

Mục lục

Mục lục	1
Mở đầu.....	3
Chương 1: Kiến trúc .NET	4
1. 1 Quan hệ giữa C# và.NET	4
1. 2 CLR (Common Language Runtime)	4
1. 3 Giới thiệu IL (Intermediate Language)	5
1. 4 Thư viện (Assembly)	5
1. 5 Các lớp trong .NET	5
1. 6 Tạo ứng dụng .NET sử dụng C#.....	6
1. 7 Vai trò của .NET trong kiến trúc .NET Enterprise.....	6
Chương 2: Căn bản C#	7
2. 1 Viết chương trình C# đầu tiên	7
2. 2 Biến.....	11
2. 3 Kiểu dữ liệu cơ bản.....	12
2. 4 Điều khiển luồng	14
2. 5 Kiểu liệt kê.....	19
2. 6 Mảng.....	21
2. 7 Không gian tên (Namespace)	22
2. 8 Phương thức Main()	23
2. 9 Biên dịch nhiều tập tin C#	23
2. 10 Xuất nhập qua Console.....	24
2. 11 Sử dụng chú thích	25
2. 12 Chỉ dẫn tiền xử lý trong C#	25
Chương 3: Đối tượng và kiểu	27
3. 1 Lớp và cấu trúc	27
3. 2 Thành viên của lớp	29
3. 3 Cấu trúc (Struct)	46
3. 4 Lớp Object	53
Chương 4: Sự kế thừa	56
4. 1 Các kiểu kế thừa	56
4. 3 Từ khóa hỗ trợ	59

4. 4 Đa hình (polymorphism)	60
Chương 5: Toán tử và chuyển kiểu.....	72
5. 1 Toán tử.....	72
5. 3 Quá tải toán tử	74
5. 4 Chuyển kiểu do người dùng định nghĩa	79
Chương 6: Sự ủy nhiệm, sự kiện và quản lý lỗi.....	81
6. 1 Sự ủy nhiệm (delegate).....	81
6. 2 Sự kiện (Event).....	82
6. 3 Quản lý lỗi và biệt lệ	85
Chapter 7: Quản lý bộ nhớ và con trỏ.....	89
7. 1 Quản lý bộ nhớ	89
7. 2 Giải phóng tài nguyên	90
7. 3 Mã không an toàn	93
Chương 8: Chuỗi, biểu thức quy tắc và tập hợp	97
8. 1 System.String.....	97
8. 2 Biểu thức quy tắc	98
8. 3 Nhóm các đối tượng	100
Chương 9: Reflection.....	104
9. 1 Thuộc tính (attribute) tùy chọn.....	104
9. 2 Reflection.....	106
Hướng dẫn phần thực hành.....	110
Tài liệu tham khảo.....	110

Mở đầu

Lập trình hướng đối tượng (OOP) đóng một vai trò quan trọng trong việc xây dựng và phát triển ứng dụng. Đặc biệt trong các ngôn ngữ lập trình thế hệ thứ 4 (như java hay c#) hầu như được xây dựng là những ngôn ngữ thuận đối tượng nhằm hỗ trợ những nguyên lý căn bản trong lập trình hướng đối tượng cũng như các tính năng nâng cao dựa trên OOP giúp cho việc xây dựng và phát triển ứng dụng trên OOP dễ dàng và nhanh chóng hơn. Do đó việc tiếp cận và nắm vững các nguyên lý lập trình hướng đối tượng rất quan trọng đối với sinh viên cho việc sử dụng và ứng dụng nó cho các môn học liên quan đến lập trình và các môn học chuyên ngành ở các học kì tiếp theo.

Mục tiêu của môn học:

- Ôn tập lại các vấn đề về kỹ thuật lập trình, cách thức phát triển ứng dụng đơn giản trên C#.
- Cung cấp cho sinh viên tiếp cận và sử dụng ngôn ngữ lập trình C#.
- Cung cấp cho sinh viên kiến thức về lập trình hướng đối tượng trên ngôn ngữ lập trình C# bao gồm tính đóng gói, kế thừa, đa hình, giao tiếp, attribute, reflection.
- Cung cấp các kiến thức về xử lý và thao tác dữ liệu trên tập tin văn bản và nhị phân, XML.
- Cung cấp các kiến thức về sử dụng các cấu trúc dữ liệu được dựng sẵn trên .Net trong quá trình phát triển ứng dụng như Stack, Queue, ArrayList, HashTable.
- Giới thiệu việc xây dựng và phát triển ứng dụng trên môi trường môi trường .Net. Cung cấp cho sinh viên tiếp cận và làm quen với môi trường phát triển ứng dụng dựa trên Visual Studio 2005.

Chương 1: Kiến trúc .NET

Microsoft Visual C# là một ngôn ngữ mạnh mẽ nhưng đơn giản chủ yếu hướng đến các nhà phát triển xây dựng ứng dụng trên nền tảng .NET của Microsoft. C# kế thừa những đặc trưng tốt nhất của ngôn ngữ C++ và Microsoft Visual Basic, và loại bỏ đi một số đặc trưng không thống nhất và lạc hậu với mục tiêu tạo ra một ngôn ngữ rõ ràng và logic hơn. Sự kì vọng của C# đã được bổ sung một số đặc trưng mới quan trọng bao gồm Generic, cơ chế lặp và phương thức ẩn tên... Môi trường phát triển cung cấp bởi Visual Studio 2005 làm cho những đặc trưng này trở nên dễ sử dụng và nâng cao năng suất cho các nhà phát triển ứng dụng.

Mục đích của chương:

- Giới thiệu ngôn ngữ C#.
- Giới thiệu các thành phần quan trọng của nền tảng .Net.
- So sánh C# với ngôn ngữ lập trình C và một số các ngôn ngữ lập trình khác.

1. 1 Quan hệ giữa C# và.NET

C# là một ngôn ngữ lập trình mới và có các đặc trưng:

- Nó được thiết kế riêng để dùng cho nền tảng .NET.
- Nó là một ngôn ngữ thuần đối tượng được thiết kế dựa trên kinh nghiệm của các ngôn ngữ hướng đối tượng khác.
- C# là một ngôn ngữ độc lập. Nó được thiết kế để có thể sinh ra mã đích trong môi trường .NET, nó không phải là một phần của .NET bởi vậy có một vài đặc trưng được hỗ trợ bởi .NET nhưng C# không hỗ trợ.

1. 2 CLR (Common Language Runtime)

Điểm tập trung của nền tảng .NET là môi trường thực hiện việc thực thi ứng dụng được gọi là CLR