nó được thực hiện trước.

Ví dụ, x = 7 + 3 \* 2; ở đây, x được gán giá trị 13, chứ không phải 20 bởi vì toán tử \* có quyền ưu tiên cao hơn toán tử +, vì thế đầu tiên nó thực hiện phép nhân 3 \* 2 và sau đó thêm với 7.

Bảng dưới đây liệt kê thứ tự ưu tiên của các toán tử. Các toán tử với quyền ưu tiên cao nhất xuất hiện trên cùng của bảng, và các toán tử có quyền ưu tiên thấp nhất thì ở bên dưới cùng của bảng. Trong một biểu thức, các toán tử có quyền ưu tiên cao nhất được tính toán đầu tiên.

[**Ví dụ**](http://vietjack.com/csharp/thu_tu_uu_tien_toan_tu_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Loại** | **Toán tử** | **Thứ tự ưu tiên** |
| Postfix | () [] -> . ++ - - | Trái sang phải |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Phải sang trái |
| Tính nhân | \* / % | Trái sang phải |
| Tính cộng | + - | Trái sang phải |
| Dịch chuyển | << >> | Trái sang phải |
| Quan hệ | < <= > >= | Trái sang phải |
| Cân bằng | == != | Trái sang phải |
| Phép AND bit | & | Trái sang phải |
| Phép XOR bit | ^ | Trái sang phải |
| Phép OR bit | | | Trái sang phải |
| Phép AND logic | && | Trái sang phải |
| Phép OR logic | || | Trái sang phải |
| Điều kiện | ?: | Phải sang trái |
| Gán | = += -= \*= /= %=>>= <<= &= ^= |= | Phải sang trái |
| Dấu phảy | , |  |

# Lệnh If, Else, Switch (Decision Making)

Các cấu trúc điều khiển luồng yêu cầu lập trình viên xác định một hoặc nhiều điều kiện để được đánh giá và kiểm tra bởi chương trình, cùng với các lệnh được thực hiện nếu điều kiện được xác định là đúng, hoặc các lệnh khác được thực hiện nếu điều kiện xác định là sai.

Dưới đây là mẫu chung của một cấu trúc điều khiển luồng hay gặp trong ngôn ngữ lập trình.



C# cung cấp các loại lệnh điều khiển luồng sau. Bạn nhấn vào link để thấy chi tiết.

|  |  |
| --- | --- |
| **Lệnh** | **Miêu tả** |
| **Lệnh if** | Một lệnh **if** bao gồm một biểu thức logic theo sau bởi một hoặc nhiều lệnh khác. |
| **Lệnh if...else** | Một lệnh **if** có thể theo sau bởi một lệnh **else** (tùy ý: có hoặc không), mà có thể được thực hiện khi biểu thức logic có giá trị false. |
| **Lồng các lệnh if** | Bạn có thể sử dụng lệnh **if** hoặc lệnh **else if** bên trong lệnh **if** hoặc **else if** khác |
| **Lệnh switch** | Lệnh **switch** cho phép kiểm tra điều kiện của một biến trước khi thực thi các lệnh |
| **Lồng các lệnh switch** | Bạn có thể sử dụng một lệnh **switch**bên trong một lệnh **switch**khác |

## Toán tử ? :

Chúng ta đã bàn về **toán tử điều kiện ? :** trong chương trước mà có thể được dùng để thay thế cho lệnh **if...else**. Nó có mẫu chung như sau:

Exp1 ? Exp2 : Exp3;

Trong đó Exp1, Exp2 và Exp3 là các biểu thức. Chú ý việc sử dụng và đặt của dấu hai chấm.

Giá trị của biểu thức Exp1 trước dấu ? có giá trị **true**, Exp2 được thực hiện, và giá trị của nó là giá trị của biểu thức. Nếu Exp1 là **false** thì Exp3 được thực hiện và giá trị của nó là giá trị của biểu thức.

# Vòng lặp (Loop)

Có một tình huống mà bạn cần phải thực hiện một đoạn code một vài lần. Nhìn chung, các câu lệnh được thực hiện một cách tuần tự. Câu lệnh đầu tiên của hàm được thực hiện trước, sau đó đến câu thứ 2 và tiếp tục.

Ngôn ngữ lập trình cung cấp cho chúng ta nhiều cấu trúc điều khiển và cho phép bạn thực hiện những phần phức tạp.

Vòng lặp cho phép thực hiện một lệnh và một nhóm lệnh nhiều lần , dưới đây là dạng tổng quát:



C# hỗ trợ những lệnh điều khiển sau đây. Click chuột vào link để xem chi tiết.

|  |  |
| --- | --- |
| **Kiểu vòng lặp** | **Miêu tả** |
| **Vòng lặp while** | Lặp lại một hoặc một nhóm các lệnh trong khi điều kiện đã cho là đúng. Nó kiểm tra điều kiện trước khi thực hiện thân vòng lặp. |
| **Vòng lặp for** | Thực thi một dãy các lệnh nhiều lần và tóm tắt đoạn code mà quản lý biến vòng lặp. |
| **Vòng lặp do...while** | Giống lệnh while, ngoại trừ ở điểm là nó kiểm tra điều kiện ở cuối thân vòng lặp. |
| **Lồng các vòng lặp** | Bạn có thể sử dụng một hoặc nhiều vòng lặp trong các vòng lặp while, for hoặc do..while khác. |

## Các lệnh điều khiển vòng lặp

Các lệnh điều khiển vòng lặp thay đổi sự thực thi lệnh từ dãy thông thường của nó. Khi sự thực thi lệnh rời khỏi một phạm vi, tất cả các đối tượng tự động mà được tạo ra trong phạm vi đó bị hủy.

C# hỗ trợ các lệnh điều khiển vòng lặp sau đây. Click vào các đường link sau để biết thêm chi tiết.

|  |  |
| --- | --- |
| **Lệnh điều khiển** | **Miêu tả** |
| **Lệnh break** | Kết thúc **vòng lặp** hoặc lệnh **switch** và chuyển sang thực thi vòng lặp hoặc lệnh switch ngay sau nó. |
| **Lệnh continue** | Khi gặp lệnh này thì chương trình sẽ bỏ qua các câu lệnh ở dưới nó (trong cùng một câu lệnh lặp) để thực hiện vòng lặp mới. |

## Vòng lặp vô hạn

Một vòng lặp là vòng lặp vô hạn khi một điều kiện không bao giờ false. Vòng lặp **for** thường được sử dụng cho mục đích này. Khi bạn để ba biểu thức điều kiện trong vòng lặp for trống thì bạn sẽ tạo ra một vòng lặp vô hạn.

### **Ví dụ**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vong lap vo han ");

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

for (; ; )

{

Console.WriteLine("Dong nay se duoc in mai mai");

}

}

}

}

Khi biểu thức điều kiện vắng mặt, nó được giả sử là luôn đúng. Bạn có thể có một biểu thức khởi tạo và biểu thức lượng gia, nhưng các lập trình viên C# thường sử dụng for(;;) để biểu thị một vòng lặp vô hạn.

# Tính đóng gói (Encapsulation)

**Encapsulation (Tính đóng gói)** được định nghĩa là "tiến trình đóng gói một hoặc nhiều mục bên trong một gói logic hoặc vật lý". Tính đóng gói, trong phương pháp lập trình hướng đối tượng, ngăn cản việc truy cập tới chi tiết của trình trình triển khai.

Tính trừu tượng và tính đóng gói là hai đặc điểm có liên quan với nhau trong lập trình hướng đối tượng. Tính trừu tượng cho phép tạo các thông tin liên quan có thể nhìn thấy và tính đóng gói cho lập trình viên khả năng *triển khai độ trừu tượng đã được kế thừa*.

Tính đóng gói được triển khai bởi sử dụng **Access Specifier**. Một Access Specifier định nghĩa phạm vi và tính nhìn thấy của một thành viên lớp. C# hỗ trợ các Access Specifier sau:

* Public
* Private
* Protected
* Internal
* Protected internal

Public Access Specifier

Public Access Specifier cho phép một lớp trưng bày các biến thành viên và các hàm thành viên của nó tới các hàm và đối tượng khác. Bất kỳ thành viên public nào có thể được truy cập từ bên ngoài lớp đó.

Ví dụ sau minh họa Public Access Specifier :

Để minh họa rõ ràng tính đóng gói , mình tạo hai lớp có tên lần lượt là: **Rectangle** và **ExecuteRectangle**.

Lớp **Rectangle**: chứa các thuộc tính, phương thức

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

public double length;

public double width;

//cac phuong thuc

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**: chứa phương thức **main()** để thao tác trên đối tượng **Rectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi ");

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

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Display();

Console.ReadLine();

Console.ReadKey();

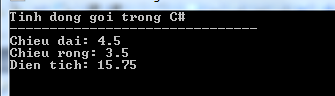
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Trong ví dụ, các biến thành viên length và width được khai báo là **public**, vì thế chúng có thể được truy cập từ hàm Main() bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

Hàm thành viên *Display()* và *GetArea()* cũng có thể truy cập các biến này một cách trực tiếp mà không cần sử dụng bất kỳ instance nào của lớp.

Hàm thành viên *Display()* cũng được khai báo là **public**, vì thế nó cũng có thể được truy cập từ hàm **Main()** bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

Private Access Specifier

Private Access Specifier cho phép một lớp ẩn các biến thành viên và các hàm thành viên của nó với các hàm và đối tượng khác. Chỉ có các hàm trong cùng lớp đó có thể truy cập tới các thành viên private. Ngay cả khi một Instance của một lớp cũng không thể truy cập các thành viên private của nó.

Ví dụ sau minh họa Private Access Specifier :

Lớp **Rectangle**

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

private double length;

private double width;

//cac phuong thuc

public void Acceptdetails()

{

Console.WriteLine("Nhap chieu dai: ");

length = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Nhap chieu rong: ");

width = Convert.ToDouble(Console.ReadLine());

}

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi ");

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

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Acceptdetails();

r.Display();

Console.ReadLine();

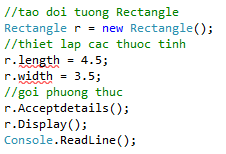
Console.ReadKey();

}

}

}

Trong ví dụ, các biến thành viên length và width được khai báo **private**, vì thế chúng không thể được truy cập từ hàm **Main()**. Trong Visual Studio 2010 sẽ báo hiệu một dấu gạch đỏ bên dưới hai biến này, giống như:



Các hàm thành viên *AcceptDetails()* và *Display()* có thể truy cập các biến này. Khi các hàm thành viên *AcceptDetails()* và *Display()* được khai báo **public**, chúng có thể được truy cập từ hàm **Main()** bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

Protected Access Specifier

Protected Access Specifier cho phép một lớp con truy cập các biến thành viên và các hàm thành viên của lớp cơ sở của nó. Cách này giúp triển khai tính kế thừa. Chúng ta sẽ thảo luận chi tiết về tính kế thừa trong chương sau đó.

Internal Access Specifier

Internal Access Specifier cho phép một lớp trưng bày các biến thành viên và các hàm thành viên của nó tới các hàm và đối tượng khác trong *Assembly* hiện tại. Nói cách khác, bất kỳ thành viên nào với Internal Access Specifier có thể được truy cập từ bất kỳ lớp hoặc phương thức được định nghĩa bên trong ứng dụng mà thành viên đó được định nghĩa.

Ví dụ sau minh họa Internal Access Specifier :

Lớp **Rectangle**

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

internal double length;

internal double width;

//cac phuong thuc

double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi ");

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

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Display();

Console.ReadLine();

Console.ReadKey();

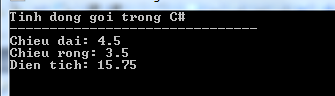
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Trong ví dụ, bạn chú ý rằng hàm thành viên **GetArea()** không được khai báo với bất kỳ Access Specifier nào. Thì theo mặc định, Access Specifier của một thành viên lớp nếu chúng ta không khai báo là **private**.

Protected Internal Access Specifier

Protected Internal Access Specifier cho phép một lớp ẩn các biến thành viên và các hàm thành viên của nó với các hàm và đối tượng khác, ngoại trừ một lớp con bên trong cùng ứng dụng đó. Điều này cũng được sử dụng trong khi triển khai tính kế thừa .

# Phương thức (Method)

Một phương thức là một nhóm lệnh cùng nhau thực hiện một tác vụ. Mỗi chương trình C# có ít nhất một lớp với một phương thức là Main.

Để sử dụng một phương thức , bạn cần:

* Định nghĩa phương thức
* Gọi phương thức

## Định nghĩa phương thức

Khi bạn định nghĩa một phương thức, về cơ bản, bạn khai báo các phần tử của cấu trúc của nó. Cú pháp để định nghĩa một phương thức là như sau:

<Access Specifier> <Kiểu\_trả\_về> <tên\_phương\_thức>(danh\_sách\_tham\_số)

{

phần thân phương thức

}

Dưới đây là chi tiết về các phần tử trong một phương thức:

* **Access Specifier**: Định nghĩa tính nhìn thấy của một biến hoặc một phương thức với lớp khác.
* **Kiểu\_trả\_về**: Một phương thức có thể trả về một giá trị. Kiểu trả về là kiểu dữ liệu của giá trị mà phương thức trả về. Nếu phương thức không trả về bất kỳ giá trị nào, thì kiểu trả về là **void**.
* **tên\_phương\_thức**: Tên phương thức là một định danh duy nhất và nó là phân biệt kiểu chữ. Nó không thể giống bất kỳ định danh nào khác đã được khai báo trong lớp đó.
* **danh\_sách\_tham\_số**: Danh sách tham số được bao quanh trong dấu ngoặc đơn, các tham số này được sử dụng để truyền và nhận dữ liệu từ một phương thức. Danh sách tham số liên quan tới kiểu, thứ tự, và số tham số của một phương thức. Các tham số là tùy ý, tức là một phương thức có thể không chứa tham số nào.
* **phần thân phương thức**: Phần thân phương thức chứa tập hợp các chỉ thị cần thiết để hoàn thành hoạt động đã yêu cầu.

## Ví dụ

Chương trình sau minh họa một hàm *FindMax* nhận hai giá trị integer và trả về số nào lớn hơn trong hai số. Nó có Access Specifier, vì thế nó có thể được truy cập từ bên ngoài lớp bởi sử dụng một Instance (sự thể hiện) của lớp đó.

using System;

namespace TPCenter

{

class TestCsharp

{

public int FindMax(int num1, int num2)

{

/\* khai bao bien cuc bo \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

...

}

}

## Gọi phương thức

Bạn có thể gọi một phương thức bởi sử dụng tên của phương thức đó. Ví dụ sau minh họa cách gọi phương thức :

using System;

namespace TPCenter

{

class TestCsharp

{

public int FindMax(int num1, int num2)

{

/\* khai bao bien cuc bo \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

static void Main(string[] args)

{

Console.WriteLine("Goi phuong thuc ");

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

/\* phan dinh nghia bien cuc bo \*/

int a = 100;

int b = 200;

int ret;

TestCsharp n = new TestCsharp();

//goi phuong thuc FindMax

ret = n.FindMax(a, b);

Console.WriteLine("Gia tri lon nhat la: {0}", ret);

Console.ReadLine();

Console.ReadKey();

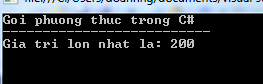
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Bạn cũng có thể gọi phương thức public từ các lớp khác bằng việc sử dụng Instance (sự thể hiện) của lớp đó. Ví dụ, phương thức **FindMax** thuộc lớp *UngDungToan*, bạn có thể gọi nó từ lớp *TestCsharp*.

Tạo hai lớp có tên lần lượt là **UngDungToan** và **TestCsharp** có nội dung như sau:

Lớp **UngDungToan**: chứa phương thức cần gọi

using System;

namespace TPCenter

{

class UngDungToan

{

public int FindMax(int num1, int num2)

{

/\* khai bao bien cuc bo \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()**.

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Goi phuong thuc ");

Console.WriteLine("Goi phuong thuc thong qua instance cua lop");

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

/\* phan dinh nghia bien cuc bo \*/

int a = 100;

int b = 200;

int ret;

//tao doi tuong UngDungToan

UngDungToan n = new UngDungToan();

//goi phuong thuc FindMax

ret = n.FindMax(a, b);

Console.WriteLine("Gia tri lon nhat la: {0}", ret);

Console.ReadLine();

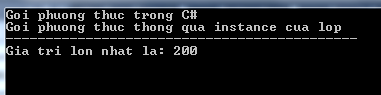
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Gọi phương thức đệ qui

Một phương thức có thể gọi chính nó. Điều này được biết đến là **đệ qui**. Ví dụ sau tính toán giai thừa của số đã cho bởi sử dụng một hàm đệ qui :

using System;

namespace TPCenter

{

class TestCsharp

{

public int TinhGiaiThua(int num)

{

/\* khai bao bien cuc bo \*/

int result;

if (num == 1)

{

return 1;

}

else

{

result = TinhGiaiThua(num - 1) \* num;

return result;

}

}

static void Main(string[] args)

{

Console.WriteLine("Goi phuong thuc ");

Console.WriteLine("Tinh giai thua ");

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

TestCsharp n = new TestCsharp();

//goi phuong thuc

Console.WriteLine("6! = {0}", n.TinhGiaiThua(6));

Console.WriteLine("7! = {0}", n.TinhGiaiThua(7));

Console.WriteLine("8! = {0}", n.TinhGiaiThua(8));

Console.ReadLine();

Console.ReadKey();

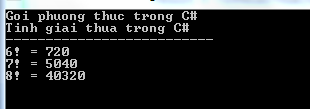
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Truyền tham số cho phương thức

Khi phương thức với các tham số được gọi, bạn cần truyền các tham số cho phương thức đó. Có 3 cách mà tham số có thể được truyền tới một phương thức :

|  |  |
| --- | --- |
| **Kỹ thuật** | **Miêu tả** |
| [**Truyền tham số bởi giá trị**](http://vietjack.com/csharp/truyen_tham_so_boi_gia_tri_trong_csharp.jsp) | Phương thức này sao chép giá trị thực sự của một tham số vào trong tham số chính thức của hàm đó. Trong trường hợp này, các thay đổi được tạo ra với tham số chính thức bên trong hàm này sẽ không ảnh hưởng tới tham số đó |
| [**Truyền tham số bởi tham chiếu**](http://vietjack.com/csharp/truyen_tham_so_boi_tham_chieu_trong_csharp.jsp) | Phương thức này sao chép tham chiếu tới vị trí bộ nhớ của một tham số vào trong tham số chính thức. Nghĩa là các thay đổi được tạo ra tới tham số chính thức ảnh hưởng tới tham số đó |
| [**Truyền tham số bởi output**](http://vietjack.com/csharp/truyen_tham_so_boi_output_trong_csharp.jsp) | Phương thức này giúp ích khi trả về nhiều hơn một giá trị |

# Nullable

C# cung cấp một kiểu dữ liệu đặc biệt, kiểu **nullable**, từ đó bạn có thể gán dãy các giá trị thông thường cũng như các giá trị null .

Ví dụ: Bạn có thể lưu giữ bất kỳ giá trị từ -2,147,483,648 tới 2,147,483,647 hoặc null trong một biến Nullable<Int32>. Tương tự, bạn có thể gán true, false hoặc null trong một biến Nullable<bool>. Cú pháp để khai báo một kiểu **nullable**  là như sau:

< kiểu\_dữ\_liệu > ? <tên\_biến> = null;

Ví dụ sau minh họa cách sử dụng kiểu dữ liệu nullable :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Nullable ");

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

int? num1 = null;

int? num2 = 45;

double? num3 = new double?();

double? num4 = 3.14157;

bool? boolval = new bool?();

// hien thi gia tri

Console.WriteLine("Minh hoa gia tri Nullable kieu int: {0}, {1}, {2}, {3}", num1, num2, num3, num4);

Console.WriteLine("Minh hoa gia tri Nullable kieu boolean: {0}", boolval);

Console.ReadLine();

Console.ReadKey();

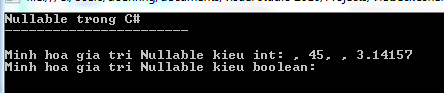
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Toán tử ??

Toán tử Null Coalescing hay toán tử ?? được sử dụng với các kiểu giá trị nullable và các kiểu tham chiếu. Nó được sử dụng để chuyển đổi một toán hạng tới toán hạng với kiểu nullable, tại đây có thể là một sự chuyển đổi ngầm định.

Với toán tử này sẽ làm code bạn gọn, sạch và chuyên nghiệp hơn. Toán tử ?? đã được giới thiệu từ C# phiên bản 2.0. Toán tử ?? có 2 toán hạng. Nếu giá trị của toán hạng đầu tiên là null, thì toán tử trả về giá trị của toán hạng thứ hai; nếu không thì, nó trả về giá trị của toán hạng đầu tiên. Ví dụ sau minh họa toán tử ?? :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Nullable ");

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

double? num1 = null;

double? num2 = 3.14157;

double num3;

num3 = num1 ?? 5.34;

Console.WriteLine("Gia tri cua num3 la: {0}", num3);

num3 = num2 ?? 5.34;

Console.WriteLine("Gia tri cua num3 la: {0}", num3);

Console.ReadLine();

Console.ReadKey();

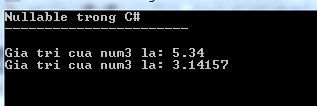
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:

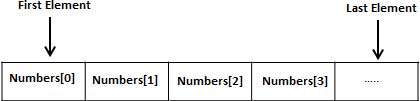


# Mảng (Array)

Một mảng lưu giữ một tập hợp các phần tử có kích cỡ cố định trong cùng kiểu. Một mảng được sử dụng để lưu giữ một tập hợp dữ liệu, nhưng nó thường hữu ích hơn khi nghĩ về một mảng như là một tập hợp các biến cùng kiểu được lưu giữ tại các vị trí bộ nhớ kề nhau.

Thay vì khai báo biến một cách rời rạc, như biến number0, number1,… và number99, bạn có thể khai báo một mảng các giá trị như numbers[0], numbers[1] và … numbers[99] để biểu diễn các giá trị riêng biệt. Một thành viên cụ thể của mảng có thể được truy cập qua index (chỉ số).

Tất cả mảng đều bao gồm các vị trí nhớ liền kề nhau. Địa chỉ thấp nhất tương ứng với thành viên đầu tiền và địa chỉ cao nhất tương ứng với thành viên cuối cùng của mảng.



## Khai báo mảng

Để khai báo một mảng trong ngôn ngữ C#, bạn có thể sử dụng cú pháp:

kiểu\_dữ\_liệu[] tên\_mảng;

Tại đây:

* *kiểu\_dữ\_liệu* được sử dụng để xác định kiểu của phần tử trong mảng.
* *[ ]* xác định rank hay kích cỡ của mảng.
* *tên\_mảng* xác định tên mảng.

Ví dụ:

double[] balance;

## Khởi tạo mảng

Việc khai báo một mảng không khởi tạo mảng trong bộ nhớ. Khi biến mảng được khởi tạo, bạn có thể gán giá trị cho mảng đó.

Mảng là một kiểu tham chiếu, vì thế bạn cần sử dụng từ khóa **new**  để tạo một Instance (sự thể hiện) của mảng đó. Ví dụ:

double[] balance = new double[10];

## Gán giá trị cho một mảng

Bạn có thể gán giá trị cho các phần tử mảng riêng biệt bởi sử dụng chỉ số mảng, như:

double[] balance = new double[10];

balance[0] = 4500.0;

Bạn có thể gán giá trị cho mảng tại thời điểm khai báo mảng, như sau:

double[] balance = { 2340.0, 4523.69, 3421.0};

Bạn cũng có thể tạo và khai báo một mảng, như sau:

int [] marks = new int[5] { 99, 98, 92, 97, 95};

Bạn cũng có thể bỏ qua kích cỡ mảng, như:

int [] marks = new int[] { 99, 98, 92, 97, 95};

Bạn có thể sao chép một biến mảng vào trong biến mảng mục tiêu khác. Trong tình huống này, cả biến mục tiêu và biến nguồn đều trỏ tới cùng vị trí bộ nhớ:

int [] marks = new int[] { 99, 98, 92, 97, 95};

int[] score = marks;

Khi bạn tạo một mảng, C# compiler ngầm định khởi tạo mỗi phần tử mảng thành một giá trị mặc định phụ thuộc vào kiểu mảng. Ví dụ, với một mảng int, thì tất cả phần tử được khởi tạo là 0.

## Truy cập các phần tử mảng

Một phần tử được truy cập bởi chỉ mục mảng. Điều này được thực hiện bởi việc đặt chỉ số của phần tử bên trong dấu ngoặc vuông ở sau tên mảng. Ví dụ:

double salary = balance[9];

Ví dụ sau minh họa khái niệm về khai báo, gán và truy cập mảng đã đề cập ở trên:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Mang ");

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

int [] n = new int[10]; /\* n la mot mang gom 10 so nguyen \*/

int i,j;

/\* khoi tao cac phan tu cua mang n \*/

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

{

n[i] = i + 100;

}

/\* hien thi gia tri cac phan tu cua mang n \*/

for (j = 0; j < 10; j++)

{

Console.WriteLine("Phan tu [{0}] = {1}", j, n[j]);

}

Console.ReadKey();

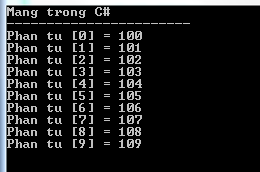
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Sử dụng vòng lặp *foreach*

Trong ví dụ trước, chúng ta đã sử dụng một vòng lặp for để truy cập mỗi phần tử trong mảng. Bạn cũng có thể sử dụng một lệnh **foreach** để duyệt qua một mảng :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Mang ");

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

int[] n = new int[10]; /\* n la mot mang gom 10 so nguyen \*/

/\* khoi tao cac phan tu trong mang n \*/

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

{

n[i] = i + 100;

}

/\* hien thi cac gia tri cua phan tu trong mang n \*/

foreach (int j in n)

{

int i = j - 100;

Console.WriteLine("Phan tu [{0}] = {1}", i, j);

i++;

}

Console.ReadKey();

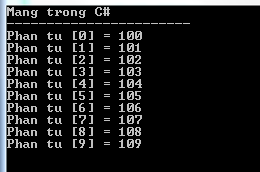
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Chi tiết về mảng

Mảng là một phần rất quan trọng trong ngôn ngữ C#. Dưới đây là những định nghĩa quan trọng liên quan đến mảng mà được trình bày rõ ràng hơn cho các lập trình viên C#:

|  |  |
| --- | --- |
| **Khái niệm** | **Miêu tả** |
| **Mảng đa chiều** | C# hỗ trợ mảng đa chiều. Mẫu đơn giản nhất của mảng đa chiều là mảng hai chiều |
| **Jagged array** | C# hỗ trợ mảng đa chiều, mà là mảng của các mảng |
| **Truyền mảng tới hàm** | Bạn có thể truyền cho hàm một con trỏ tới một mảng bằng việc xác định tên mảng mà không cần chỉ số của mảng |
| **Mảng tham số** | Được sử dụng để truyền một số lượng chưa biết của các tham số tới một hàm |
| **Lớp Array** | Được định nghĩa trong System namespace, nó là lớp cơ sở cho tất cả mảng, và cung cấp các thuộc tính và phương thức để làm việc với mảng |

# Chuỗi (String)

, bạn có thể sử dụng các chuỗi (string) như là mảng các ký tự. Tuy nhiên, phổ biến hơn là để sử dụng từ khóa **string** để khai báo một biến chuỗi. Từ khóa string là một alias cho lớp **System.String** .

Tạo một đối tượng String

Bạn có thể tạo đối tượng String bởi sử dụng một trong các phương thức sau:

* Bằng việc gán một hằng chuỗi cho một biến String
* Sử dụng một constructor của lớp String
* Sử dụng toán tử nối chuỗi (+)
* Bởi việc thu nhận một thuộc tính hoặc gọi một phương thức mà trả về một chuỗi
* Bằng việc gọi một phương thức định dạng để chuyển đổi một giá trị hoặc một đối tượng thành biểu diễn chuỗi của nó.

Ví dụ sau minh họa các phương thức để tạo một chuỗi :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Cac cach tao chuoi ");

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

//su dung phep gan hang chuoi va toan tu noi chuoi

string fname, lname;

fname = "Tran Minh";

lname = "Chinh";

string fullname = fname + " " + lname;

Console.WriteLine("Ho va ten: {0}", fullname);

//su dung constructor cua lop string

char[] letters = { 'H', 'e', 'l', 'l', 'o' };

string greetings = new string(letters);

Console.WriteLine("\nLoi chao bang tieng Anh: {0}", greetings);

//tu cac phuong thuc ma tra ve mot chuoi

string[] sarray = { "TPCenter", "xin", "chao", "cac","ban" };

string message = String.Join(" ", sarray);

Console.WriteLine("\nThong diep: {0}", message);

//dinh dang phuong thuc de chuyen doi mot gia tri

DateTime waiting = new DateTime(2016, 8, 1, 17, 58, 1);

string chat = String.Format("Thong diep duoc gui luc {0:t} ngay {0:D}", waiting);

Console.WriteLine("\nThong diep: {0}", chat);

Console.ReadKey();

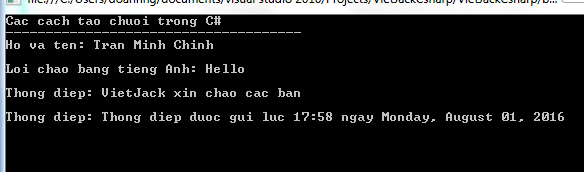
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Các thuộc tính của lớp String

Lớp String có hai thuộc tính:

|  |  |
| --- | --- |
| **STT** | **Thuộc tính** |
| 1 | **Chars**  Lấy đối tượng *Char* tại một vị trí cụ thể trong đối tượng *String* hiện tại |
| 2 | **Length**  Lấy số ký tự của đối tượng String hiện tại |

Phương thức của lớp String

Lớp String có một số phương thức mà hữu ích cho bạn trong khi làm việc với các đối tượng String . Bảng dưới đây liệt kê các phương thức được sử dụng phổ biến nhất:

|  |  |
| --- | --- |
| **STT** | **Phương thức** |
| 1 | **public static int Compare(string strA, string strB)**  So sánh hai đối tượng String cụ thể và trả về một integer mà chỉ vị trí có liên quan của chúng trong thứ tự sắp xếp |
| 2 | **public static int Compare(string strA, string strB, bool ignoreCase )**  So sánh hai đối tượng String cụ thể và trả về một integer mà chỉ vị trí có liên quan của chúng trong thứ tự sắp xếp. Tuy nhiên, nó bỏ qua sự phân biệt kiểu nếu tham số Boolean là true |
| 3 | **public static string Concat(string str0, string str1)**  Nối chuỗi hai đối tượng String |
| 4 | **public static string Concat(string str0, string str1, string str2)**  Nối chuỗi ba đối tượng String |
| 5 | **public static string Concat(string str0, string str1, string str2, string str3)**  Nối chuỗi bốn đối tượng String |
| 6 | **public bool Contains(string value)**  Trả về một giá trị chỉ dẫn có hay không đối tượng String đã cho xuất hiện bên trong chuỗi này |
| 7 | **public static string Copy(string str)**  Tạo một đối tượng String mới với cùng giá trị như chuỗi đã cho |
| 8 | **public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)**  Sao chép một số ký tự cụ thể từ một vị trí đã cho của đối tượng String tới một vị trí đã xác định trong một mảng các ký tự Unicode |
| 9 | **public bool EndsWith(string value)**  Xác định có hay không phần kết thúc của đối tượng String là so khớp với chuỗi đã cho |
| 10 | **public bool Equals(string value)**  Xác định có hay không đối tượng String hiện tại và đối tượng String đã cho là có cùng giá trị |
| 11 | **public static bool Equals(string a, string b)**  Xác định có hay không hai đối tượng String đã cho có cùng giá trị |
| 12 | **public static string Format(string format, Object arg0)**  Thay thế một hoặc nhiều mục định dạng trong một chuỗi đã cho với biểu diễn chuỗi của một đối tượng cụ thể |
| 13 | **public int IndexOf(char value)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của ký tự Unicode đã cho trong chuỗi hiện tại |
| 14 | **public int IndexOf(string value)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của chuỗi đã cho trong Instance (sự thể hiện) này |
| 15 | **public int IndexOf(char value, int startIndex)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của ký tự Unicode đã cho trong chuỗi này, bắt đầu tìm kiếm tại vị trí của ký tự đã cho |
| 16 | **public int IndexOf(string value, int startIndex)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của chuỗi đã cho trong Instance (sự thể hiện) này, bắt đầu tìm kiếm tại vị trí của ký tự đã cho |
| 17 | **public int IndexOfAny(char[] anyOf)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của Instance (sự thể hiện) này của bất kỳ ký tự nào trong một mảng ký tự Unicode đã xác định |
| 18 | **public int IndexOfAny(char[] anyOf, int startIndex)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện đầu tiên của Instance (sự thể hiện) này của bất kỳ ký tự nào trong một mảng ký tự Unicode đã xác định, bắt đầu tìm kiếm tại vị trí của ký tự đã cho |
| 19 | **public string Insert(int startIndex, string value)**  Trả về một chuỗi mới trong đó một chuỗi đã cho được chèn tại một vị trí có chỉ mục đã xác định trong đối tượng String hiện tại |
| 20 | **public static bool IsNullOrEmpty(string value)**  Chỉ rằng có hay không chuỗi đã cho là null hoặc là một chuỗi Empty |
| 21 | **public static string Join(string separator, params string[] value)**  Nối chuỗi tất cả phần tử của một mảng chuỗi, bởi sử dụng Separator (bộ tách) đã cho giữa mỗi phần tử |
| 22 | **public static string Join(string separator, string[] value, int startIndex, int count)**  Nối chuỗi các phần tử đã xác định của một mảng chuỗi, bởi sử dụng Separator (bộ tách) đã cho giữa mỗi phần tử |
| 23 | **public int LastIndexOf(char value)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện cuối cùng của ký tự Unicode đã cho bên trong đối tượng String hiện tại |
| 24 | **public int LastIndexOf(string value)**  Trả về chỉ mục (dựa trên cơ sở 0) cho sự xuất hiện cuối cùng của một chuỗi đã cho bên trong đối tượng String hiện tại |
| 25 | **public string Remove(int startIndex)**  Gỡ bỏ tất cả ký tự trong Instance hiện tại, bắt đầu tại vị trí đã xác định và tiếp tục tới vị trí cuối cùng, và trả về chuỗi đó |
| 26 | **public string Remove(int startIndex, int count)**  Gỡ bỏ số ký tự đã cho trong chuỗi hiện tại bắt đầu tại một vị trí đã xác định và trả về chuỗi đó |
| 27 | **public string Replace(char oldChar, char newChar)**  Thay thế tất cả ký tự Unicode đã cho xuất hiện trong đối tượng String hiện tại với ký tự Unicode đã xác định và trả về chuỗi mới |
| 28 | **public string Replace(string oldValue, string newValue)**  Thay thế tất cả chuỗi đã cho xuất hiện trong đối tượng String hiện tại với đối tượng string đã xác định và trả về chuỗi mới |
| 29 | **public string[] Split(params char[] separator)**  Trả về một mảng chuỗi mà chứa các chuỗi phụ trong đối tượng String hiện tại, được giới hạn bởi các phần tử của một mảng ký tự Unicode đã cho |
| 30 | **public string[] Split(char[] separator, int count)**  Trả về một mảng chuỗi mà chứa các chuỗi phụ trong đối tượng String hiện tại, được giới hạn bởi các phần tử của một mảng ký tự Unicode đã cho. Tham số int xác định số chuỗi phụ tối đa để trả về |
| 31 | **public bool StartsWith(string value)**  Xác định có hay không phần bắt đầu của instance của chuỗi này so khớp với chuỗi đã cho |
| 32 | **public char[] ToCharArray()**  Trả về một mảng ký tự Unicode với tất cả ký tự trong đối tượng String hiện tại |
| 33 | **public char[] ToCharArray(int startIndex, int length)**  Trả về một mảng ký tự Unicode với tất cả ký tự trong đối tượng String hiện tại, bắt đầu từ chỉ mục đã xác định và tới độ dài đã cho |
| 34 | **public string ToLower()**  Trả về một bản sao của chuỗi này đã được biến đổi thành chữ thường |
| 35 | **public string ToUpper()**  Trả về một bản sao của chuỗi này đã được biến đổi thành chữ hoa |
| 36 | **public string Trim()**  Gỡ bỏ tất cả ký tự whitespace từ đối tượng String hiện tại |

Bạn có thể vào thư viện MSDN để lấy danh sách đầy đủ các phương thức và constructor của lớp String.

Ví dụ

Ví dụ sau minh họa một số phương thức trên:

**So sánh chuỗi :**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("So sanh chuoi ");

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

string str1 = "So sanh chuoi ";

string str2 = "So sanh chuoi trong Csharp";

if (String.Compare(str1, str2) == 0)

{

Console.WriteLine(str1 + " va " + str2 + " la giong nhau.");

}

else

{

Console.WriteLine(str1 + " va " + str2 + " la khong giong nhau.");

}

Console.ReadKey();

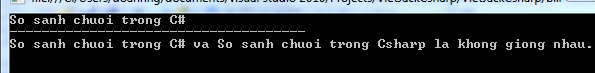
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



**Kiểm tra chuỗi con :**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Kiem tra chuoi con ");

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

string str = "Chuoi con ";

if (str.Contains("trong"))

{

Console.WriteLine("Tim thay chuoi con 'trong'.");

}

Console.ReadKey();

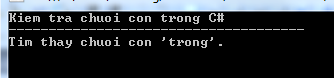
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



**Lấy chuỗi con :**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Lay chuoi con ");

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

string str = "Lay chuoi con ";

Console.WriteLine("Chuoi ban dau: " +str);

string substr = str.Substring(10);

Console.WriteLine("Chuoi con: " +substr);

Console.ReadKey();

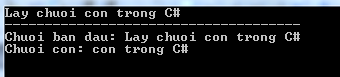
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



**Nối chuỗi :**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Noi chuoi ");

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

string[] starray = new string[]{"Hoc C# co ban va nang cao tai TPCenter.",

"Chuong nay trinh bay ve chuoi .",

"Chung ta dang tim hieu ve noi chuoi .",

"TPCenter chuc cac ban hoc tot.",

"Xin tran trong cam on!"};

string str = String.Join("\n", starray);

Console.WriteLine(str);

Console.ReadKey();

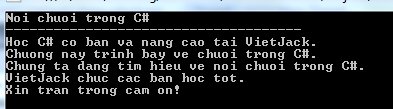
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Cấu trúc (Struct)

Một cấu trúc (structure) là một kiểu dữ liệu. Nó giúp bạn tạo một biến đơn mà giữ dữ liệu liên quan của các kiểu dữ liệu đa dạng. Từ khóa **struct**  được sử dụng để tạo một cấu trúc (structure).

Các cấu trúc được sử dụng để biểu diễn một bản ghi (record). Giả sử bạn muốn theo dõi các cuốn sách trong một thư viện. Bạn có thể muốn theo dõi các thuộc tính sau của mỗi cuốn sách:

* Tên sách
* Tác giả
* Thể loại
* ID (mã sách)

Định nghĩa cấu trúc

Để định nghĩa cấu trúc, bạn phải sử dụng lệnh **struct**. Câu lệnh struct định nghĩa một kiểu dữ liệu mới, với hơn một thành viên trong chương trình của bạn.

Ví dụ dưới đây là cách bạn khai báo cấu trúc Book:

struct Book

{

public string ten\_sach;

public string tac\_gia;

public string the\_loai;

public int ma\_sach;

};

Chương trình sau minh họa cách sử dụng cấu trúc trên :

using System;

//cau truc book

struct Book

{

public string ten\_sach;

public string tac\_gia;

public string the\_loai;

public int ma\_sach;

};

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Struct ");

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

Book Book1; /\* khai bao Book1 thuoc kieu cau truc Book \*/

Book Book2; /\* khai bao Book2 thuoc kieu cau truc Book \*/

/\* thong tin chi tiet ve Book1 \*/

Book1.ten\_sach = "English Grammar in Use";

Book1.tac\_gia = "Raymond Murphy";

Book1.the\_loai = "Tieng Anh";

Book1.ma\_sach = 6495407;

/\* thong tin chi tiet ve Book2 \*/

Book2.ten\_sach = "Toan hoc cao cap";

Book2.tac\_gia = "Tran Van A";

Book2.the\_loai = "Toan hoc";

Book2.ma\_sach = 6495700;

/\* in cac thong tin cua Book1\*/

Console.WriteLine("In thong tin cua cuon sach 1:");

Console.WriteLine("Ten sach: {0}", Book1.ten\_sach);

Console.WriteLine("Tac gia: {0}", Book1.tac\_gia);

Console.WriteLine("The loai: {0}", Book1.the\_loai);

Console.WriteLine("Ma sach: {0}", Book1.ma\_sach);

/\* in cac thong tin cua Book2 \*/

Console.WriteLine("\nIn thong tin cua cuon sach 2:");

Console.WriteLine("Ten sach: {0}", Book2.ten\_sach);

Console.WriteLine("Tac gia: {0}", Book2.tac\_gia);

Console.WriteLine("The loai: {0}", Book2.the\_loai);

Console.WriteLine("Ma sach: {0}", Book2.ma\_sach);

Console.ReadKey();

}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Đặc điểm của cấu trúc

Ở trên, bạn đã sử dụng một cấu trúc Books đơn giản. Các cấu trúc là khá khác với kiểu cấu trúc truyền thống trong C hoặc C++. Cấu trúc có các đặc điểm sau:

* Cấu trúc có thể có các phương thức, các trường, indexer, thuộc tính, phương thức operator, và sự kiện.
* Cấu trúc có thể có các constructor đã được định nghĩa, nhưng không có destructor. Tuy nhiên, bạn không thể định nghĩa một constructor mặc định cho một cấu trúc. Constructor mặc định được định nghĩa tự động và không thể bị thay đổi.
* Không giống các Lớp, cấu trúc không thể kế thừa từ cấu trúc hoặc lớp khác.
* Cấu trúc không thể được sử dụng như là một cơ sở cho cấu trúc hoặc lớp khác.
* Một cấu trúc có thể triển khai một hoặc nhiều Interface.
* Thành viên cấu trúc không thể được xác định ở dạng abstract, virtual, hoặc protected.
* Khi bạn tạo một đối tượng Struct bởi sử dụng toán tử **new**, nó lấy đối tượng đã tạo và constructor thích hợp được gọi. Không giống Lớp, cấu trúc có thể được khởi tạo mà không cần sử dụng toán tử new.
* Nếu toán tử new không được sử dụng, thì các trường chưa được gán và đối tượng không thể được sử dụng tới khi tất cả trường đó được khởi tạo.

Phân biệt Class và Structure

Lớp và Cấu trúc có một số điểm khác nhau cơ bản sau:

* Các Lớp là các kiểu tham chiếu, còn cấu trúc là các kiểu giá trị.
* Cấu trúc không hỗ trợ tính kế thừa.
* Cấu trúc không có constructor mặc định.

Từ các điểm trên, chúng ta viết lại ví dụ trên:

using System;

struct Book

{

private string ten\_sach;

private string tac\_gia;

private string the\_loai;

private int ma\_sach;

public void nhapGiaTri(string t, string a, string s, int id)

{

ten\_sach = t;

tac\_gia = a;

the\_loai = s;

ma\_sach = id;

}

public void display()

{

Console.WriteLine("Tieu de: {0}", ten\_sach);

Console.WriteLine("Tac gia: {0}", tac\_gia);

Console.WriteLine("The loai: {0}", the\_loai);

Console.WriteLine("Ma sach: {0}", ma\_sach);

}

};

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Struct ");

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

Book Book1 = new Book(); /\* Khai bao Book1 thuoc kieu cau truc Book \*/

Book Book2 = new Book(); /\* Khai bao Book2 thuoc kieu cau truc Book \*/

/\* thong tin Book1 \*/

Book1.nhapGiaTri("English Grammer in Use",

"Raymond Murphy", "Tieng Anh", 6495407);

/\* thong tin book2 \*/

Book2.nhapGiaTri("Toan hoc cao cap",

"Tran Van A", "Toan hoc", 6495700);

/\* In thong tin Book1 \*/

Console.WriteLine("In thong tin cua cuon sach 1:");

Book1.display();

/\* In thong tin Book2 \*/

Console.WriteLine("\nIn thong tin cua cuon sach 2:");

Book2.display();

Console.ReadKey();

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

# Enum

Một Enumeration (liệt kê) là một tập hợp các hằng số nguyên được đặt tên. Một kiểu enum được khai báo bởi sử dụng từ khóa **enum** .

Các kiểu liệt kê là kiểu dữ liệu giá trị. Nói cách khác, kiểu liệt kê chứa các giá trị của nó và không thể kế thừa hoặc không thể truyền tính kế thừa.

Khai báo biến *enum*

Cú pháp chung để khai báo một Enumeration là:

enum <tên\_enum>

{

danh\_sách\_enum

};

Tại đây,

* *tên\_enum* xác định tên kiểu liệt kê.
* *danh\_sách\_enum* là danh sách các định danh được phân biệt nhau bởi dấu phảy.

Mỗi biểu tượng trong danh sách liệt kê này đại diện cho một giá trị nguyên, biểu tượng sau có giá trị lớn hơn của biểu tượng ở trước. Theo mặc định, giá trị của biểu tượng kiểu liệt kê đầu tiên là 0. Ví dụ:

enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

Ví dụ

Ví dụ sau minh họa cách sử dụng của biến enum :

using System;

namespace TPCenter

{

public class TestCsharp

{

enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

public static void Main(string[] args)

{

Console.WriteLine("Enum ");

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

int dau\_tuan = (int)Days.Mon;

int cuoi\_tuan = (int)Days.Fri;

Console.WriteLine("Thu hai: {0}", dau\_tuan);

Console.WriteLine("Thu sau: {0}", cuoi\_tuan);

Console.ReadKey();

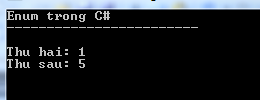
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Lớp (Class)

Khi bạn định nghĩa một lớp (class) , bạn định nghĩa một blueprint cho một kiểu dữ liệu. Điều này không thực sự định nghĩa bất kỳ dữ liệu nào, nhưng nó định nghĩa ý nghĩa của tên lớp đó. Tức là, một đối tượng của lớp đó gồm những cái gì, các hoạt động nào có thể được thực hiện trên đối tượng đó. Các đối tượng là instance (sự thể hiện) của một lớp. Các phương thức và các biến mà cấu tạo nên một lớp được gọi là các thành viên của lớp đó.

Định nghĩa một Class

Một định nghĩa lớp bắt đầu với từ khóa class được theo sau bởi tên lớp; và phần thân lớp được bao quanh bởi các dấu ngoặc ôm. Dưới đây là form chung của một định nghĩa lớp :

<access specifier> class tên\_lớp

{

// các biến thành viên

<access specifier> <kiểu\_dữ\_liệu> biến1;

<access specifier> <kiểu\_dữ\_liệu> biến2;

...

<access specifier> <kiểu\_dữ\_liệu> biếnN;

// các phương thức thành viên

<access specifier> <kiểu\_trả\_về> tên\_phương\_thức1(danh\_sách\_tham\_số)

{

// phần thân phương thức

}

<access specifier> <kiểu\_trả\_về> tên\_phương\_thức2(danh\_sách\_tham\_số)

{

// phần thân phương thức

}

...

<access specifier> <kiểu\_trả\_về> tên\_phương\_thứcN(danh\_sách\_tham\_số)

{

// phần thân phương thức

}

}

Ghi chú:

* Access specifier xác định các qui tắc truy cập cho các thành viên cũng như chính lớp đó. Nếu không được đề cập, thì Access Specifier mặc định cho một kiểu lớp là **internal**. Chế độ truy cập mặc định cho các thành viên là **private**.
* kiểu\_dữ\_liệu xác định kiểu biến, và trả về kiểu dữ liệu mà phương thức trả về.
* Để truy cập các thành viên lớp, bạn sử dụng toán tử dot (.).
* Toán tử dot (.) liên kết với tên của một đối tượng với tên của một thành viên.

Ví dụ sau minh họa các khái niệm về lớp được đề cập ở trên: tạo hai class có tên lần lượt là **Box** và **TestCsharp** trong hai file riêng biệt.

Lớp **Box**: chứa các thuộc tính của một hộp

using System;

namespace TPCenter

{

class Box

{

public double chieu\_dai;

public double chieu\_rong;

public double chieu\_cao;

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên đối tượng **Box**

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Class ");

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

Box Box1 = new Box(); // tao doi tuong Box1

Box Box2 = new Box(); // tao doi tuong Box2

double the\_tich = 0.0; // the tich cua box

// thong tin cua box1

Box1.chieu\_cao = 5.0;

Box1.chieu\_dai = 6.0;

Box1.chieu\_rong = 7.0;

// thong tin cua box2

Box2.chieu\_cao = 10.0;

Box2.chieu\_dai = 12.0;

Box2.chieu\_rong = 13.0;

// Tinh va in the tich cua box1

the\_tich = Box1.chieu\_cao \* Box1.chieu\_dai \* Box1.chieu\_rong;

Console.WriteLine("The tich cua Box1 la: {0}", the\_tich);

// Tinh va in the tich cua box2

the\_tich = Box2.chieu\_cao \* Box2.chieu\_dai \* Box2.chieu\_rong;

Console.WriteLine("The tich cua Box2 la: {0}", the\_tich);

Console.ReadKey();

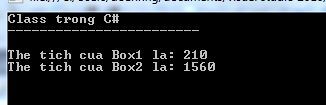
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Hàm thành viên và tính đóng gói

Một hàm thành viên của một lớp là một hàm mà có định nghĩa và nguyên mẫu (prototype) của nó bên trong định nghĩa lớp tương tự như bất kỳ biến nào khác. Nó hoạt động trên bất kỳ đối tượng nào của lớp mà nó là thành viên và có truy cập tới tất cả thành viên của một lớp cho đối tượng đó.

Các biến thành viên là các thuộc tính của một đối tượng (từ bối cảnh thiết kế) và chúng được giữ private để triển khai tính đóng gói. Những biến này chỉ có thể được truy cập bởi sử dụng các hàm thành viên public.

Ví dụ sau minh họa cách sử dụng các khái niệm trên để thiết lập và lấy giá trị của các thành viên khác nhau trong một lớp: tạo hai lớp có tên lần lượt là **Box** và **TestCsharp** trong hai file riêng biệt.

Lớp **Box**: chứa các thuộc tính và phương thức thành viên

using System;

namespace TPCenter

{

class Box

{

private double chieu\_dai;

private double chieu\_rong;

private double chieu\_cao;

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public void setChieuRong(double bre)

{

chieu\_rong = bre;

}

public void setChieuCao(double hei)

{

chieu\_cao = hei;

}

public double tinhTheTich()

{

return chieu\_dai \* chieu\_rong \* chieu\_cao;

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên đối tượng **Box**

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Class ");

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

Box Box1 = new Box(); // tao doi tuong Box1

Box Box2 = new Box(); // tao doi tuong Box2

double the\_tich;

// nhap thong tin cho Box1

Box1.setChieuDai(6.0);

Box1.setChieuRong(7.0);

Box1.setChieuCao(5.0);

// nhap thong tin cho Box2

Box2.setChieuDai(12.0);

Box2.setChieuRong(13.0);

Box2.setChieuCao(10.0);

// tinh va in the tich Box1

the\_tich = Box1.tinhTheTich();

Console.WriteLine("The tich Box1 la: {0}", the\_tich);

// tinh va in the tich Box2

the\_tich = Box2.tinhTheTich();

Console.WriteLine("The tich Box2 la: {0}", the\_tich);

Console.ReadKey();

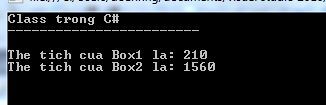
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Constructor

Một **constructor** của một Class, là một hàm thành viên đặc biệt của một lớp, được thực thi bất cứ khi nào chúng ta tạo các đối tượng mới của lớp đó.

Một constructor có tên giống như tên lớp và nó không có bất kỳ kiểu trả về nào. Dưới đây là ví dụ minh họa khái niệm constructor :

using System;

namespace TPCenter

{

class Line

{

private double chieu\_dai;

public Line()

{

Console.WriteLine("Doi tuong dang duoc tao");

}

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public double getChieuDai()

{

return chieu\_dai;

}

static void Main(string[] args)

{

Console.WriteLine("Constructor ");

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

//tao doi tuong Line bang constructor

Line line = new Line();

// thiet lap chieu dai cho duong

line.setChieuDai(6.0);

Console.WriteLine("Chieu dai cua duong la: {0}", line.getChieuDai());

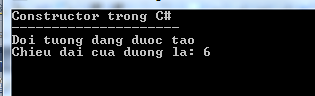
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Một **constructor mặc định**  không có bất kỳ tham số nào, nhưng nếu bạn cần, một constructor có thể có tham số. Những constructor này được gọi là **constructor được tham số hóa**. Kỹ thuật này giúp bạn gán giá trị khởi đầu cho một đối tượng tại thời điểm tạo ra nó, như trong ví dụ sau:

using System;

namespace TPCenter

{

class Line

{

private double chieu\_dai;

public Line(double len) //constructor co tham so

{

Console.WriteLine("Doi tuong dang duoc tao, chieu dai = {0}", len);

chieu\_dai = len;

}

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public double getChieuDai()

{

return chieu\_dai;

}

static void Main(string[] args)

{

Console.WriteLine("Constructor ");

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

//tao doi tuong Line bang constructor

Line line = new Line(10.0);

Console.WriteLine("Chieu dai cua duong la: {0}", line.getChieuDai());

// thiet lap chieu dai cho duong

line.setChieuDai(6.0);

Console.WriteLine("Chieu dai cua duong la: {0}", line.getChieuDai());

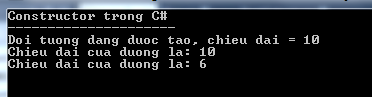
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Destructor

Một **destructor** , là một hàm thành viên đặc biệt của một lớp, được thực thi bất cứ khi nào một đối tượng của lớp đó thoát ra khởi phạm vi. Một **destructor** có tên giống tên lớp với một dẫu ngã (~) ở trước và nó có thể: không trả về một giá trị hoặc không nhận bất kỳ tham số nào.

Destructor có thể rất hữu ích để giải phóng tài nguyên bộ nhớ trước khi thoát khỏi chương trình. Destructor không thể bị kế thừa hoặc nạp chồng.

Ví dụ sau minh họa khái niệm về destructor :

using System;

namespace TPCenter

{

class Line

{

private double chieu\_dai;

public Line() //constructor

{

Console.WriteLine("Doi tuong dang duoc tao.");

}

~Line() // destructor

{

Console.WriteLine("Doi tuong dang bi xoa!!!");

}

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public double getChieuDai()

{

return chieu\_dai;

}

static void Main(string[] args)

{

Console.WriteLine("Constructor ");

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

//tao doi tuong Line bang constructor

Line line = new Line();

// thiet lap chieu dai cho duong

line.setChieuDai(6.0);

Console.WriteLine("Chieu dai cua duong la: {0}", line.getChieuDai());

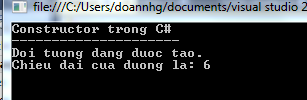
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Thành viên Static của một Class

Chúng ta có thể định nghĩa các thành viên lớp là static bởi sử dụng từ khóa **static** . Khi chúng ta khai báo một thành viên lớp là static, nghĩa là, dù cho có bao nhiêu đối tượng của lớp được tạo, thì chỉ có một bản sao của thành viên static.

Từ khóa **static** ngụ ý rằng chỉ có một instance (sự thể hiện) của thành viên tồn tại cho một lớp đó. Các biến static được sử dụng để định nghĩa các hằng số (constant) bởi vì giá trị của chúng có thể được thu nhận bằng việc gọi lớp đó mà không cần tạo một instance của nó. Các biến static có thể được khởi tạo bên ngoài hàm thành viên hoặc định nghĩa lớp. Bạn cũng có thể khởi tạo các biến static bên trong định nghĩa lớp.

Ví dụ sau minh họa cách sử dụng của các **biến static** : tạo hai class có tên lần lượt là **ThanhVienStatic** và **TestCsharp** như sau:

Lớp **ThanhVienStatic**: chứa thành viên static và các phương thức

using System;

namespace TPCenter

{

class ThanhVienStatic

{

public static int num; //thanh vien static

public void count()

{

num++;

}

public int getNum()

{

return num;

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên đối tượng **ThanhVienStatic** này

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Thanh vien Static ");

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

//tao cac doi tuong ThanhVienStatic

ThanhVienStatic s1 = new ThanhVienStatic();

ThanhVienStatic s2 = new ThanhVienStatic();

//goi phuong thuc count()

s1.count();

s1.count();

s1.count();

s2.count();

s2.count();

s2.count();

Console.WriteLine("Gia tri bien num cho doi tuong s1 la: {0}", s1.getNum());

Console.WriteLine("Gia tri bien num cho doi tuong s2 la: {0}", s2.getNum());

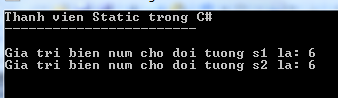
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Bạn cũng có thể khai báo một **hàm thành viên** là **static**. Những hàm này chỉ có thể truy cập các **biến static**. Hàm static có thể tồn tại trước cả khi đối tượng được tạo. Ví dụ sau minh họa cách sử dụng của **hàm static** : ở trên bạn đã tạo hai lớp này rồi, bây giờ chỉ cần thay đổi phần code của hai lớp một chút như sau:

Lớp **ThanhVienStatic**: chứa thành viên static và các phương thức static

using System;

namespace TPCenter

{

class ThanhVienStatic

{

public static int num; // thanh vien static

public void count()

{

num++;

}

//phuong thuc static

public static int getNum()

{

return num;

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên đối tượng **ThanhVienStatic** này

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Phuong thuc Static ");

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

//tao cac doi tuong ThanhVienStatic

ThanhVienStatic s = new ThanhVienStatic();

//goi phuong thuc

s.count();

s.count();

s.count();

Console.WriteLine("Gia tri cua num: {0}", ThanhVienStatic.getNum());

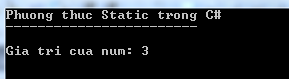
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Tính kế thừa (Inheritance)

Một trong những khái niệm quan trọng nhất trong lập trình hướng đối tượng là **Tính kế thừa (Inheritance)**. Tính kế thừa cho phép chúng ta định nghĩa một lớp trong điều kiện một lớp khác, mà làm cho nó dễ dàng hơn để tạo và duy trì một ứng dụng. Điều này cũng cung cấp một cơ hội để tái sử dụng tính năng code và thời gian thực thi nhanh hơn.

Khi tạo một lớp, thay vì viết toàn bộ các thành viên dữ liệu và các hàm thành viên mới, lập trình viên có thể nên kế thừa các thành viên của một lớp đang tồn tại. Lớp đang tồn tại này được gọi là **Base Class - lớp cơ sở**, và lớp mới được xem như là **Derived Class – lớp thừa kế**.

Ý tưởng của tính kế thừa triển khai mối quan hệ **IS-A** (Là Một). Ví dụ, mammal **IS A** animal, dog **IS-A** mammal, vì thế dog **IS-A** animal, và ....

## Lớp cơ sở (Base Class) và Lớp thừa kế (Derived Class)

Một lớp có thể được kế thừa từ hơn một lớp khác, nghĩa là, nó có thể kế thừa dữ liệu và hàm từ nhiều Lớp hoặc Interface cơ sở.

Cú pháp để tạo lớp kế thừa là:

<acess-specifier> class <base\_class>

{

...

}

class <derived\_class> : <base\_class>

{

...

}

Xét một lớp cơ sở Shape và lớp kế thừa Rectangle sau: tạo 3 lớp có tên lần lượt là **Shape, HinhChuNhat, TestCsharp** trong đó:

Lớp **Shape** là lớp cơ sở

using System;

namespace TPCenter

{

class Shape

{

protected int chieu\_rong;

protected int chieu\_cao;

public void setChieuRong(int w)

{

chieu\_rong = w;

}

public void setChieuCao(int h)

{

chieu\_cao = h;

}

}

}

Lớp **HinhChuNhat** là lớp kế thừa

using System;

namespace TPCenter

{

class HinhChuNhat : Shape

{

public int tinhDienTich()

{

return (chieu\_cao \* chieu\_rong);

}

}

}

Lớp **TestCsharp** chứa phương thức **main()** để thao tác trên đối tượng **HinhChuNhat**

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Tinh ke thua ");

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

//tao doi tuong HinhChuNhat

HinhChuNhat hcn = new HinhChuNhat();

hcn.setChieuRong(5);

hcn.setChieuCao(7);

// in dien tich cua doi tuong.

Console.WriteLine("Dien tich hinh chu nhat: {0}", hcn.tinhDienTich());

Console.ReadKey();

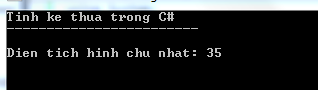
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Khởi tạo Lớp cơ sở (Base Class)

Lớp kế thừa (Derived Class) kế thừa các biến thành viên và các phương thức thành viên từ lớp cơ sở. Vì thế, đối tượng của lớp cha nên được tạo trước khi lớp phụ được tạo. Bạn có thể cung cấp các chỉ thị để khởi tạo lớp phụ trong danh sách khởi tạo thành viên.

Chương trình ví dụ sau minh họa cách khởi tạo Lớp cơ sở (Base Class) : tạo 3 lớp có tên lần lượt là **HinhChuNhat, ChiPhiXayDung, TestCsharp** như sau:

Lớp **HinhChuNhat** là lớp cơ sở

using System;

namespace TPCenter

{

class HinhChuNhat

{

//cac bien thanh vien

protected double chieu\_dai;

protected double chieu\_rong;

// constructor

public HinhChuNhat(double l, double w)

{

chieu\_dai = l;

chieu\_rong = w;

}

//phuong thuc

public double tinhDienTich()

{

return chieu\_dai \* chieu\_rong;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", chieu\_dai);

Console.WriteLine("Chieu rong: {0}", chieu\_rong);

Console.WriteLine("Dien tich: {0}", tinhDienTich());

}

}

}

Lớp **ChiPhiXayDung** kế thừa lớp **HinhChuNhat**

using System;

namespace TPCenter

{

class ChiPhiXayDung : HinhChuNhat

{

private double cost;

public ChiPhiXayDung(double l, double w) : base(l, w)

{ }

public double tinhChiPhi()

{

double chi\_phi;

chi\_phi = tinhDienTich() \* 70;

return chi\_phi;

}

public void hienThiThongTin()

{

base.Display();

Console.WriteLine("Chi phi: {0}", tinhChiPhi());

}

}

}

Lớp **TestCsharp** chứa phương thức **main()** để thao tác trên đối tượng **ChiPhiXayDung**

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Tinh ke thua ");

Console.WriteLine("Khoi tao lop co so");

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

//tao doi tuong ChiPhiXayDung

ChiPhiXayDung t = new ChiPhiXayDung(4.5, 7.5);

t.hienThiThongTin();

Console.ReadLine();

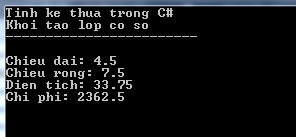
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Đa kế thừa

**C# không hỗ trợ đa kế thừa**. Tuy nhiên, bạn có thể sử dụng Interface để triển khai đa kế thừa. Ví dụ sau minh họa cách sử dụng Interface để triển khai đa kế thừa : chúng ta tạo 2 lớp có tên lần lượt là **Shape, HinhChuNhat, TestCsharp** và một interface có tên là **ChiPhiSon** như sau:

Lớp **Shape** là lớp cơ sở

using System;

namespace TPCenter

{

class Shape

{

protected int chieu\_rong;

protected int chieu\_cao;

public void setChieuRong(int w)

{

chieu\_rong = w;

}

public void setChieuCao(int h)

{

chieu\_cao = h;

}

}

}

interface **ChiPhiSon**

using System;

namespace TPCenter

{

public interface ChiPhiSon

{

int tinhChiPhi(int dien\_tich);

}

}

Lớp **HinhChuNhat** là lớp kế thừa lớp Shape và interface ChiPhiSon

using System;

namespace TPCenter

{

class HinhChuNhat : Shape, ChiPhiSon

{

public int tinhDienTich()

{

return (chieu\_rong \* chieu\_cao);

}

public int tinhChiPhi(int dien\_tich)

{

return dien\_tich \* 70;

}

}

}

Lớp **TestCsharp** chứa phương thức **main()** để thao tác trên đối tượng **HinhChuNhat**

using System;

namespace TPCenter

{

public class TestCsharp

{

public static void Main(string[] args)

{

Console.WriteLine("Tinh ke thua ");

Console.WriteLine("Vi du minh hoa Da ke thua");

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

//tao doi tuong HinhChuNhat

HinhChuNhat hcn = new HinhChuNhat();

int dien\_tich;

hcn.setChieuRong(5);

hcn.setChieuCao(7);

dien\_tich = hcn.tinhDienTich();

// in dien tich va chi phi.

Console.WriteLine("Tong dien tich: {0}", hcn.tinhDienTich());

Console.WriteLine("Tong chi phi son: 0", hcn.tinhChiPhi(dien\_tich));

Console.ReadLine();

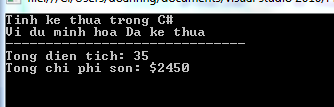
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Tính đa hình (Polymorphism)

Từ **polymorphism** (tính đa hình) nghĩa là có nhiều hình thái. Trong lập trình hướng đối tượng, tính đa hình thường được diễn đạt như là "một Interface, nhiều hàm".

Tính đa hình có thể là **static** hoặc **dynamic**. Trong đó, kiểu **đa hình static** có thể được gọi là đa hình tĩnh và kiểu **đa hình dynamic** có thể được gọi là đa hình động.

Trong đa hình tĩnh, phần phản hồi tới một hàm được xác định tại compile time. Trong khi đó với đa hình động, nó được quyết định tại runtime.

Đa hình static

Kỹ thuật liên kết một hàm với một đối tượng trong thời gian biên dịch được gọi là Early Binding. Nó cũng được gọi là Static Binding. C# cung cấp hai kỹ thuật để triển khai đa hình tĩnh. Chúng là:

* Nạp chồng hàm (Function overloading)
* Nạp chồng toán tử (Operator overloading)

Chúng ta sẽ bàn luận về nạp chồng toán tử trong chương sau.

Nạp chồng hàm

Bạn có thể có nhiều định nghĩa cho cùng tên hàm trong cùng một phạm vi. Các định nghĩa này của hàm phải khác nhau: như kiểu và/hoặc số tham số trong danh sách tham số. , bạn không thể nạp chồng các khai báo hàm mà chỉ khác nhau ở kiểu trả về.

Ví dụ sau minh họa cách sử dụng hàm **print()** để in các kiểu dữ liệu khác nhau :

using System;

namespace TPCenter

{

public class TestCsharp

{

void print(int i)

{

Console.WriteLine("In so nguyen: {0}", i);

}

void print(double f)

{

Console.WriteLine("In so thuc: {0}", f);

}

void print(string s)

{

Console.WriteLine("In chuoi: {0}", s);

}

static void Main(string[] args)

{

Console.WriteLine("Tinh da hinh ");

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

//tao doi tuong TestCsharp

TestCsharp p = new TestCsharp();

// goi ham print()

p.print(5);

p.print(500.263);

p.print("Hoc C# co ban va nang cao");

Console.ReadKey();

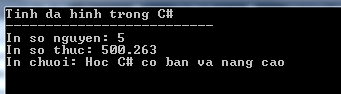
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Đa hình dynamic

C# cho phép bạn tạo các lớp abstract (trừu tượng) mà được sử dụng để cung cấp trình triển khai cục bộ lớp của một Interface. Trình triển khai (Implementation) được hoàn thành khi một lớp kế thừa kế thừa từ nó. Các lớp Abstract chứa các phương thức abstract, mà được triển khai bởi lớp kế thừa. Lớp kế thừa này có tính năng chuyên dụng hơn.

Dưới đây là một số qui tắc về các lớp abstract :

* Bạn không thể tạo một Instance (sự thể hiện) của một lớp abstract.
* Bạn không thể khai báo một phương thức abstract ở bên ngoài một lớp abstract.
* Khi một lớp được khai báo là **sealed**, nó không thể được kế thừa, các lớp abstract không thể được khai báo là sealed.

Ví dụ sau minh họa một lớp abstract : tạo 3 lớp có tên lần lượt là **Shape, HinhChuNhat, TestCsharp** như sau:

Lớp **Shape**: là một lớp abstract

using System;

namespace TPCenter

{

abstract class Shape

{

public abstract int tinhDienTich();

}

}

Lớp **HinhChuNhat**: là một lớp kế thừa lớp Shape

using System;

namespace TPCenter

{

class HinhChuNhat : Shape

{

private int chieu\_dai;

private int chieu\_rong;

public HinhChuNhat(int a = 0, int b = 0)

{

chieu\_dai = a;

chieu\_rong = b;

}

public override int tinhDienTich()

{

Console.WriteLine("Dien tich hinh chu nhat:");

return (chieu\_rong \* chieu\_dai);

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên đối tượng HinhChuNhat

using System;

namespace TPCenter

{

public class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Tinh da hinh ");

Console.WriteLine("Vi du minh hoa Da hinh dong");

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

HinhChuNhat r = new HinhChuNhat(10, 7);

double a = r.tinhDienTich();

Console.WriteLine("Dien tich: {0}", a);

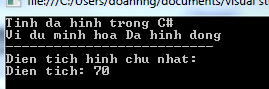
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Khi bạn có một hàm được định nghĩa trong một lớp mà bạn muốn được triển khai một lớp được kế thừa, bạn sử dụng hàm **virtual** . Các hàm virtual có thể được triển khai một cách khác nhau trong lớp được kế thừa khác nhau và việc gọi những hàm này sẽ được quyết định tại runtime.

Đa hình động được triển khai bởi các lớp **abstract** và các hàm **virtual**.

Ví dụ sau minh họa điều này: tạo 5 lớp có tên lần lượt là như sau:

Lớp **Shape**: lớp cơ sở

using System;

namespace TPCenter

{

class Shape

{

protected int chieu\_rong, chieu\_cao;

public Shape(int a = 0, int b = 0)

{

chieu\_rong = a;

chieu\_cao = b;

}

public virtual int tinhDienTich()

{

Console.WriteLine("Dien tich cua class cha: ");

return 0;

}

}

}

Lớp **HinhChuNhat**: là lớp kế thừa lớp Shape

using System;

namespace TPCenter

{

class HinhChuNhat : Shape

{

public HinhChuNhat( int a=0, int b=0): base(a, b)

{

}

public override int tinhDienTich()

{

Console.WriteLine("Dien tich cua class HinhChuNhat: ");

return (chieu\_rong \* chieu\_cao);

}

}

}

Lớp **TamGiac**: là lớp kế thừa lớp Shape

using System;

namespace TPCenter

{

class TamGiac : Shape

{

public TamGiac(int a = 0, int b = 0) : base(a, b)

{

}

public override int tinhDienTich()

{

Console.WriteLine("Dien tich cua class TamGiac: ");

return (chieu\_cao \* chieu\_rong / 2);

}

}

}

Lớp **HienThiDuLieu**: in các dữ liệu

using System;

namespace TPCenter

{

class HienThiDuLieu

{

public void hienThiDienTich(Shape sh)

{

int a;

a = sh.tinhDienTich();

Console.WriteLine("Dien tich: {0}", a);

}

}

}

Lớp **TestCsharp**: chứa phương thức **main()** để thao tác trên các đối tượng

using System;

namespace TPCenter

{

public class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Tinh da hinh ");

Console.WriteLine("Vi du minh hoa Da hinh dong");

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

HienThiDuLieu c = new HienThiDuLieu();

HinhChuNhat r = new HinhChuNhat(10, 7);

TamGiac t = new TamGiac(10, 5);

c.hienThiDienTich(r);

c.hienThiDienTich(t);

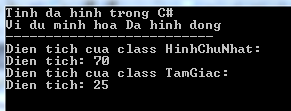
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Nạp chồng toán tử (Operator Overloading)

Operator Overloading là Nạp chồng toán tử. Bạn có thể tái định nghĩa hoặc nạp chồng hầu hết các toán tử có sẵn . Vì thế, một lập trình viên có thể sử dụng các toán tử với các kiểu tự định nghĩa (user-defined). Các toán tử được nạp chồng là các hàm với các tên đặc biệt: từ khóa operator được theo sau bởi biểu tượng cho toán tử đang được định nghĩa. Tương tự như bất kỳ hàm nào khác, một toán tử được nạp chồng có một kiểu trả về và một danh sách tham số.

Ví dụ, bạn xét hàm sau:

public static Box operator+ (Box b, Box c)

{

Box box = new Box();

box.chieu\_dai = b.chieu\_dai + c.chieu\_dai;

box.chieu\_rong = b.chieu\_rong + c.chieu\_rong;

box.chieu\_cao = b.chieu\_cao + c.chieu\_cao;

return box;

}

Hàm trên triển khai toán tử cộng (+) cho một lớp Box tự định nghĩa (user-defined). Nó cộng các thuộc tính của hai đối tượng Box và trả về đối tượng kết quả Box.

## Triển khai Nạp chồng toán tử

Ví dụ dưới đây minh họa cách triển khai nạp chồng toán tử : tạo hai lớp có tên lần lượt là **Box, TestCsharp** như sau:

Lớp **Box**: chứa các thuộc tính và phương thức

using System;

namespace TPCenter

{

class Box

{

private double chieu\_dai;

private double chieu\_rong;

private double chieu\_cao;

public double tinhTheTich()

{

return chieu\_dai \* chieu\_rong \* chieu\_cao;

}

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public void setChieuRong(double bre)

{

chieu\_rong = bre;

}

public void setChieuCao(double hei)

{

chieu\_cao = hei;

}

// nap chong toan tu + de cong hai doi tuong Box.

public static Box operator +(Box b, Box c)

{

Box box = new Box();

box.chieu\_dai = b.chieu\_dai + c.chieu\_dai;

box.chieu\_rong = b.chieu\_rong + c.chieu\_rong;

box.chieu\_cao = b.chieu\_cao + c.chieu\_cao;

return box;

}

}

}

Lớp **TestCsharp**: chứa phương thức main() để thao tác trên đối tượng Box

using System;

namespace TPCenter

{

public class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Nap chong toan tu ");

Console.WriteLine("Vi du minh hoa nap chong toan tu");

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

//tao cac doi tuong Box1, Box2 va Box3

Box Box1 = new Box();

Box Box2 = new Box();

Box Box3 = new Box();

double the\_tich = 0.0;

// nhap thong tin Box1

Box1.setChieuDai(6.0);

Box1.setChieuRong(7.0);

Box1.setChieuCao(5.0);

// nhap thong tin Box2

Box2.setChieuDai(12.0);

Box2.setChieuRong(13.0);

Box2.setChieuCao(10.0);

// tinh va hien thi the tich Box1

the\_tich = Box1.tinhTheTich();

Console.WriteLine("The tich cua Box1 la: {0}", the\_tich);

// tinh va hien thi the tich Box2

the\_tich = Box2.tinhTheTich();

Console.WriteLine("The tich cua Box2 la: {0}", the\_tich);

// con hai doi tuong

Box3 = Box1 + Box2;

// tinh va hien thi the tich Box3

the\_tich = Box3.tinhTheTich();

Console.WriteLine("The tich cua Box3 la: {0}", the\_tich);

Console.ReadKey();

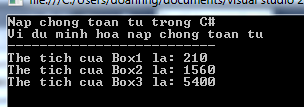
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Toán tử có thể nạp chồng và không thể nạp chồng

Bảng dưới miêu tả các toán tử có thể nạp chồng :

|  |  |
| --- | --- |
| **Toán tử** | **Miêu tả** |
| +, -, !, ~, ++, -- | Những toán tử một ngôi này nhận một toán hạng và **có thể** được nạp chồng |
| +, -, \*, /, % | Những toán tử nhị phân này nhận một toán hạng và **có thể** được nạp chồng |
| =, !=, <, >, <=, >= | Các toán tử so sánh **có thể** được nạp chồng |
| &&, || | Các toán tử logic điều kiện **không thể** được nạp chồng một cách trực tiếp |
| +=, -=, \*=, /=, %= | Các toán tử gán **không thể** được nạp chồng |
| =, ., ?:, ->, new, is, sizeof, typeof | Các toán tử này **không thể** được nạp chồng |

## Ví dụ

Từ các khái niệm trên, chúng ta kế thừa ví dụ trên và nạp chồng thêm một số toán tử : ở trên chúng ta đã tạo hai lớp **Box, TestCsharp**, bây giờ chúng ta sửa lại code của mỗi lớp như sau:

Lớp **Box**: chứa các thuộc tính và phương thức

using System;

namespace TPCenter

{

class Box

{

private double chieu\_dai;

private double chieu\_rong;

private double chieu\_cao;

public double tinhTheTich()

{

return chieu\_dai \* chieu\_rong \* chieu\_cao;

}

public void setChieuDai(double len)

{

chieu\_dai = len;

}

public void setChieuRong(double bre)

{

chieu\_rong = bre;

}

public void setChieuCao(double hei)

{

chieu\_cao = hei;

}

// nap chong toan tu +

public static Box operator +(Box b, Box c)

{

Box box = new Box();

box.chieu\_dai = b.chieu\_dai + c.chieu\_dai;

box.chieu\_rong = b.chieu\_rong + c.chieu\_rong;

box.chieu\_cao = b.chieu\_cao + c.chieu\_cao;

return box;

}

//nap chong toan tu ==

public static bool operator ==(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai == rhs.chieu\_dai && lhs.chieu\_cao == rhs.chieu\_cao && lhs.chieu\_rong == rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong toan tu !=

public static bool operator !=(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai != rhs.chieu\_dai || lhs.chieu\_cao != rhs.chieu\_cao || lhs.chieu\_rong != rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong toan tu <

public static bool operator <(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai < rhs.chieu\_dai && lhs.chieu\_cao < rhs.chieu\_cao && lhs.chieu\_rong < rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong toan tu >

public static bool operator >(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai > rhs.chieu\_dai && lhs.chieu\_cao > rhs.chieu\_cao && lhs.chieu\_rong > rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong toan tu <=

public static bool operator <=(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai <= rhs.chieu\_dai && lhs.chieu\_cao <= rhs.chieu\_cao && lhs.chieu\_rong <= rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong toan tu >=

public static bool operator >=(Box lhs, Box rhs)

{

bool status = false;

if (lhs.chieu\_dai >= rhs.chieu\_dai && lhs.chieu\_cao >= rhs.chieu\_cao && lhs.chieu\_rong >= rhs.chieu\_rong)

{

status = true;

}

return status;

}

//nap chong phuong thuc ToString()

public override string ToString()

{

return String.Format("({0}, {1}, {2})", chieu\_dai, chieu\_rong, chieu\_cao);

}

}

}

Lớp **TestCsharp**: chứa phương thức main() để thao tác trên đối tượng Box

using System;

namespace TPCenter

{

public class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Nap chong toan tu ");

Console.WriteLine("Vi du minh hoa nap chong toan tu");

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

//tao cac doi tuong Box1, Box2, Box2, Box4

Box Box1 = new Box();

Box Box2 = new Box();

Box Box3 = new Box();

Box Box4 = new Box();

double the\_tich = 0.0;

// nhap thong tin Box1

Box1.setChieuDai(6.0);

Box1.setChieuRong(7.0);

Box1.setChieuCao(5.0);

// nhap thong tin Box2

Box2.setChieuDai(12.0);

Box2.setChieuRong(13.0);

Box2.setChieuCao(10.0);

//hien thi thong tin cac Box boi su dung phuong thuc nap chong ToString():

Console.WriteLine("Thong tin Box 1: {0}", Box1.ToString());

Console.WriteLine("Thong tin Box 2: {0}", Box2.ToString());

// tinh the tich Box1

the\_tich = Box1.tinhTheTich();

Console.WriteLine("The tich cua Box1 la: {0}", the\_tich);

// tinh the tich Box2

the\_tich = Box2.tinhTheTich();

Console.WriteLine("The tich cua Box2 la: {0}", the\_tich);

// cong hai doi tuong

Box3 = Box1 + Box2;

Console.WriteLine("Thong tin Box 3: {0}", Box3.ToString());

// tinh the tich cua Box3

the\_tich = Box3.tinhTheTich();

Console.WriteLine("The tich cua Box3 la: {0}", the\_tich);

//so sanh cac Box

if (Box1 > Box2)

Console.WriteLine("Box1 la lon hon Box2");

else

Console.WriteLine("Box1 la khong lon hon Box2");

if (Box1 < Box2)

Console.WriteLine("Box1 la nho hon Box2");

else

Console.WriteLine("Box1 la khong nho hon Box2");

if (Box1 >= Box2)

Console.WriteLine("Box1 la lon hon hoac bang Box2");

else

Console.WriteLine("Box1 la khong lon hon hoac bang Box2");

if (Box1 <= Box2)

Console.WriteLine("Box1 la nho hon hoac bang Box2");

else

Console.WriteLine("Box1 la khong nho hon hoac bang Box2");

if (Box1 != Box2)

Console.WriteLine("Box1 la khong bang Box2");

else

Console.WriteLine("Box1 bang Box2");

Box4 = Box3;

if (Box3 == Box4)

Console.WriteLine("Box3 bamg Box4");

else

Console.WriteLine("Box3 la khong bang Box4");

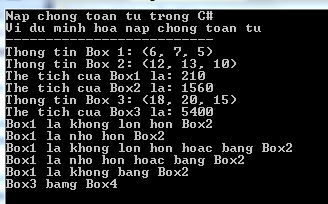
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Interface

Một Interface được định nghĩa như là một giao ước có tính chất cú pháp (syntactical contract) mà tất cả lớp kế thừa Interface đó nên theo. Interface định nghĩa phần "**Là gì**" của giao ước và các lớp kế thừa định nghĩa phần "**Cách nào**" của giao ước đó.

Interface định nghĩa các thuộc tính, phương thức và sự kiện, mà là các thành viên của Interface đó. Các Interface chỉ chứa khai báo của các thành viên này. Việc định nghĩa các thành viên là trách nhiệm của lớp kế thừa. Nó thường giúp ích trong việc cung cấp một Cấu trúc chuẩn mà các lớp kế thừa nên theo.

## Khai báo Interface

Các Interface được khai báo bởi sử dụng từ khóa interface . Nó tương tự như khai báo lớp. Theo mặc định, các lệnh Interface là puclic. Ví dụ sau minh họa một khai báo Interface :

public interface ITransactions

{

// các thành viên của interface

//các phương thức

void hienThiThongTinGiaoDich();

double laySoLuong();

}

## Ví dụ

Sau đây là ví dụ minh họa trình triển khai của Interface trên: tạo 2 lớp có tên lần lượt là **GiaoDichHangHoa, TestCsharp** và một interface có tên là **GiaoDich**

interface **GiaoDich**

using System;

namespace TPCenter

{

public interface GiaoDich

{

// cac thanh vien cua interface

//cac phuong thuc

void hienThiThongTinGiaoDich();

double laySoLuong();

}

}

Lớp **GiaoDichHangHoa** kế thừa interface **GiaoDich**

using System;

namespace TPCenter

{

class GiaoDichHangHoa : GiaoDich

{

private string ma\_hang\_hoa;

private string ngay;

private double so\_luong;

public GiaoDichHangHoa()

{

ma\_hang\_hoa = " ";

ngay = " ";

so\_luong = 0.0;

}

public GiaoDichHangHoa(string c, string d, double a)

{

ma\_hang\_hoa = c;

ngay = d;

so\_luong = a;

}

public double laySoLuong()

{

return so\_luong;

}

public void hienThiThongTinGiaoDich()

{

Console.WriteLine("Ma hang hoa: {0}", ma\_hang\_hoa);

Console.WriteLine("Ngay giao dich: {0}", ngay);

Console.WriteLine("So luong: {0}", laySoLuong());

}

}

}

Lớp **TestCsharp** chứa phương thức main() để thao tác trên đối tượng GiaoDichHangHoa

using System;

namespace TPCenter

{

public class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Interface ");

Console.WriteLine("Vi du minh hoa interface");

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

//tao cac doi tuong GiaoDichHangHoa

GiaoDichHangHoa t1 = new GiaoDichHangHoa("001", "8/10/2012", 78900.00);

GiaoDichHangHoa t2 = new GiaoDichHangHoa("002", "9/10/2012", 451900.00);

t1.hienThiThongTinGiaoDich();

t2.hienThiThongTinGiaoDich();

Console.ReadKey();

}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

# Namespaces

Một **namespace**  được thiết kế để cung cấp một cách để giữ một tập hợp các tên được phân biệt riêng rẽ nhau. Các tên lớp được khai báo trong một namespace không xung đột với cùng tên đó của lớp được khai báo trong namespace khác.

Định nghĩa một Namespace

Một định nghĩa namespace bắt đầu với từ khóa **namespace** được theo sau bởi tên của namespace đó, như sau:

namespace tên\_namespace

{

// phần khai báo code

}

Để gọi phiên bản đã kích hoạt của namespace của hàm hoặc biến, bạn phụ thêm vào sau tên của namespace đó như sau:

tên\_namespace.tên\_phần\_tử;

Ví dụ sau minh họa cách sử dụng của các namespace :

using System;

namespace first\_namespace

{

class namespace\_cl

{

public void func()

{

Console.WriteLine("Ben trong namespace thu nhat!");

}

}

}

namespace second\_namespace

{

class namespace\_cl

{

public void func()

{

Console.WriteLine("Ben trong namespace thu hai!");

}

}

}

class TesCsharp

{

static void Main(string[] args)

{

Console.WriteLine("namespace ");

Console.WriteLine("Vi du minh hoa cach su dung cua namespace");

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

first\_namespace.namespace\_cl fc = new first\_namespace.namespace\_cl();

second\_namespace.namespace\_cl sc = new second\_namespace.namespace\_cl();

fc.func();

sc.func();

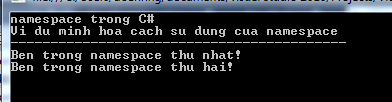
Console.ReadKey();

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Từ khóa *using*

Từ khóa **using** biểu diễn rằng chương trình đang sử dụng các tên trong namespace đã cho. Ví dụ: chúng ta đang sử dụng **System** namespace trong các chương trình của chúng ta. Lớp **Console** được định nghĩa ở đây. Chúng ta viết:

Console.WriteLine ("Hoc C# co ban va nang cao!!!");

Chúng ta có thể viết tên đầy đủ là:

System.Console.WriteLine("Hoc C# co ban va nang cao!!!");

Bạn cũng có thể tránh việc phụ thêm vào các namespace bởi sử dụng directive là **using** namespace. Chỉ thị này nói cho compiler rằng phần code tiếp theo đang sử dụng các tên trong namespace đã xác định.

Giờ viết lại ví dụ trên bởi sử dụng *using* directive :

using System;

using first\_namespace;

using second\_namespace;

namespace first\_namespace

{

class namespace\_cl

{

public void func()

{

Console.WriteLine("Ben trong namespace thu nhat!");

}

}

}

namespace second\_namespace

{

class namespace\_cl

{

public void func()

{

Console.WriteLine("Ben trong namespace thu hai!");

}

}

}

class TesCsharp

{

static void Main(string[] args)

{

Console.WriteLine("namespace ");

Console.WriteLine("Vi du minh hoa cach su dung cua namespace");

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

first\_namespace.namespace\_cl fc = new first\_namespace.namespace\_cl();

second\_namespace.namespace\_cl sc = new second\_namespace.namespace\_cl();

fc.func();

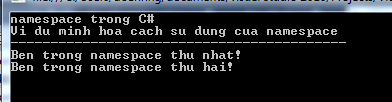
sc.func();

Console.ReadKey();

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Lồng Namespace

, bạn có thể định nghĩa một namespace bên trong các namespace khác, như sau:

namespace tên\_namespace\_1

{

// phần khai báo code

namespace tên\_namespace\_2

{

// phần khai báo code

}

}

Bạn có thể truy cập các thành viên của các namespace được lồng vào nhau này bởi sử dụng toán tử dot (.) , như sau:

using System;

using first\_namespace;

using first\_namespace.second\_namespace;

namespace first\_namespace

{

class abc

{

public void func()

{

Console.WriteLine("Ben trong namespace thu nhat!");

}

}

namespace second\_namespace

{

class efg

{

public void func()

{

Console.WriteLine("Ben trong namespace thu hai!");

}

}

}

}

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("namespace ");

Console.WriteLine("Vi du minh hoa cach su dung cua namespace");

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

abc fc = new abc();

efg sc = new efg();

fc.func();

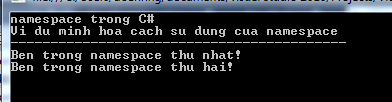
sc.func();

Console.ReadKey();

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Chỉ thị tiền xử lý (Preprocessor Directives)

Các chỉ thị tiền xử lý (Preprocessor Directive) cung cấp chỉ lệnh tới compiler để tiền xử lý thông tin trước khi sự biên dịch thực sự bắt đầu.

Tất cả chỉ thị tiền xử lý bắt đầu với #, và chỉ có các ký tự white-space có thể xuất hiện ở trước một chỉ thị tiền xử lý trong một dòng. Các chỉ thị tiền xử lý không là các lệnh, vì thế chúng không kết thúc với một dấu chấm phảy (;).

Bộ biên dịch của C# không có một bộ tiền xử lý riêng biệt, tuy nhiên, các chỉ thị này được xử lý như khi thực sự có một bộ tiền xử lý riêng vậy. , các chỉ thị tiền xử lý được sử dụng để giúp ích việc biên dịch có điều kiện. Không giống các chỉ thị tiền xử lý trong C và C++, chúng không được sử dụng để tạo các macro. Một chỉ thị tiền xử lý phải chỉ là một chỉ lệnh trên một dòng.

## Các chỉ thị tiền xử lý

Dưới đây là bảng liệt kê các chỉ thị tiền xử lý có sẵn :

|  |  |
| --- | --- |
| **Chỉ thị tiền xử lý** | **Miêu tả** |
| #define | Nó định nghĩa một dãy ký tự, được gọi là các biểu tượng |
| #undef | Nó cho phép bạn không định nghĩa (undefine) một biểu tượng |
| #if | Nó cho phép kiểm tra một biểu tượng hoặc nhiều biểu tượng để thấy nếu chúng ước lượng là true |
| #else | Nó cho phép tạo một chỉ thị có điều kiện phức hợp, cùng với #if |
| #elif | Nó cho phép tạo một chỉ thị có điều kiện phức hợp |
| #endif | Xác định phần cuối của một chỉ thị có điều kiện (conditional directive) |
| #line | Nó cho phép bạn sửa đổi số dòng của compiler và (tùy ý) tên file cho Error và Warning |
| #error | Nó cho phép tạo một error từ một vị trí cụ thể trong code của bạn |
| #warning | Nó cho phép tạo một mức độ cảnh báo từ một vị trí cụ thể trong code của bạn |
| #region | Nó cho phép bạn xác định một khối code mà bạn có thể mở rộng hoặc thu gọn bởi sử dụng đặc điểm của Visual Studio Code Editor |
| #endregion | Nó đánh dấu phần cuối của một khối #region |

## Chỉ thị tiền xử lý #define

Chỉ thị tiền xử lý #define tạo các hằng biểu tượng.

#define cho phép bạn tạo một biểu tượng như vậy, bởi sử dụng biểu tượng dạng biểu thức được truyền tới chỉ thị tiền xử lý #if, biểu thức ước lượng là true. Cú pháp của nó như sau:

#define symbol

Ví dụ sau minh họa điều này:

#define PI

using System;

namespace TPCenter

{

class TestClass

{

static void Main(string[] args)

{

Console.WriteLine("Chi thi tien xu ly ");

Console.WriteLine("Vi du minh hoa chi thi tien xu ly #define");

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

#if (PI)

Console.WriteLine("PI da duoc dinh nghia");

#else

Console.WriteLine("PI chua duoc dinh nghia");

#endif

Console.ReadKey();

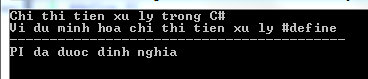
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Chỉ thị có điều kiện (Conditional Directive)

Bạn có thể sử dụng chỉ thị tiền xử lý #if để tạo một chỉ thị có điều kiện (Conditional Directive). Các chỉ thị có điều kiện là hữu ích khi kiểm tra một biểu tượng hoặc các biểu tượng để kiểm tra nếu chúng ước lượng là true. Nếu chúng ước lượng là true, compiler ước lượng tất cả code giữa chỉ thị #if và chỉ thị tiếp theo.

Cú pháp cho chỉ thị có điều kiện là:

#if symbol [operator symbol]...

Tại đây, *symbol* là tên của biểu tượng bạn muốn kiểm tra. Bạn cũng có thể sử dụng true và false hoặc phụ thêm vào sau biểu tượng với toán tử phủ định.

*operator symbol* là toán tử được sử dụng để ước lượng biểu tượng đó. Các toán tử có thể là một trong các:

* == (bằng)
* != (không bằng)
* && (và)
* || (hoặc)

Bạn cũng có thể nhóm các biểu tượng và toán tử bởi các dấu ngoặc đơn. Các chỉ thị có điều kiện được sử dụng để biên dịch code cho debug hoặc khi biên dịch cho một sự định cấu hình cụ thể. Một chỉ thị có điều kiện bắt đầu với một chỉ thị tiền xử lý #if phải được kết thúc một cách rõ ràng bởi một chỉ thị **#endif**.

Ví dụ sau minh họa sự sử dụng các chỉ thị có điều kiện :

#define DEBUG

#define VC\_V10

using System;

namespace TPCenter

{

class TestClass

{

static void Main(string[] args)

{

Console.WriteLine("Chi thi tien xu ly ");

Console.WriteLine("Vi du minh hoa chi thi tien xu ly #define");

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

#if (DEBUG && !VC\_V10)

Console.WriteLine("DEBUG da duoc dinh nghia");

#elif (!DEBUG && VC\_V10)

Console.WriteLine("VC\_V10 da duoc dinh nghia");

#elif (DEBUG && VC\_V10)

Console.WriteLine("DEBUG va VC\_V10 da duoc dinh nghia");

#else

Console.WriteLine("DEBUG va VC\_V10 chua duoc dinh nghia");

#endif

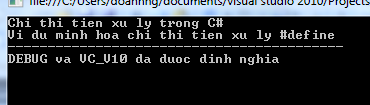
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Regular Expression

Công cụ hỗ trợ: http://topnguyen.net/blog/use-verbal-expressions-to-create-readable-regexs-in-c-sharp

Một **regular expression** là một pattern mà có thể được so khớp với một văn bản đầu vào. .Net Framework cung cấp một phương tiện regular expression mà cho phép so khớp với như thế. , một pattern gồm một hoặc nhiều hằng ký tự, toán tử, hoặc construct.

## Construct cho định nghĩa Regular Expression

Có nhiều loại ký tự, toán tử và construct đa dạng mà giúp bạn định nghĩa Regular Expression . Bạn theo các link sau để có chi tiết về các construct này.

* [Character escape](http://vietjack.com/csharp/escape_character_trong_csharp.jsp)
* [Lớp Character](http://vietjack.com/csharp/lop_character_trong_csharp.jsp)
* [Anchor](http://vietjack.com/csharp/anchor_trong_csharp.jsp)
* [Grouping construct](http://vietjack.com/csharp/grouping_construct_trong_csharp.jsp)
* [Quantifier](http://vietjack.com/csharp/quantifier_trong_csharp.jsp)
* [Backreference construct](http://vietjack.com/csharp/backreference_construct_trong_csharp.jsp)
* [Alternation construct](http://vietjack.com/csharp/alternation_construct_trong_csharp.jsp)
* [Substitution](http://vietjack.com/csharp/substitution_trong_csharp.jsp)
* [Miscellaneous constructs](http://vietjack.com/csharp/construct_hon_hop_trong_csharp.jsp)

## Lớp Regex

Lớp Regex được sử dụng để biểu diễn một Regular Expression. Nó có các phương thức được sử dụng phổ biến sau:

|  |  |
| --- | --- |
| **STT** | **Phương thức** |
| 1 | **public bool IsMatch(string input)**  Chỉ rằng có hay không Regular Expression đã cho trong Regex constructor này tìm thấy một match trong chuỗi đầu vào đã xác định |
| 2 | **public bool IsMatch(string input, int startat)**  Chỉ rằng có hay không Regular Expression đã cho trong Regex constructor này tìm thấy một match trong chuỗi đầu vào đã xác định, bắt đầu tại vị trí startat đã cho trong chuỗi |
| 3 | **public static bool IsMatch(string input, string pattern)**  Chỉ rằng có hay không Regular Expression đã cho tìm thấy một match trong chuỗi đầu vào đã xác định |
| 4 | **public MatchCollection Matches(string input)**  Tìm kiếm chuỗi đầu vào đã xác định về tất cả sự xuất hiện của một Regular Expression |
| 5 | **public string Replace(string input, string replacement)**  Trong một chuỗi đầu vào đã xác định, thay thế tất cả chuỗi mà so khớp với một Regular Expression pattern với một chuỗi thay thế đã cho |
| 6 | **public string[] Split(string input)**  Chia một chuỗi đầu vào thành một mảng các chuỗi phụ tại vị trí được định nghĩa bởi một Regular Expression pattern đã xác định trong Regex constructor |

Để có danh sách đầy đủ các phương thức và thuộc tính, bạn vui lòng đọc Microsoft Documentation về C#.

## Ví dụ 1

Ví dụ sau so khớp với các từ mà bắt đầu với 'S':

using System;

using System.Text.RegularExpressions;

namespace TPCenter

{

class TestCsharp

{

private static void showMatch(string text, string expr)

{

Console.WriteLine("Expression: " + expr);

MatchCollection mc = Regex.Matches(text, expr);

foreach (Match m in mc)

{

Console.WriteLine(m);

}

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Regular Expression ");

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

string str = "Sao hom nay lai thay sao suyen son sang";

Console.WriteLine("So khop voi cac tu bat dau bang chu cai 'S': ");

showMatch(str, @"\bS\S\*");

Console.ReadKey();

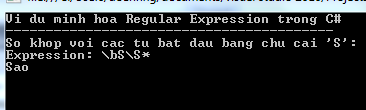
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Ví dụ 2

Ví dụ sau so khớp với các từ mà bắt đầu với 's' và kết thúc với 'n':

using System;

using System.Text.RegularExpressions;

namespace TPCenter

{

class TestCsharp

{

private static void showMatch(string text, string expr)

{

Console.WriteLine("Expression: " + expr);

MatchCollection mc = Regex.Matches(text, expr);

foreach (Match m in mc)

{

Console.WriteLine(m);

}

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Regular Expression ");

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

string str = "Sao hom nay lai thay sao suyen son sang";

Console.WriteLine("So khop voi tu bat dau bang 's' va ket thuc bang 'n':");

showMatch(str, @"\bs\S\*n\b");

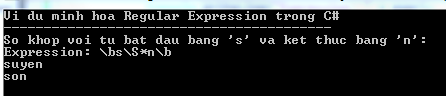
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Ví dụ 3

Ví dụ sau thay thế các white space:

using System;

using System.Text.RegularExpressions;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Regular Expression ");

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

string str1 = "Hello World ";

string pattern = "\\s+";

string str2 = " ";

Regex rgx = new Regex(pattern);

string result = rgx.Replace(str1, str2);

Console.WriteLine("Chuoi ban dau: {0}", str1);

Console.WriteLine("Chuoi sau khi da thay the: {0}", result);

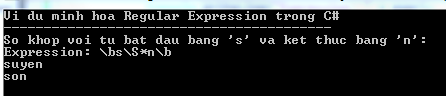
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Exception Handling

Một Exception (ngoại lệ) là một vấn đề xuất hiện trong khi thực thi một chương trình. Một Exception là một phản hồi về một tình huống ngoại lệ mà xuất hiện trong khi một chương trình đang chạy, ví dụ như chia cho số 0.

Exception cung cấp một cách để truyền điều khiển từ một phần của một chương trình tới phần khác. Exception Handling (Xử lý ngoại lệ) được xây dựng dựa trên 4 từ khóa là: **try**, **catch**, **finally**, và **throw**.

* **try**: Một khối try nhận diện một khối code mà ở đó các exception cụ thể được kích hoạt. Nó được theo sau bởi một hoặc nhiều khối catch.
* **catch**: Một chương trình bắt một Exception với một Exception Handler tại vị trí trong một chương trình nơi bạn muốn xử lý vấn đề đó. Từ khóa **catch**  chỉ dẫn việc bắt một exception.
* **finally**: Một khối finally được sử dụng để thực thi một tập hợp lệnh đã cho, dù có hay không một exception đươc ném hoặc không được ném. Ví dụ, nếu bạn mở một file, nó phải được đóng, nếu không sẽ có một exception được tạo ra.
* **throw**: Một chương trình ném một exception khi có một vấn đề xuất hiện. Điều này được thực hiện bởi sử dụng từ khóa **throw** .

## Cú pháp

Giả sử một khối tạo một Exeption, một phương thức bắt một exception bởi sử dụng kết hợp các từ khóa try và catch. Một khối try/catch được đặt xung quanh code mà có thể tạo một exception. Code bên trong một khối try/catch được xem như là code được bảo vệ, và cú pháp để sử dụng try/catch như sau:

try

{

// các lệnh có thể gây ra ngoại lệ (exception)

}

catch( tên\_ngoại\_lệ e1 )

{

// phần code để xử lý lỗi

}

catch( tên\_ngoại\_lệ e2 )

{

// phần code để xử lý lỗi

}

catch( tên\_ngoại\_lệ eN )

{

// phần code để xử lý lỗi

}

finally

{

// các lệnh được thực thi

}

Bạn có thể liệt kê nhiều lệnh catch để bắt các kiểu exception khác nhau trong trường hợp khối try của bạn xuất hiện nhiều hơn một exception trong các tình huống khác nhau.

## Lớp Exception

Các Exception được biểu diễn bởi các lớp. Các lớp Exception chủ yếu được kế thừa một cách trực tiếp hoặc không trực tiếp từ lớp **System.Exception** . Một số lớp Exception kế thừa từ lớp System.Exception là các lớp **System.ApplicationException** và **System.SystemException**.

Lớp **System.ApplicationException** hỗ trợ các exception được tạo bởi các chương trình ứng dụng. Vì thế, các exception được định nghĩa bởi lập trình viên nên kế thừa từ lớp này.

Lớp **System.SystemException** là lớp cơ sở cho tất cả system exception tiền định nghĩa.

Bảng sau cung cấp một số lớp Exception tiền định nghĩa được kế thừa từ lớp *Sytem.SystemException* :

|  |  |
| --- | --- |
| **Lớp Exception** | **Miêu tả** |
| System.IO.IOException | Xử lý các I/O error |
| System.IndexOutOfRangeException | Xử lý các lỗi được tạo khi một phương thức tham chiếu tới một chỉ mục bên ngoài dãy mảng |
| System.ArrayTypeMismatchException | Xử lý các lỗi được tạo khi kiểu là không phù hợp với kiểu mảng |
| System.NullReferenceException | Xử lý các lỗi được tạo từ việc tham chiếu một đối tượng null |
| System.DivideByZeroException | Xử lý các lỗi được tạo khi chia cho số 0 |
| System.InvalidCastException | Xử lý lỗi được tạo trong khi ép kiểu |
| System.OutOfMemoryException | Xử lý lỗi được tạo từ việc thiếu bộ nhớ rỗi |
| System.StackOverflowException | Xử lý lỗi được tạo từ việc tràn ngăn xếp (stack) |

## Xử lý ngoại lệ ( Exception Handling)

C# cung cấp một giải pháp mang tính cấu trúc cao để xử lý ngoại lệ trong form các khối try và catch. Sử dụng các khối này, các lệnh chương trình được phân biệt riêng rẽ với các lệnh xử lý ngoại lệ .

Những khối xử lý ngoại lệ này được triển khai bởi sử dụng các từ khóa **try, catch** và **finally** . Ví dụ sau ném một exception khi chia cho số 0.

using System;

namespace TPCenter

{

class TestCsharp

{

int result;

TestCsharp()

{

result = 0;

}

public void phepChia(int num1, int num2)

{

try

{

result = num1 / num2;

}

catch (DivideByZeroException e)

{

Console.WriteLine("Bat Exception: {0}", e);

}

finally

{

Console.WriteLine("Ket qua: {0}", result);

}

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Exception ");

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

TestCsharp d = new TestCsharp();

d.phepChia(25, 0);

Console.ReadKey();

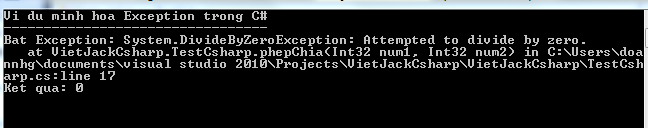
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Tạo User-Defined Exception

Bạn cũng có thể định nghĩa exception cho riêng bạn. Các lớp User-Defined Exception được kế thừa từ lớp **ApplicationException** . Ví dụ sau minh họa điều này:

Tạo 3 lớp có tên lần lượt là như sau:

Lớp **Temperature**

using System;

namespace TPCenter

{

class Temperature

{

int temperature = 0;

public void showTemp()

{

if (temperature == 0)

{

throw (new TempIsZeroException("Muc nhiet do 0!!!"));

}

else

{

Console.WriteLine("Nhiet do: {0}", temperature);

}

}

}

}

Lớp **TempIsZeroException**: đây là một lớp ngoại lệ tự định nghĩa, kế thừa lớp **ApplicationException**

using System;

namespace TPCenter

{

class TempIsZeroException : ApplicationException

{

public TempIsZeroException(string message)

: base(message)

{

}

}

}

Lớp **TestCsharp**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Xu ly ngoai le ");

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

Temperature temp = new Temperature();

try

{

temp.showTemp();

}

catch (TempIsZeroException e)

{

Console.WriteLine("TempIsZeroException: {0}", e.Message);

}

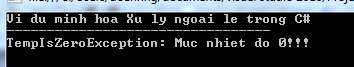
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Ném các Object

Bạn có thể ném một đối tượng nếu nó: hoặc trực tiếp hoặc gián tiếp được kế thừa từ lớp **System.Exception** . Bạn có thể sử dụng một lệnh throw trong khối catch để ném đối tượng hiện diện đó:

Catch(Exception e)

{

...

Throw e

}

# File I/O

Một **file** là một tập hợp dữ liệu được lưu giữ trong một disk với một tên cụ thể và một path thư mục. Khi một file được mở để đọc hoặc ghi, nó trở thành một **stream**.

Về cơ bản, stream là dãy các byte truyền qua path. Có hai stream quan trọng: **Input stream** và **Output stream**. Input stream được sử dụng để đọc dữ liệu từ file (hoạt động read) và Output stream được sử dụng để ghi vào trong file (hoạt động write).

## Lớp I/O

System.IO namespace có nhiều lớp đa dạng mà được sử dụng để thực hiện các hoạt động khác nhau với File, như tạo và xóa file, đọc và ghi một File, đóng một File, …

Bảng sau hiển thị một số lớp **non-abstract** được sử dụng phổ biến trong System.IO namespace :

|  |  |
| --- | --- |
| **I/O Class** | **Miêu tả** |
| BinaryReader | Đọc dữ liệu gốc (primitive data) từ một binary stream |
| BinaryWriter | Ghi dữ liệu gốc trong định dạng nhị phân |
| BufferedStream | Một nơi lưu giữ tạm thời cho một stream |
| Directory | Giúp ích trong việc thao tác một cấu trúc thư mục |
| DirectoryInfo | Được sử dụng để thực hiện các hoạt động trên các thư mục |
| DriveInfo | Cung cấp thông tin cho các Drive |
| File | Giúp ích trong việc thao tác các File |
| FileInfo | Được sử dụng để thực hiện các hoạt động trên các File |
| FileStream | Được sử dụng để đọc và ghi bất kỳ vị trí nào trong một File |
| MemoryStream | Được sử dụng để truy cập ngẫu nhiên tới stream được lưu giữ trong bộ nhớ |
| Path | Thực hiện các hoạt động trên thông tin path |
| StreamReader | Được sử dụng để đọc các ký tự từ một stream |
| StreamWriter | Được sử dụng để ghi các ký tự tới một stream |
| StringReader | Được sử dụng để đọc từ một string buffer |
| StringWriter | Được sử dụng để ghi vào một string buffer |

## Lớp FileStream

Lớp **FileStream** trong System.IO namespace giúp đỡ trong việc đọc từ, ghi và đóng các File. Lớp này kế thừa từ lớp abstract là Stream.

Bạn cần tạo một đối tượng **FileStream** để tạo một File mới hoặc mở một File đang tồn tại. Cú pháp để tạo một đối tượng **FileStream**  như sau:

FileStream <tên\_đối\_tượng> = new FileStream( <tên\_file>, <FileMode>, <FileAccess>, <FileShare>);

Ví dụ: chúng ta tạo một đối tượng FileStream là **F** để đọc một File với tên **sample.txt**, như sau:

FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);

|  |  |
| --- | --- |
| **Tham số** | **Miêu tả** |
| FileMode | **FileMode** enumerator định nghĩa các phương thức đa dạng để mở các File. Các thành viên của FileMode enumerator là:   * **Append**: Nó mở một File đang tồn tại và đặt con trỏ tại phần cuối của File, hoặc tạo File, nếu File đó chưa tồn tại * **Create**: Nó tạo một File mới * **CreateNew**: Nó xác định tới Hệ điều hành rằng nó nên tạo một File mới * **Open**: Nó mở một File đang tồn tại * **OpenOrCreate**: Nó xác định tới Hệ điều hành rằng nó nên mở một File nếu nó tồn tại, nếu không thì nó nên tạo một File mới * **Truncate**: Nó mở một File đang tồn tại và truncate (cắt) kích cỡ của nó về 0 byte |
| FileAccess | **FileAccess** enumerators có các thành viên là: **Read**, **ReadWrite** và **Write**. |
| FileShare | **FileShare** enumerators có các thành viên sau:   * **Inheritable**: Nó cho phép một File truyền tính kế thừa tới các tiến trình con * **None**: Nó từ chối việc chia sẻ File hiện tại * **Read**: Nó cho phép mở File để đọc * **ReadWrite**: Nó cho phép mở File để đọc và ghi * **Write**: Nó cho phép mở File để ghi |

## Ví dụ

Dưới đây là ví dụ minh họa cách sử dụng của lớp **FileStream**:

using System;

using System.IO;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa File I/O ");

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

FileStream F = new FileStream("binary.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);

for (int i = 1; i <= 20; i++)

{

F.WriteByte((byte)i);

}

F.Position = 0;

for (int i = 0; i <= 20; i++)

{

Console.Write(F.ReadByte() + " ");

}

F.Close();

Console.ReadKey();

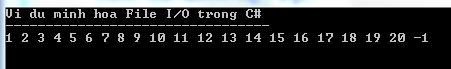
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Hoạt động File nâng cao

Ví dụ trước minh họa các hoạt động đơn giản trên File . Tuy nhiên, để lợi dụng tối đa sức mạnh của các lớp System.IO , bạn cần biết các thuộc tính và phương thức được sử dụng phổ biến về các lớp này.

|  |
| --- |
| **Chủ đề và Miêu tả** |
| [**Đọc và ghi Text File**](http://vietjack.com/csharp/doc_va_ghi_text_file_trong_csharp.jsp)  Lớp **StreamReader** và **StreamWriter** giúp thực hiện việc đọc và ghi các File văn bản |
| [**Đọc và ghi File nhị phân**](http://vietjack.com/csharp/doc_va_ghi_binary_file_trong_csharp.jsp)  Lớp **BinaryReader** và **BinaryWriter** giúp thực hiện việc đọc và ghi các File nhị phân |
| [**Thao tác Windows file system**](http://vietjack.com/csharp/windows_file_system_trong_csharp.jsp)  Nó cung cấp cho lập trình viên C# khả năng duyệt và xác định vị trí các File và thư mục trong Windows |

# Attributes

Một **attribute** , là một thẻ tường thuật, được sử dụng để truyền thông tin tới runtime về các hành vi của các phần tử đa dạng như các lớp, phương thức, cấu trúc, enum, assembly, … trong chương trình của bạn. Bạn có thể thêm thông tin tường thuật tới một chương trình bởi việc sử dụng một **Attribute**. Một thẻ tường thuật được miêu tả bởi các dấu ngoặc móc vuông ([]) được đặt bên trên phần tử mà nó được sử dụng cho.

Các Attribute được sử dụng để thêm metadata, ví dụ như chỉ lệnh biên dịch và thông tin khác như comment, miêu tả, phương thức và các lớp tới một chương trình. .Net Framework cung cấp hai kiểu Attribute: các Attribute được định nghĩa trước và các Custom Attribute.

## Xác định một Attribute

Cú pháp để xác định một Attribute như sau:

[attribute(positional\_parameter, name\_parameter = giá\_trị, ...)]

element

Tên của Attribute và giá trị của nó được xác định bên trong dấu ngoặc vuông, ở trước phần tử từ đó thuộc tính được áp dụng cho. *positional\_parameter* xác định thông tin thiết yếu và *name\_parameter* xác định thông tin tùy ý.

## Attribute được định nghĩa trước

.Net Framework cung cấp 3 Attribute được định nghĩa trước:

* AttributeUsage
* Conditional
* Obsolete

## AttributeUsage

Attribute được định nghĩa trước **AttributeUsage** miêu tả cách một lớp custom Attribute có thể được sử dụng. Nó xác định kiểu của các item, mà từ đó Attribute có thể áp dụng cho.

Cú pháp để xác định Attribute này như sau:

[AttributeUsage(

validon,

AllowMultiple=allowmultiple,

Inherited=inherited

)]

Tại đây:

* Tham số *validon* xác định các phần tử ngôn ngữ mà Attribute có thể được đặt. Nó là một sự tổ hợp giá trị của một **AttributeTargets** enumerator. Giá trị mặc định là **AttributeTargets.All**.
* Tham số *allowmultiple* (tùy ý) cung cấp giá trị cho thuộc tính **AllowMultiple** của attribute này, một giá trị Boolean. Nếu điều này là true, Attribute là multiuse. Giá trị mặc định là false (tức là single-use).
* Tham số *inherited* (tùy ý) cung cấp giá trị cho thuộc tính **Inherited** của attribute này, một giá trị Boolean. Nếu nó là true, Attribute được kế thừa bởi các lớp kế thừa. Giá trị mặc định là false (không được kế thừa).

Ví dụ:

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Feild |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

## Conditional

Attribute tiền định nghĩa này đánh dấu một phương thức có điều kiện mà sự thực thi của nó phụ thuộc vào một tiến trình tiền xử lý định danh đã cho.

Nó tạo sự biên dịch có điều kiện của các lời gọi phương thức, phụ thuộc vào giá trị đã cho, như **Debug** hoặc **Trace**. Ví dụ: nó hiển thị các giá trị của các biến trong khi debug một code.

Cú pháp để xác định Attribute này là như sau:

[Conditional(

conditionalSymbol

)]

Ví dụ:

[Conditional("DEBUG")]

Sau đây là ví dụ minh họa Conditional : Tạo hai lớp có tên lần lượt là **TestAttribute, TestScharp** như sau:

Lớp **TestAttribute**:

#define DEBUG

using System;

using System.Diagnostics;

namespace TPCenterCsharp

{

class TestAttribute

{

[Conditional("DEBUG")]

public static void Message(string msg)

{

Console.WriteLine(msg);

}

}

}

Lớp **TestCsharp**:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static void function1()

{

TestAttribute.Message("Trong Function 1.");

function2();

}

static void function2()

{

TestAttribute.Message("Trong Function 2.");

}

public static void Main()

{

Console.WriteLine("Attribute ");

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

TestAttribute.Message("Trong ham Main.");

function1();

Console.ReadKey();

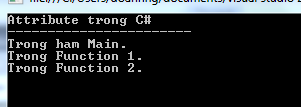
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Obsolete

Attribute tiền định nghĩa này đánh dấu một thực thể chương trình mà không nên được sử dụng. Nó cho bạn khả năng để thông báo cho compiler để loại bỏ một phần tử target cụ thể. Ví dụ, khi một phương thức mới đang được sử dụng trong một lớp và nếu bạn vẫn muốn giữ lại phương thức cũ trong lớp này, bạn có thể đánh dấu nó là obsolete bằng việc hiển thị một thông báo là phương thức mới nên được sử dụng, thay cho phương thức cũ.

Cú pháp để xác định Attribute này là như sau:

[Obsolete(

message

)]

[Obsolete(

message,

iserror

)]

Tại đây,

* Tham số *message* là một chuỗi miêu tả lý do tại sao item là obsolete và cái gì được sử dụng thay cho nó.
* Tham số *iserror* là một giá trị Boolean. Nếu giá trị của nó là true, compiler nên đối xử sự sử dụng của item này như là một lỗi. Giá trị mặc định là false (tức là compiler tạo một warning).

Ví dụ sau minh họa obsolete :

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

[Obsolete("Dung su dung PhuongThuc1, ban nen su dung PhuongThuc2", true)]

static void PhuongThuc1()

{

Console.WriteLine("Day la phuong thuc 1.");

}

static void PhuongThuc2()

{

Console.WriteLine("Day la phuong thuc 2.");

}

public static void Main()

{

PhuongThuc1(); // co dau gach do bao hieu tai day

}

}

}

Khi đó, ở lời gọi **PhuongThuc1()** trong hàm **main()** sẽ xuất hiện dấu gạch đỏ, nếu bạn rê chuột qua sẽ có một thông báo có dạng:

'TPCenterCsharp.TestCsharp.PhuongThuc1()' is obsolete: 'Dung su dung PhuongThuc1, ban nen su dung PhuongThuc2'

## Tạo Custom Attribute

Còn gọi là Attribute tùy biến hay Attribute do người dùng tự định nghĩa. .Net Framework cho phép tạo các Custom Attribute mà có thể được sử dụng để lưu giữ thông tin tường thuật và có thể được thu nhận tại runtime. Thông tin này có thể liên quan tới bất kỳ phần tử target nào phụ thuộc vào chuẩn thiết kế và yêu cầu ứng dụng.

Tạo và sử dụng Custom Attribute bao gồm 4 bước sau:

* Khai báo một Custom Attribute
* Xây dựng Custom Attribute
* Áp dụng Attribute trên một phần tử chương trình target
* Truy cập các Attribute thông qua Reflection

Bước cuối cùng liên quan tới việc viết một chương trình đơn giản để đọc qua metadata để tìm ra các notation đa dạng. Metadata là dữ liệu hoặc thông tin được sử dụng để miêu tả dữ liệu khác. Chương trình này nên sử dụng các Reflection để truy cập các Attribute tại runtime. Chúng ta sẽ bàn luận điều này trong chương tới.

## Khai báo một Custom Attribute

Một Custom Attribute mới nên được kế thừa từ lớp **System.Attribute** . Ví dụ:

//a custom attribute BugFix to be assigned to a class and its members

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Field |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

public class DeBugInfo : System.Attribute

Trong code trên, chúng ta đã khai báo một Custom Attribute là *DeBugInfo*.

## Xây dựng Custom Attribute

Chúng ta cùng xây dựng Custom Attribute có tên là *DeBugInfo*, mà lưu giữ thông tin thu được bởi việc debug bất kỳ chương trình nào. Nó có thể giữ thông tin sau:

* Số hiệu code để bug
* Tên lập trình viên, người nhận diện bug đó
* Ngày review cuối cùng của code đó
* Một thông báo dạng chuỗi để lưu giữ các lưu ý của lập trình viên

Lớp *DeBugInfo* có 3 thuộc tính private để lưu giữ 3 thông tin đầu tiên và một thuộc tính public để lưu giữ thông báo đó. Vì thế, số hiệu bug, tên lập trình viên, và ngày review là các tham số vị trí tương ứng của lớp DeBugInfo và thông báo là một tham số tùy ý.

Mỗi Attribute phải có ít nhất một constructor. Các tham số vị trí tương ứng nên được truyền thông qua constructor đó. Ví dụ sau minh họa lớp *DeBugInfo* trên:

//Vi du minh hoa mot custom attribute BugFix

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Field |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

public class DeBugInfo : System.Attribute

{

private int bugNo;

private string developer;

private string lastReview;

public string message;

public DeBugInfo(int bg, string dev, string d)

{

this.bugNo = bg;

this.developer = dev;

this.lastReview = d;

}

public int BugNo

{

get

{

return bugNo;

}

}

public string Developer

{

get

{

return developer;

}

}

public string LastReview

{

get

{

return lastReview;

}

}

public string Message

{

get

{

return message;

}

set

{

message = value;

}

}

}

## Áp dụng Custom Attribute

Custom Attribute được áp dụng bằng việc đặt nó ngay trước target của nó:

[DeBugInfo(45, "Tran Nam", "2/8/2016", Message = "Kieu tra ve khong hop le")]

[DeBugInfo(49, Thanh Phuong, "10/10/2016", Message = "Bien chua duoc su dung")]

class Rectangle

{

//cac bien thanh vien

protected double chieu\_dai;

protected double chieu\_rong;

public Rectangle(double l, double w)

{

chieu\_dai = l;

chieu\_rong = w;

}

[DeBugInfo(55, "Tran Nam", "2/8/2016", Message = "Kieu tra ve khong hop le")]

public double tinhDienTich()

{

return chieu\_dai \* chieu\_rong;

}

[DeBugInfo(56, Thanh Phuong, "10/10/2016")]

public void Display()

{

Console.WriteLine("Chieu dai: {0}", chieu\_dai);

Console.WriteLine("Chieu rong: {0}", chieu\_rong);

Console.WriteLine("Dien tich: {0}", tinhDienTich());

}

}

Trong chương tới, chúng ta thu hồi thông tin Attribute bởi sử dụng một đối tượng lớp Reflection .

# Reflection

Các đối tượng **Reflection** được sử dụng để thu được thông tin kiểu tại runtime. Các lớp này cung cấp truy cập tới metadata của một chương trình đang chạy là trong **System.Reflection** namespace .

**System.Reflection** namespace chứa các lớp mà cho phép bạn thu được thông tin về ứng dụng và để thêm các kiểu, giá trị, và các đối tượng một cách động tới Application.

Các ứng dụng của Reflection

Reflection có các ứng dụng sau:

* Nó cho phép quan sát thông tin attribute tại runtime.
* Nó cho phép thẩm tra các kiểu đa dạng trong một Assembly và khởi tạo các kiểu này.
* Nó cho phép Late Binding tới các phương thức và các thuộc tính.
* Nó cho phép tạo các kiểu mới tại runtime và sau đó thực hiện một số tác vụ bởi sử dụng những kiểu này.

Quan sát Metadata

Chúng ta đã đề cập trong chương trước rằng với việc sử dụng Reflection, bạn có thể quan sát thông tin attribute.

Đối tượng **MemberInfo** của lớp **System.Reflection**  cần được khởi tạo để phát hiện ra các attribute được liên kết với một lớp. Để làm điều này, bạn định nghĩa một đối tượng của lớp target, như:

System.Reflection.MemberInfo info = typeof(MyClass);

Ví dụ sau minh họa điều này: tạo 3 lớp có tên lần lượt là **HelpAttribute, MyClass, TestCsharp** như sau:

Lớp **HelpAttribute**

using System;

namespace TPCenterCsharp

{

[AttributeUsage(AttributeTargets.All)]

public class HelpAttribute : System.Attribute

{

public readonly string Url;

public string Topic // Topic la mot name parameter

{

get

{

return topic;

}

set

{

topic = value;

}

}

public HelpAttribute(string url) // url la mot positional parameter

{

this.Url = url;

}

private string topic;

}

}

Lớp **MyClass**

using System;

namespace TPCenterCsharp

{

[HelpAttribute("Thong tin tren lop MyClass")]

class MyClass

{

}

}

Lớp **TestCsharp**

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Reflection ");

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

System.Reflection.MemberInfo info = typeof(MyClass);

object[] attributes = info.GetCustomAttributes(true);

for (int i = 0; i < attributes.Length; i++)

{

System.Console.WriteLine(attributes[i]);

}

Console.ReadKey();

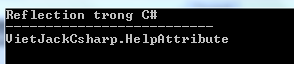
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau: hiển thị tên của các custom attribute được đính kèm tới lớp *MyClass*



Ví dụ

Trong ví dụ này, chúng ta sử dụng attribute là *DeBugInfo* được tạo trong chương trước và sử dụng Reflection để đọc metadata trong *Rectangle* class.

Tạo 3 lớp có tên lần lượt là

Lớp **Rectangle**:

using System;

namespace TPCenterCsharp

{

[DeBugInfo(45, "Tran Nam", "2/8/2016", Message = "Kieu tra ve khong hop le")]

[DeBugInfo(49, Thanh Phuong, "10/10/2016", Message = "Bien khong duoc su dung")]

class Rectangle

{

//cac bien thanh vien

protected double chieu\_dai;

protected double chieu\_rong;

public Rectangle(double l, double w)

{

chieu\_dai = l;

chieu\_rong = w;

}

[DeBugInfo(55, "Tran Nam", "2/8/2016", Message = "Kieu tra ve khong hop le")]

public double tinhDienTich()

{

return chieu\_dai \* chieu\_rong;

}

[DeBugInfo(56, Thanh Phuong, "19/10/2016")]

public void Display()

{

Console.WriteLine("Chieu dai: {0}", chieu\_dai);

Console.WriteLine("Chieu rong: {0}", chieu\_rong);

Console.WriteLine("Dien tich: {0}", tinhDienTich());

}

}

}

Lớp **DeBugInfo**:

using System;

using System.Reflection;

namespace TPCenterCsharp

{

//Mot custom attribute BugFix

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Field |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

public class DeBugInfo : System.Attribute

{

private int bugNo;

private string developer;

private string lastReview;

public string message;

public DeBugInfo(int bg, string dev, string d)

{

this.bugNo = bg;

this.developer = dev;

this.lastReview = d;

}

public int BugNo

{

get

{

return bugNo;

}

}

public string Developer

{

get

{

return developer;

}

}

public string LastReview

{

get

{

return lastReview;

}

}

public string Message

{

get

{

return message;

}

set

{

message = value;

}

}

}

}

Lớp **TestCsharp**:

using System;

using System.Reflection;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Reflection ");

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

Rectangle r = new Rectangle(4.5, 7.5);

r.Display();

Type type = typeof(Rectangle);

//lap qua cac attribtue cua lop Rectangle

foreach (Object attributes in type.GetCustomAttributes(false))

{

DeBugInfo dbi = (DeBugInfo)attributes;

if (null != dbi)

{

Console.WriteLine("Bug no: {0}", dbi.BugNo);

Console.WriteLine("Developer: {0}", dbi.Developer);

Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);

Console.WriteLine("Remarks: {0}", dbi.Message);

}

}

//lap qua cac method attribtue

foreach (MethodInfo m in type.GetMethods())

{

foreach (Attribute a in m.GetCustomAttributes(true))

{

DeBugInfo dbi = (DeBugInfo)a;

if (null != dbi)

{

Console.WriteLine("Bug no: {0}, for Method: {1}", dbi.BugNo, m.Name);

Console.WriteLine("Developer: {0}", dbi.Developer);

Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);

Console.WriteLine("Remarks: {0}", dbi.Message);

}

}

}

Console.ReadLine();

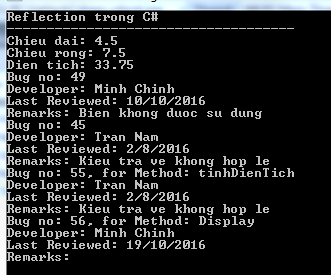
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Properties

**Thuộc tính - Property** là các thành viên được đặt tên của các lớp, cấu trúc, và Interface. Các biến thành viên hoặc các phương thức trong một lớp hoặc cấu trúc được gọi là các **Field**. Thuộc tính là một sự kế thừa của các Field và được truy cập bởi sử dụng cùng cú pháp. Chúng sử dụng **accessor** thông qua các giá trị của các Private Field có thể được đọc, được viết và được thao tác.

Thuộc tính (Property) không đặt tên các vị trí lưu giữ. Thay vào đó, chúng có accessors mà đọc, ghi hoặc tính toán các giá trị của chúng.

Ví dụ, chúng ta có một lớp với tên Student, với các Private Field cho age, name, và code. Chúng ta không thể trực tiếp truy cập các Field này từ bên ngoài phạm vi lớp đó, nhưng chúng ta có thể có các thuộc tính để truy cập các Private Field này.

## Accessor

, **accessor** là một thuộc tính chứa các lệnh có thể thực thi, mà giúp đỡ trong việc *lấy* (đọc hoặc tính toán) hoặc *thiết lập* (ghi) thuộc tính. Các khai báo accessor có thể thu được một get accessor, một set accessor, hoặc cả hai. Ví dụ:

// khai báo một thuộc tính Code có kiểu dữ liệu string:

public string Code

{

get

{

return code;

}

set

{

code = value;

}

}

// khai báo một thuộc tính Name có kiểu dữ liệu String:

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

// khai báo một thuộc tính Age có kiểu dữ liệu int:

public int Age

{

get

{

return age;

}

set

{

age = value;

}

}

## Ví dụ

Ví dụ dưới đây minh họa cách sử dụng của các thuộc tính : tạo 2 lớp có tên lần lượt là **Student, TestCsharp** như sau:

Lớp **Student**:

using System;

namespace TPCenterCsharp

{

class Student

{

private string code = "N/A";

private string name = "unknown";

private int age = 0;

// khai bao thuoc tinh Code co kieu string:

public string Code

{

get

{

return code;

}

set

{

code = value;

}

}

// khai bao thuoc tinh Name co kieu string:

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

// khai bao thuoc tinh Age co kieu int:

public int Age

{

get

{

return age;

}

set

{

age = value;

}

}

public override string ToString()

{

return "MSSV = " + Code + ", Ho Ten = " + Name + ", Tuoi = " + Age;

}

}

}

Lớp **TestCsharp**:

using System;

using System.Reflection;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Thuoc tinh (Property) ");

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

// tao mot doi tuong Student

Student s = new Student();

// thiet lap cac thuoc tinh code, name va age cho Student

s.Code = "001";

s.Name = Thanh Phuong;

s.Age = 21;

Console.WriteLine("Thong tin sinh vien: {0}", s);

//bay gio tang age them 1

s.Age += 1;

Console.WriteLine("Thong tin sinh vien: {0}", s);

Console.ReadLine();

Console.ReadKey();

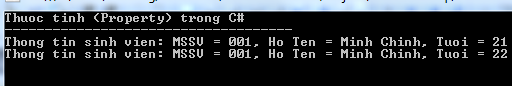
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Thuộc tính abstract

Một lớp Abstract có thể có một thuộc tính abstract, mà nên được triển khai trong lớp kế thừa. Chương trình sau minh họa điều này:

Tạo 3 lớp có tên lần lượt là **Person, Student, TestCsharp** như sau:

Lớp abstract **Person**:

using System;

namespace TPCenterCsharp

{

public abstract class Person

{

public abstract string Name

{

get;

set;

}

public abstract int Age

{

get;

set;

}

}

}

Lớp **Student**:

using System;

namespace TPCenterCsharp

{

class Student : Person

{

private string code = "N/A";

private string name = "N/A";

private int age = 0;

// khai bao thuoc tinh Code co kieu string:

public string Code

{

get

{

return code;

}

set

{

code = value;

}

}

// khai bao thuoc tinh Name co kieu string:

public override string Name

{

get

{

return name;

}

set

{

name = value;

}

}

// khai bao thuoc tinh Age co kieu int:

public override int Age

{

get

{

return age;

}

set

{

age = value;

}

}

public override string ToString()

{

return "MSSV = " + Code + ", Ho ten = " + Name + ", Tuoi = " + Age;

}

}

}

Lớp **TestCsharp**

using System;

using System.Reflection;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Thuoc tinh (Property) ");

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

// tao mot doi tuong Student

Student s = new Student();

// thiet lap code, name va age cho Student

s.Code = "001";

s.Name = Thanh Phuong;

s.Age = 21;

Console.WriteLine("Thong tin sinh vien: {0}", s);

//bay gio tang age them 1

s.Age += 1;

Console.WriteLine("Thong tin sinh vien: {0}", s);

Console.ReadLine();

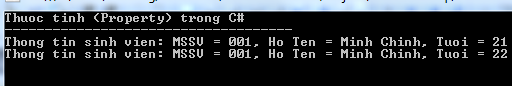
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Indexer

Một **indexer**  cho phép một đối tượng để được lập chỉ mục, ví dụ như một mảng. Khi bạn định nghĩa một indexer cho một lớp, thì lớp này vận hành tương tự như một **virtual array (mảng ảo)**. Sau đó, bạn có thể truy cập instance (sự thể hiện) của lớp này bởi sử dụng toán tử truy cập mảng là ([ ]).

## Cú pháp

Một Indexer một chiều có cú pháp như sau :

kiểu\_phần\_tử this[int index]

{

get

{

// trả về giá trị được xác định bởi index

}

set

{

// thiết lập giá trị được xác định bởi index

}

}

## Sự sử dụng của Indexer

Việc khai báo hành vi của một Indexer là tương tự như một thuộc tính. Bạn sử dụng **get** accessor và **set** accessor để định nghĩa một Indexer. Tuy nhiên, các thuộc tính trả về hoặc thiết lập một thành viên dữ liệu cụ thể, trong khi đó, Indexer trả về hoặc thiết lập một giá trị cụ thể từ instance của đối tượng. Nói cách khác, nó chia dữ liệu của instance thành các phần nhỏ hơn và đánh chỉ mục mỗi phần, lấy hoặc thiết lập mỗi phần.

Việc định nghĩa một thuộc tính bao gồm việc cung cấp một tên thuộc tính. Indexer không được định nghĩa với các tên, nhưng với từ khóa **this** , mà tham chiếu tới instance của đối tượng. Ví dụ sau đây minh họa khái niệm này:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

private string[] namelist = new string[size];

static public int size = 10;

public TestCsharp()

{

for (int i = 0; i < size; i++)

namelist[i] = "N/A";

}

public string this[int index]

{

get

{

string tmp;

if (index >= 0 && index <= size - 1)

{

tmp = namelist[index];

}

else

{

tmp = "";

}

return (tmp);

}

set

{

if (index >= 0 && index <= size - 1)

{

namelist[index] = value;

}

}

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Indexer ");

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

//tao doi tuong TestCsharp

TestCsharp names = new TestCsharp();

names[0] = "Nam";

names[1] = "Hoang";

names[2] = "Dung";

names[3] = "Hue";

names[4] = "Huong";

names[5] = "Phuc";

names[6] = "Trung";

for (int i = 0; i < TestCsharp.size; i++)

{

Console.WriteLine(names[i]);

}

Console.ReadKey();

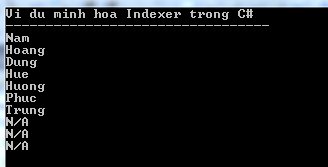
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Nạp chồng Indexer

Indexer có thể được nạp chồng. Indexer cũng có thể được khai báo với nhiều tham số và mỗi tham số có thể là một kiểu khác nhau. Nó không cần thiết để các Indexer phải là integer. C# cho phép các chỉ mục có thể là các kiểu giá trị khác, ví dụ, một kiểu string.

Ví dụ sau minh họa cách nạp chồng Indexer :

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

private string[] namelist = new string[size];

static public int size = 10;

public TestCsharp()

{

for (int i = 0; i < size; i++)

{

namelist[i] = "N/A";

}

}

public string this[int index]

{

get

{

string tmp;

if (index >= 0 && index <= size - 1)

{

tmp = namelist[index];

}

else

{

tmp = "";

}

return (tmp);

}

set

{

if (index >= 0 && index <= size - 1)

{

namelist[index] = value;

}

}

}

public int this[string name]

{

get

{

int index = 0;

while (index < size)

{

if (namelist[index] == name)

{

return index;

}

index++;

}

return index;

}

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Indexer ");

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

//tao doi tuong TestCsharp

TestCsharp names = new TestCsharp();

names[0] = "Nam";

names[1] = "Hoang";

names[2] = "Dung";

names[3] = "Hue";

names[4] = "Huong";

names[5] = "Phuc";

names[6] = "Trung";

//su dung indexer thu nhat voi tham so int

for (int i = 0; i < TestCsharp.size; i++)

{

Console.WriteLine(names[i]);

}

//su dung indexer thu nhat voi tham so string

Console.WriteLine(names["Hue"]);

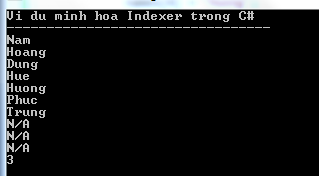
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Delegate

Delegate là tương tự như con trỏ tới các hàm, trong C hoặc trong C++. Một **Delegate** là một biến kiểu tham chiếu mà giữ tham chiếu tới một phương thức. Tham chiếu đó có thể được thay đổi tại runtime.

Đặc biệt, các delegate được sử dụng để triển khai các sự kiện và các phương thức call-back. Tất cả delegate được kế thừa một cách ngầm định từ lớp **System.Delegate** .

## Khai báo Delegate

Khai báo Delegate quyết định các phương thức mà có thể được tham chiếu bởi Delegate đó. Một Delegate có thể tham chiếu tới một phương thức, mà có cùng dấu hiệu như của Delegate đó.

Ví dụ, bạn xét một delegate sau đây:

public delegate int MyDelegate (string s);

Delegate trên có thể được sử dụng để tham chiếu bất kỳ phương thức mà có một tham số *string* đơn và trả về một biến kiểu *int*.

Cú pháp để khai báo delegate là:

delegate <kiểu\_trả\_về> <tên\_delegate> <danh\_sách\_tham\_số>

## Khởi tạo Delegate

Khi một kiểu delegate được khai báo, một đối tượng delegate phải được tạo với từ khóa **new** và được liên kết với một phương thức cụ thể. Khi tạo một delegate, tham số được truyền tới biểu thức new được viết tương tự như một lời gọi phương thức, nhưng không có tham số tới phương thức đó. Ví dụ:

public delegate void printString(string s);

...

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

Ví dụ sau minh họa cách khai báo, khởi tạo và sử dụng một delegate mà có thể được sử dụng để tham chiếu các phương thức mà nhận một tham số integer và trả về một giá trị integer.

using System;

delegate int NumberChanger(int n);

namespace TPCenterCsharp

{

class TestCsharp

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Delegate ");

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

//tao cac doi tuong delegate

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

//goi cac phuong thuc boi su dung cac doi tuong delegate

nc1(25);

Console.WriteLine("Gia tri cua num la: {0}", getNum());

nc2(5);

Console.WriteLine("Gia tri cua num la: {0}", getNum());

Console.ReadKey();

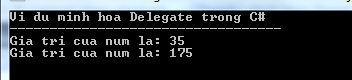
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Multicast một Delegate

Các đối tượng Delegate có thể được hợp thành bởi sử dụng toán tử "+". Một delegate được hợp thành gọi hai Delegate mà nó được hợp thành từ đó. Chỉ có các delegate cùng kiểu có thể được hợp thành. Toán tử "-" có thể được sử dụng để gỡ bỏ một delegate thành phần từ một delegate được hợp thành.

Sử dụng thuộc tính này của các delegate, bạn có thể tạo một danh sách triệu hồi của các phương thức mà sẽ được gọi khi delegate đó được triệu hồi. Điều này được gọi là **Multicasting** của một Delegate. Chương trình ví dụ sau minh họa Multicasting của một Delegate :

using System;

delegate int NumberChanger(int n);

namespace TPCenterCsharp

{

class TestCsharp

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Delegate ");

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

//tao cac doi tuong delegate

NumberChanger nc;

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

nc = nc1;

nc += nc2;

//goi multicast

nc(5);

Console.WriteLine("Gia tri cua num la: {0}", getNum());

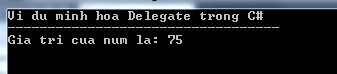
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Cách sử dụng Delegate

Ví dụ sau minh họa cách sử dụng của delegate . Delegate với tên *printString* có thể được sử dụng để tham chiếu phương thức mà nhận một chuỗi như là input và không trả về cái gì.

Chúng ta sử dụng delegate này để gọi hai phương thức: phương thức đầu tiên in chuỗi tới Console, và phương thức thứ hai in nó tới một File.

using System;

using System.IO;

namespace TPCenterCsharp

{

class TestCsharp

{

static FileStream fs;

static StreamWriter sw;

// khai bao delegate

public delegate void printString(string s);

// phuong thuc de in tren console

public static void WriteToScreen(string str)

{

Console.WriteLine("Chuoi la: {0}", str);

}

//phuong thuc nay de ghi du lieu vao file

public static void WriteToFile(string s)

{

fs = new FileStream("c:\\message.txt",

FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

sw.WriteLine(s);

sw.Flush();

sw.Close();

fs.Close();

}

// phuong thuc nay nhan delegate lam tham so va su dung no de

// goi cac phuong thuc neu can

public static void sendString(printString ps)

{

ps("Hoc C# co ban va nang cao tai TPCenter");

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Delegate ");

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

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

sendString(ps1);

sendString(ps2);

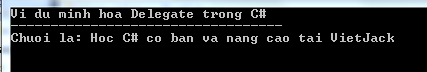
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Event

**Sự kiện (Event)** là các hành động của người dùng, ví dụ như nhấn phím, click, di chuyển chuột, … Các Application cần phản hồi các sự kiện này khi chúng xuất hiện. Ví dụ, các ngắt (interrupt). Các sự kiện (Event) được sử dụng để giao tiếp bên trong tiến trình.

## Sử dụng Delegate với Event

Các Event được khai báo và được tạo trong một lớp và được liên kết với Event Handler bởi sử dụng các Delegate bên trong cùng lớp đó hoặc một số lớp khác. Lớp mà chứa Event được sử dụng để công bố event đó. Điều này được gọi là lớp **Publisher**. Một số lớp khác mà chấp nhận Event này được gọi là lớp **Subscriber**. Các Event sử dụng mô hình **Publisher-Subscriber**.

Một **Publisher**  là một đối tượng mà chứa định nghĩa của event và delegate đó. Mối liên hệ event-delegate cũng được định nghĩa trong đối tượng này. Một đối tượng lớp Publisher triệu hồi Event và nó được thông báo tới các đối tượng khác.

Một **Subscriber**  là một đối tượng mà chấp nhận event và cung cấp một Event Handler. Delegate trong lớp Publisher triệu hồi phương thức (Event Handler) của lớp Subscriber.

## Khai báo Event

Để khai báo một Event bên trong một lớp, đầu tiên một kiểu delegate cho Event đó phải được khai báo. Ví dụ:

public delegate void BoilerLogHandler(string status);

Tiếp theo, chính Event đó được khai báo, bởi sử dụng từ khóa **event** :

//định nghĩa event dựa vào delegate ở trên

public event BoilerLogHandler BoilerEventLog;

Code trên định nghĩa một delegate với tên là *BoilerLogHandler* và một Event với tên là *BoilerEventLog*, mà triệu hồi delegate đó khi nó được tạo ra.

## Ví dụ 1

Tạo hai lớp có tên lần lượt là **EventTest, TestCsharp** như sau:

Lớp **EventTest**:

using System;

namespace TPCenterCsharp

{

class EventTest

{

private int value;

public delegate void NumManipulationHandler();

public event NumManipulationHandler ChangeNum;

protected virtual void OnNumChanged()

{

if (ChangeNum != null)

{

ChangeNum();

}

else

{

Console.WriteLine("Kich hoat su kien!");

}

}

public EventTest(int n)

{

SetValue(n);

}

public void SetValue(int n)

{

if (value != n)

{

value = n;

OnNumChanged();

}

}

}

}

Lớp **TestCsharp**:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Su kien (Event) ");

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

//tao doi tuong EventTest

EventTest e = new EventTest(5);

e.SetValue(7);

e.SetValue(11);

Console.ReadKey();

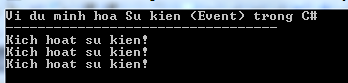
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Ví dụ 2

Ví dụ này cung cấp một ứng dụng đơn giản để xử lý sự cố cho một hệ thống nồi hơn đun nước nóng. Khi kỹ sư bảo dưỡng kiểm tra nồi hơi, nhiệt độ và áp suất nồi hơi được tự động ghi lại vào trong một log file cùng với các ghi chú của kỹ sư bảo dưỡng này.

Tạo 4 lớp có tên lần lượt là **Boiler, DelegateBoilerEvent, BoilerInfoLogger, TestCsharp** như sau:

Lớp **Boiler**:

using System;

namespace TPCenterCsharp

{

class Boiler

{

private int temp;

private int pressure;

public Boiler(int t, int p)

{

temp = t;

pressure = p;

}

public int getTemp()

{

return temp;

}

public int getPressure()

{

return pressure;

}

}

}

Lớp **DelegateBoilerEvent**: đóng vai trò như là event publisher

using System;

namespace TPCenterCsharp

{

class DelegateBoilerEvent

{

public delegate void BoilerLogHandler(string status);

//dinh nghia su kien dua vao delegate tren

public event BoilerLogHandler BoilerEventLog;

public void LogProcess()

{

string remarks = "OK!";

Boiler b = new Boiler(100, 12);

int t = b.getTemp();

int p = b.getPressure();

if(t > 150 || t < 80 || p < 12 || p > 15)

{

remarks = "Can duy tri";

}

OnBoilerEventLog("Thong tin log:\n");

OnBoilerEventLog("Nhiet do: " + t + "\nAp suat: " + p);

OnBoilerEventLog("\nThong bao: " + remarks);

}

protected void OnBoilerEventLog(string message)

{

if (BoilerEventLog != null)

{

BoilerEventLog(message);

}

}

}

}

Lớp **BoilerInfoLogger**:

using System;

using System.IO;

namespace TPCenterCsharp

{

class BoilerInfoLogger

{

FileStream fs;

StreamWriter sw;

public BoilerInfoLogger(string filename)

{

fs = new FileStream(filename, FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

}

public void Logger(string info)

{

sw.WriteLine(info);

}

public void Close()

{

sw.Close();

fs.Close();

}

}

}

Lớp **TestCsharp**: event subscriber

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Logger(string info)

{

Console.WriteLine(info);

}

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa su kien ");

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

BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");

DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();

boilerEvent.BoilerEventLog += new

DelegateBoilerEvent.BoilerLogHandler(Logger);

boilerEvent.BoilerEventLog += new

DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);

boilerEvent.LogProcess();

Console.ReadLine();

Console.ReadKey();

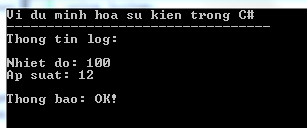
filelog.Close();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Collection

Các lớp Collection là các lớp đặc biệt để lưu giữ và thu hồi dữ liệu. Những lớp này cung cấp sự hỗ trợ cho Stack, Queue, List, và Hash Table. Đa số các lớp Collection triển khai cùng các Interface.

, các lớp Collection phục vụ các mục đích đa dạng, chẳng hạn như cấp phát bộ nhớ động cho các phần tử và truy cập một danh sách các item dựa trên một chỉ mục, … Những lớp này tạo tập hợp các đối tượng của lớp Object, mà là lớp cơ sở cho tất cả kiểu dữ liệu .

Ghi chú: Stack: ngăn xếp, push: thêm nút mới vào đỉnh stack, pop: thao tác lấy 1 phần tử từ đỉnh stack.

## Các lớp Collection và cách sử dụng của chúng

Bảng dưới liệt kê các lớp được sử dụng phổ biến của **System.Collection** namespace. Bạn truy cập link để tìm hiểu chi tiết.

|  |  |
| --- | --- |
| **Lớp** | **Miêu tả và Cách sử dụng** |
| **ArrayList** | Nó biểu diễn một tập hợp được sắp xếp của một đối tượng mà có thể được **lập chỉ mục** cho từng item riêng rẽ.  Về cơ bản, nó là một sự thay thế cho một mảng. Tuy nhiên, không giống như trong mảng, bạn có thể thêm và gỡ bỏ các item từ một list tại một vị trí đã xác định bởi sử dụng một chỉ mục và mảng chính nó có thể tự điều chỉnh kích cỡ một cách tự động. Nó cũng cho phép cấp phát bộ nhớ động, thêm, tìm kiếm và sắp xếp các item trong một list. |
| **Hashtable** | Nó sử dụng một cặp **key-value** để truy cập các phần tử trong collection này.  Một Hash Table được sử dụng khi bạn cần truy cập các phần tử bởi sử dụng key, và bạn có thể nhận diện một giá trị key hữu ích. Mỗi item trong Hash Table có một cặp **key/value**. Key được sử dụng để truy cập các item trong dạng collection này. |
| **SortedList** | Nó sử dụng một **key** cũng như một **index** để truy cập các item trong một list.  Một danh sách đã được sắp xếp là sự tổ hợp của một mảng và một Hash Table. Nó chứa một danh sách các item mà có thể được truy cập bởi sử dụng một key hoặc một chỉ mục. Nếu bạn truy cập item bởi sử dụng một chỉ mục, nó là một ArrayList, và nếu bạn truy cập item bởi sử dụng key, nó là một HashTable. Tập hợp các item luôn luôn được sắp xếp bởi giá trị key |
| **Stack** | Nó biểu diễn một tập hợp **Last-in, First-out** của các đối tượng.  Nó được sử dụng khi bạn cần truy cập các item theo dạng Last-in, First-out. Khi bạn thêm một item vào trong danh sách, nó được gọi là **pushing** và khi bạn gỡ bỏ một item, nó được gọi là **popping**. |
| **Queue** | Nó biểu diễn một tập hợp **First-in, First-out** của các đối tượng.  Nó được sử dụng khi bạn cần truy cập các item theo dạng First-in, First-out. Khi bạn thêm một item vào trong danh sách, nó được gọi là **enqueue** và khi bạn gỡ bỏ một item, nó được gọi là **deque**. |
| **BitArray** | Nó biểu diễn một mảng ở dạng **biểu diễn nhị phân** bởi sử dụng các giá trị 1 và 0.  Nó được sử dụng khi bạn cần lưu giữ các Bit nhưng không biết trước số lượng Bit. Bạn có thể truy cập các item từ BitArray collection bởi sử dụng một chỉ mục là số nguyên, mà bắt đầu từ 0. |

# Generic

**Generic**  cho phép bạn trì hoãn các đặc điểm kỹ thuật (Specification) của kiểu dữ liệu của việc lập trình các phần tử trong một lớp hoặc một phương thức, tới khi nó thực sự được sử dụng trong chương trình. Nói cách khác, Generic cho phép bạn viết một lớp hoặc phương thức mà có thể làm việc với bất kỳ kiểu dữ liệu nào.

Hiểu nôm na, đơn giản generic cũng là một kiểu dữ liệu , nó cũng như là int, float, string, bool, ..., nhưng điểm khác biệt ở đây, generic nó là một kiểu dữ liệu "tự do", nghĩa là nó có thể là kiểu dữ liệu nào cũng được, tùy vào mục đích sử dụng, nó như là 1 kiểu đại diện cho tất cả các kiểu dữ liệu còn lại.

Bạn viết các specification cho lớp và phương thức, với các tham số có thể thay thế cho các kiểu dữ liệu. Khi Compiler bắt gặp một constructor cho một lớp hoặc một lời gọi hàm cho phương thức, nó tạo code để xử lý kiểu dữ liệu cụ thể đó. Dưới đây là ví dụ đơn giản sẽ giúp bạn hiểu khái niệm này.

Bạn tạo hai lớp có tên lần lượt là **TestGeneric, TestCsharp** như sau:

Lớp **TestGeneric**:

using System;

using System.Collections.Generic;

namespace TPCenterCsharp

{

class TestGeneric <T>

{

private T[] array;

public TestGeneric(int size)

{

array = new T[size + 1];

}

public T getItem(int index)

{

return array[index];

}

public void setItem(int index, T value)

{

array[index] = value;

}

}

}

Lớp **TestCsharp**:

using System;

using System.Collections;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du minh hoa Generic ");

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

//khai bao mot mang cac so nguyen

TestGeneric<int> intArray = new TestGeneric<int>(5);

//thiet lap cac gia tri

for (int c = 0; c < 5; c++)

{

intArray.setItem(c, c \* 5);

}

//lay va hien thi cac gia tri

for (int c = 0; c < 5; c++)

{

Console.Write(intArray.getItem(c) + " ");

}

Console.WriteLine();

//khai bao mot mang ky tu

TestGeneric<char> charArray = new TestGeneric<char>(5);

//thiet lap gia tri

for (int c = 0; c < 5; c++)

{

charArray.setItem(c, (char)(c + 97));

}

//lay va hien thi cac gia tri

for (int c = 0; c < 5; c++)

{

Console.Write(charArray.getItem(c) + " ");

}

Console.WriteLine();

Console.ReadKey();

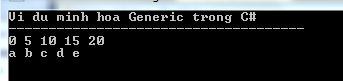
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Đặc điểm của Generic

Generic là một kỹ thuật làm cho chương trình C# của bạn phong phú hơn theo các cách sau:

* Nó giúp bạn tối đa hóa việc tái sử dụng của code, an toàn kiểu, và hiệu năng.
* Bạn có thể tạo các lớp generic collection. Thư viện lớp của .Net Framework chứa một số lớp generic collection trong *System.Collections.Generic* namespace. Bạn có thể sử dụng các lớp generic collection này thay cho các lớp collection trong *System.Collections* namespace.
* Bạn có thể tạo cho riêng mình các Class, Interface, phương thức, Event và Delegate dạng generic.
* Bạn có thể tạo các lớp generic mà cho bạn khả năng truy cập tới các phương thức trên các kiểu dữ liệu cụ thể.
* Bạn có thể lấy thông tin về các kiểu được sử dụng trong một kiểu dữ liệu generic tại runtime bởi các phương thức của Reflection.

Các phương thức Generic

Trong ví dụ trước, chúng ta đã sử dụng một lớp generic, tương tự, chúng ta có thể khai báo một phương thức generic với một tham số kiểu. Ví dụ sau minh họa điều này:

using System;

using System.Collections;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Swap<T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

static void Main(string[] args)

{

Console.WriteLine("Generic ");

Console.WriteLine("-------- Vi du trao doi gia tri ------");

int a, b;

char c, d;

a = 10;

b = 20;

c = 'I';

d = 'V';

// Hien thi cac gia tri truoc khi trao doi:

Console.WriteLine("Cac gia tri int truoc khi goi ham swap: ");

Console.WriteLine("a = {0}, b = {1}", a, b);

Console.WriteLine("Cac gia tri char truoc khi goi ham swap:");

Console.WriteLine("c = {0}, d = {1}", c, d);

//goi ham swap de trao doi gia tri

Swap<int>(ref a, ref b);

Swap<char>(ref c, ref d);

// Hien thi cac gia tri sau khi trao doi:

Console.WriteLine("Cac gia tri int sau khi goi ham swap:");

Console.WriteLine("a = {0}, b = {1}", a, b);

Console.WriteLine("Cac gia tri char sau khi goi ham swap:");

Console.WriteLine("c = {0}, d = {1}", c, d);

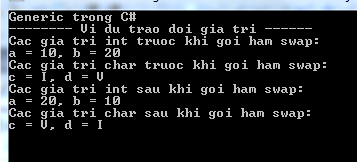
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Generic Delegate

, bạn có thể định nghĩa một Generic Delegate với các tham số kiểu. Ví dụ:

delegate T NumberChanger<T>(T n);

Ví dụ sau đây minh họa cách sử dụng của generic delegate :

using System;

using System.Collections;

delegate T NumberChanger<T>(T n);

namespace TPCenterCsharp

{

class TestCsharp

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

Console.WriteLine("Generic Delegate ");

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

//tao cac doi tuong delegate

NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);

NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

//goi hai phuong thuc boi su dung doi tuong delegate

nc1(25);

Console.WriteLine("1 - Gia tri cua num la: {0}", getNum());

nc2(5);

Console.WriteLine("2 - Gia tri cua num la: {0}", getNum());

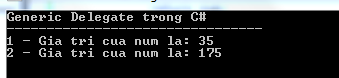
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Phương thức nặc danh (Anonymous Methods)

Chúng ta đã bàn luận rằng Delegate được sử dụng để tham chiếu bất kỳ phương thức nào mà có cùng dấu hiệu như trong Delegate đó. Nói cách khác, bạn có thể gọi một phương thức mà có thể được tham chiếu bởi một Delegate bởi sử dụng đối tượng Delegate đó.

**Phương thức nặc danh (Anonymous Method)**  cung cấp một kỹ thuật để truyền một khối code như là một tham số delegate. Các phương thức nặc danh là các phương thức không có tên, chỉ có thân phương thức.

Bạn không cần xác định kiểu trả về trong một phương thức nặc danh; nó được suy ra từ lệnh return bên trong thân phương thức nặc danh đó.

## Viết một Phương thức nặc danh

Các phương thức nặc danh (Anonymous Method) được khai báo với việc tạo instance của Delegate đó, với một từ khóa **delegate**. Ví dụ:

delegate void NumberChanger(int n);

...

NumberChanger nc = delegate(int x)

{

Console.WriteLine("Phuong thuc nac danh: {0}", x);

};

Khối *Console.WriteLine("Anonymous Method: {0}", x);* là phần thân của phương thức nặc danh.

Delegate có thể được gọi cả với các phương thức nặc danh cũng như các phương thức được đặt tên theo cùng cách, ví dụ: bằng việc truyền các tham số phương thức tới đối tượng Delegate đó.

Ví dụ:

nc(10);

## Ví dụ

Dưới đây là ví dụ minh họa khái niệm trên:

using System;

delegate void NumberChanger(int n);

namespace TPCenterCsharp

{

class TestCsharp

{

static int num = 10;

public static void AddNum(int p)

{

num += p;

Console.WriteLine("Phuong thuc co ten: {0}", num);

}

public static void MultNum(int q)

{

num \*= q;

Console.WriteLine("Phuong thuc co ten: {0}", num);

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

Console.WriteLine("Phuong thuc nac danh ");

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

//tao doi tuong delegate su dung phuong thuc nac danh

NumberChanger nc = delegate(int x)

{

Console.WriteLine("Phuong thuc nac danh: {0}", x);

};

//goi delegate su dung phuong thuc nac danh

nc(10);

//khoi tao delegate su dung phuong thuc co ten

nc = new NumberChanger(AddNum);

//khoi tao delegate su dung phuong thuc co ten

nc(5);

//khoi tao delegate su dung phuong thuc co ten khac

nc = new NumberChanger(MultNum);

//goi delegate su dung phuong thuc co ten

nc(2);

Console.ReadKey();

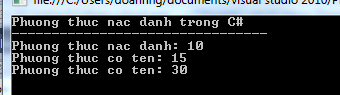
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:

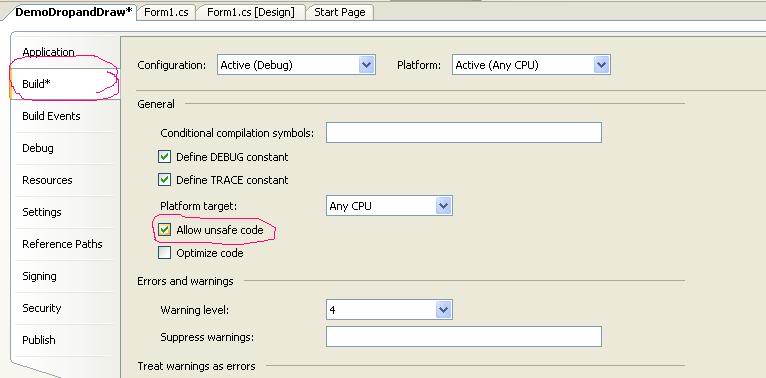


# Unsafe Code

C# cho phép sử dụng các biến con trỏ trong một hàm của khối code khi nó được đánh dấu bởi **unsafe** modifier. Khái niệm **unsafe code** hoặc unmanaged code là một khối code mà sử dụng một biến **con trỏ**.

## Cách compile unsafe code trong Visual Studio 2010

Để biên dịch và chạy các chương trình trong chế độ **/unsafe**, bạn chỉ cần click chuột phải vào **Project**, sau đó chọn **Properties** --> **Build** --> **Allow unsafe code** rồi nhấn tổ hợp phím **Ctrl + S** để lưu các thay đổi như trong hình:



## Con trỏ (Pointer)

Một **con trỏ** là một biến mà có giá trị là địa chỉ của biến khác, ví dụ: địa chỉ trực tiếp của vị trí bộ nhớ. Tương tự như bất kỳ biến hoặc hằng khác , bạn phải khai báo một con trỏ trước khi bạn có thể sử dụng nó để lưu giữ bất kỳ địa chỉ biến nào.

Form chung của một khai báo con trỏ là:

type \*var-name;

Dưới đây là các khai báo con trỏ hợp lệ :

int \*ip; /\* con trỏ tới một số nguyên \*/

double \*dp; /\* con trỏ tới một số double \*/

float \*fp; /\* con trỏ tới một số float \*/

char \*ch /\* con trỏ tới một ký tự \*/

Ví dụ sau minh họa sự sử dụng của con trỏ, sử dụng unsafe modifier :

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static unsafe void Main(string[] args)

{

Console.WriteLine("Con tro ");

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

int var = 20;

int\* p = &var;

Console.WriteLine("Du lieu: {0} ", var);

Console.WriteLine("Dia chi: {0}", (int)p);

Console.ReadKey();

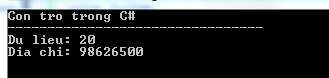
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Thay cho việc khai báo toàn bộ phương thức ở dạng unsafe, bạn cũng có thể khai báo một phần code dạng unsafe.

## Thu hồi giá trị dữ liệu bởi sử dụng con trỏ

Bạn có thể thu hồi dữ liệu đã được lưu giữ tại vị trí được tham chiếu bởi biến con trỏ, sử dụng phương thức **ToString()** . Đây là ví dụ minh họa:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Con tro ");

Console.WriteLine("Unsafe code ");

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

unsafe

{

int var = 20;

int\* p = &var;

Console.WriteLine("Du lieu: {0} ", var);

Console.WriteLine("Du lieu: {0} ", p->ToString());

Console.WriteLine("Dia chi: {0} ", (int)p);

}

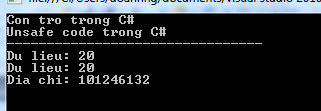
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Truyền Con trỏ như là các Tham số tới phương thức

Bạn có thể truyền một biến con trỏ tới một phương thức dưới dạng các tham số. Đây là ví dụ minh họa:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

public unsafe void swap(int\* p, int\* q)

{

int temp = \*p;

\*p = \*q;

\*q = temp;

}

public unsafe static void Main()

{

Console.WriteLine("Unsafe code ");

Console.WriteLine("Truyen con tro nhu la tham so");

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

TestCsharp p = new TestCsharp();

int var1 = 10;

int var2 = 20;

int\* x = &var1;

int\* y = &var2;

Console.WriteLine("Truoc khi trao doi: var1 = {0}, var2 = {1}", var1, var2);

p.swap(x, y);

Console.WriteLine("Sau khi trao doi: var1 = {0}, var2 = {1}", var1, var2);

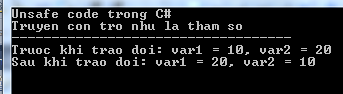
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



## Truy cập các phần tử mảng bởi sử dụng một con trỏ

, một tên mảng và một con trỏ tới cùng kiểu dữ liệu ở dạng dữ liệu mảng, là không cùng kiểu biến. Ví dụ, int \*p và int[] p, là không cùng kiểu. Bạn có thể lượng gia biến con trỏ p bởi vì nó không là cố định trong bộ nhớ, nhưng một địa chỉ mảng là cố định trong bộ nhớ, và bạn không thể lượng gia nó.

Vì thế, nếu bạn cần truy cập một dữ liệu mảng bởi sử dụng một biến con trỏ, như khi chúng ta đã làm trong C hoặc C++ (bạn có thể kiểm tra: [**Con trỏ trong C**](http://vietjack.com/lap_trinh_c/con_tro_trong_c.jsp)), bạn cần cố định con trỏ đó bởi sử dụng từ khóa **fixed** .

Sau đây là ví dụ minh họa:

using System;

namespace TPCenterCsharp

{

class TestCsharp

{

public unsafe static void Main()

{

Console.WriteLine("Unsafe code ");

Console.WriteLine("Truy cap cac phan tu mang boi su dung con tro");

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

int[] list = {10, 100, 200};

fixed (int\* ptr = list)

/\* mang dia chi trong con tro \*/

for (int i = 0; i < 3; i++)

{

Console.WriteLine("Dia chi cua list[{0}]={1}", i, (int)(ptr + i));

Console.WriteLine("Gia tri cua list[{0}]={1}", i, \*(ptr + i));

}

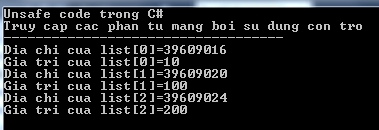
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



# Multithreading

**[Thread ]** Một **thread** được định nghĩa như là một đường thực thi (execution path) của một chương trình. Mỗi Thread định nghĩa một dòng điều khiển duy nhất. Nếu application của bạn gồm các hoạt động phức tạp và tốn thời gian, thì nó thường là rất hữu ích khi thiết lập các execution path hoặc Thread, với mỗi Thread thực hiện một công việc cụ thể.

Các Thread là các **tiến trình nhẹ**. Một ví dụ phổ biến của sự sử dụng Thread là sự triển khai lập trình tương tranh (concurrent programming) bởi các hệ điều hành hiện đại. Sử dụng các Thread tiếp kiệm sự hao phí của CPU cycle và tăng hiệu quả của một application.

Tới chương này, chúng ta đã viết các chương trình mà một Thread đơn chạy như là một tiến trình đơn, đó là trình chạy thể hiện của application. Tuy nhiên, theo cách này, application có thể thực hiện một công việc tại một thời điểm. Để làm nó thực thi nhiều hơn một tác vụ tại một thời điểm, nó có thể được phân chia thành các Thread nhỏ hơn.

Vòng đời của Thread

Vòng đời của một Thread bắt đầu khi một đối tượng của lớp System.Threading.Thread được tạo và kết thúc khi Thread đó được kết thúc hoặc hoàn thành việc thực thi.

Dưới đây là các trạng thái đa dạng trong vòng đời của một Thread :

* **Unstarted State**: Nó là tình huống khi instance của Thread được tạo, nhưng phương thức Start chưa được gọi.
* **Ready State**: Nó là tình huống khi Thread đó sẵn sàng để chạy và đợi CPU cycle.
* **Not Runnable State**: Một Thread là không thể thực thi (not executable), khi:
  + Phương thức Sleep đã được gọi.
  + Phương thức Wait đã được gọi.
  + Bị ngăn chặn bởi hoạt động I/O.
* **Dead State**: Nó là tình huống khi Thread hoàn thành sự thực thi hoặc bị hủy bỏ.

Main Thread

, lớp **System.Threading.Thread** được sử dụng để làm việc với các Thread. Nó cho phép tạo và truy cập các Thread riêng biệt trong một Multithreaded Application. Thread đầu tiên để được thực thi trong một tiến trình được gọi là **Main** Thread .

Khi một chương trình C# bắt đầu thực thi, Main Thread được tự động tạo ra. Các Thread, được tạo bởi sử dụng lớp **Thread**, được gọi các Thread con của Main Thread. Bạn có thể truy cập một Thread bởi sử dụng thuộc tính **CurrentThread** của lớp Thread.

Dưới đây là chương trình ví dụ minh họa cho sự thực thi Main Thread :

using System;

using System.Threading;

namespace TPCenterCsharp

{

class TestCsharp

{

public static void Main()

{

Console.WriteLine("Da luong ");

Console.WriteLine("Vi du minh hoa Main Thread");

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

Thread th = Thread.CurrentThread;

th.Name = "MainThread";

Console.WriteLine("Day la {0}", th.Name);

Console.ReadKey();

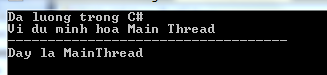
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Thuộc tính và Phương thức của lớp Thread

Bảng dưới liệt kê một số **thuộc tính** được sử dụng phổ biến nhất của lớp **Thread** :

|  |  |
| --- | --- |
| **Thuộc tính** | **Miêu tả** |
| CurrentContext | Lấy ngữ cảnh (context) hiện tại mà trong đó Thread đang thực thi |
| CurrentCulture | Lấy hoặc thiết lập culture gồm language, date, time, currency, … cho Thread hiện tại |
| CurrentPrinciple | Lấy hoặc thiết lập nguyên lý hiện tại của Thread |
| CurrentThread | Lấy Thread đang chạy hiện tại |
| CurrentUICulture | Lấy hoặc thiết lập culture hiện tại được sử dụng bởi Resource Manager để tìm kiếm cho Resource cụ thể tại runtime |
| ExecutionContext | Lấy một đối tượng ExecutionContext mà chứa thông tin về các context đa dạng của Thread hiện tại |
| IsAlive | Lấy một giá trị chỉ trạng thái thực thi của Thread hiện tại |
| IsBackground | Lấy hoặc thiết lập một giá trị chỉ rằng có hay không một Thread là Background Thread |
| IsThreadPoolThread | Lấy một giá trị chỉ rằng có hay không một Thread là của Managed Thread Pool |
| ManagedThreadId | Lấy một định danh duy nhất cho Managed Thread hiện tại |
| Name | Lấy hoặc thiết lập tên của Thread |
| Priority | Lấy hoặc thiết lập một giá trị chỉ quyền ưu tiên của một Thread |
| ThreadState | Lấy một giá trị chứa các trạng thái của Thread hiện tại |

Bảng này liệt kê các **phương thức** được sử dụng phổ biến nhất của lớp **Thread** :

|  |  |
| --- | --- |
| **STT** | **Phương thức** |
| 1 | **public void Abort()**  Tạo một ThreadAbortException trong Thread mà trên đó nó được triệu hồi, để bắt đầu tiến trình kết thúc Thread đó. Gọi phương thức này thường kết thúc Thread |
| 2 | **public static LocalDataStoreSlot AllocateDataSlot()**  Cấp phát một Unnamed Data Slot cho tất cả Thread. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 3 | **public static LocalDataStoreSlot AllocateNamedDataSlot(string name)**  Cấp phát một Named Data Slot cho tất cả Thread. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 4 | **public static void BeginCriticalRegion()**  Thông báo cho một host rằng sự thực thi là chuẩn bị đi vào một khu vực code, mà trong đó các ảnh hưởng của việc hủy bỏ một Thread hoặc các Exception không được xử lý có thể gây nguy hại tới các tác vụ khác trong miền ứng dụng |
| 5 | **public static void BeginThreadAffinity()**  Thông báo cho một Host rằng Managed code là chuẩn bị thực thi các chỉ lệnh mà phụ thuộc vào tính đồng nhất của Physical operating system thread hiện tại |
| 6 | **public static void EndCriticalRegion()**  Thông báo cho một host rằng sự thực thi là chuẩn bị đi vào một khu vực code, mà trong đó các ảnh hưởng của hủy bỏ một Thread hoặc các Exception không được xử lý bị hạn chế tới tác vụ hiện tại |
| 7 | **public static void EndThreadAffinity()**  Thông báo cho một Host rằng Managed code đã kết thúc việc thực thi các chỉ lệnh mà phụ thuộc vào tính đồng nhất của Physical Operating System Thread hiện tại |
| 8 | **public static void FreeNamedDataSlot(string name)**  Loại bỏ sự liên kết giữa một name và một slot, cho tất cả Thread trong tiến trình. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 9 | **public static Object GetData(LocalDataStoreSlot slot)**  Thu hồi giá trị từ slot đã xác định trên Thread hiện tại, bên trong miền hiện tại của Thread hiện tại. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 10 | **public static AppDomain GetDomain()**  Trả về miền hiện tại trong đó Thread đang chạy |
| 11 | **public static AppDomain GetDomain()**  Trả về một định danh miền ứng dụng duy nhất |
| 12 | **public static LocalDataStoreSlot GetNamedDataSlot(string name)**  Tìm kiếm một Named Data Slot. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 13 | **public void Interrupt()**  Interrupt (ngắt) một Thread mà trong trạng thái WaitSleepJoin |
| 14 | **public void Join()**  Chặn Thread đang gọi tới khi một Thread kết thúc, trong khi tiếp tục thực hiện COM và SendMessage Pumping. Phương thức này có các mẫu được nạp chồng khác nhau |
| 15 | **public static void MemoryBarrier()**  Đồng bộ truy cập bộ nhớ như sau: Prosessor đang thực thi Thread hiện tại không thể sắp xếp lại các chỉ lệnh theo một cách để mà quyền truy cập bộ nhớ tới lời gọi đến MemoryBarrier thực thi sau khi các truy cập bộ nhớ mà theo sau lời gọi đó đến MemoryBarrier |
| 16 | **public static void ResetAbort()**  Hủy một Abort được yêu cầu cho Thread hiện tại |
| 17 | **public static void SetData(LocalDataStoreSlot slot, Object data)**  Thiết lập dữ liệu trong slot đã cho trên Thread đang chạy hiện tại, cho miền hiện tại của Thread đó. Để tăng hiệu suất, sử dụng các Field mà được đánh dấu với attribute là ThreadStaticAttribute để thay thế |
| 18 | **public void Start()**  Bắt đầu một Thread |
| 19 | **public static void Sleep(int millisecondsTimeout)**  Làm Thread dừng trong một khoảng thời gian |
| 20 | **public static void SpinWait(int iterations)**  Làm một Thread đợi một khoảng thời gian đã được xác định trong tham số iterations |
| 21 | **public static byte VolatileRead(ref byte address)**  **public static double VolatileRead(ref double address)**  **public static int VolatileRead(ref int address)**  **public static Object VolatileRead(ref Object address)**  Đọc giá trị của một Field. Giá trị này là được viết mới nhất bởi bất kỳ Prosessor nào trong một máy tính, không quan tâm đến số lượng Prosessor hoặc trạng thái của Prosessor Cache. Phương thức này có các mẫu được nạp chồng khác nhau. Đó là các form ở trên |
| 22 | **public static void VolatileWrite(ref byte address,byte value)**  **public static void VolatileWrite(ref double address, double value)**  **public static void VolatileWrite(ref int address, int value)**  **public static void VolatileWrite(ref Object address, Object value)**  Ghi một giá trị tới một Field ngay lập tức, để mà giá trị này là nhìn thấy cho tất cả Processor trong máy tính. Phương thức này có các mẫu được nạp chồng khác nhau. Đó là các form ở trên |
| 23 | **public static bool Yield()**  Làm Thread đang gọi chuyển sự thực thi cho Thread khác mà đã sẵn sàng để chạy trên Processor hiện tại. Hệ điều hành chọn Thread để chuyển tới |

Tạo Thread

, các Thread được tạo bằng việc kế thừa lớp Thread. Sau đó, Lớp Thread được kế thừa gọi phương thức **Start()** để bắt đầu sự thực thi của Thread con.

Sau đây là ví dụ minh họa việc tạo Thread :

using System;

using System.Threading;

namespace TPCenterCsharp

{

class TestCsharp

{

public static void CallToChildThread()

{

Console.WriteLine("Thread con bat dau!!!");

}

static void Main(string[] args)

{

Console.WriteLine("Da luong ");

Console.WriteLine("Vi du minh hoa cach tao Thread");

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

ThreadStart childref = new ThreadStart(CallToChildThread);

Console.WriteLine("Trong Main Thread: tao thread con.");

Thread childThread = new Thread(childref);

childThread.Start();

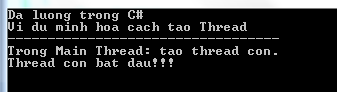
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Quản lý Thread

Lớp Thread cung cấp các phương thức đa dạng để quản lý các Thread.

Ví dụ sau minh họa cách sử dụng phương thức **sleep()** để làm một Thread dừng trong một khoảng thời gian cụ thể.

using System;

using System.Threading;

namespace TPCenterCsharp

{

class TestCsharp

{

public static void CallToChildThread()

{

Console.WriteLine("Bat dau Thread con!!!");

// Thread nay dung khoang 5000 milisecond

int sleepfor = 5000;

Console.WriteLine("Thread con dung trong khoang {0} giay", sleepfor / 1000);

Thread.Sleep(sleepfor);

Console.WriteLine("Thread con phuc hoi!!!");

}

static void Main(string[] args)

{

Console.WriteLine("Da luong ");

Console.WriteLine("Vi du minh hoa quan ly Thread");

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

ThreadStart childref = new ThreadStart(CallToChildThread);

Console.WriteLine("Trong Main Thread: tao Thread con.");

Thread childThread = new Thread(childref);

childThread.Start();

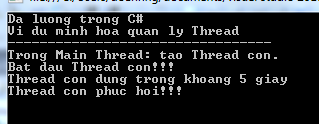
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Hủy Thread

Phương thức **Abort()** được sử dụng để hủy các Thread .

Trong thời gian runtime, chương trình hủy bỏ Thread bằng việc ném một *ThreadAbortException*. Exception này không thể được bắt, điều khiển được gửi tới khối **finally**, nếu là không.

Dưới đây là chương trình minh họa việc sử dụng phương thức Abort() để hủy bỏ các Thread :

using System;

using System.Threading;

namespace TPCenterCsharp

{

class TestCsharp

{

public static void CallToChildThread()

{

try

{

Console.WriteLine("Bat dau Thread con!!!");

// gia su chung ta dem tu 0 toi 10

for (int counter = 0; counter <= 10; counter++)

{

Thread.Sleep(500); //dung trong khoang 5 giay

Console.WriteLine(counter);

}

Console.WriteLine("Thread con hoan thanh.");

}

catch (ThreadAbortException e)

{

Console.WriteLine("Thread Abort Exception!!!");

}

finally

{

Console.WriteLine("Khong the bat Thread Exception!!!");

}

}

static void Main(string[] args)

{

Console.WriteLine("Da luong ");

Console.WriteLine("Vi du minh hoa huy Thread");

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

ThreadStart childref = new ThreadStart(CallToChildThread);

Console.WriteLine("Trong Main Thread: tao Thread con.");

Thread childThread = new Thread(childref);

childThread.Start();

//dung Main Thread trong khoang 2 giay

Thread.Sleep(2000);

//bay gio huy thread con

Console.WriteLine("Trong Main Thread: huy Thread con.");

childThread.Abort();

Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:

# Đa luồng trong C#

# Nguồn

# Tổng kết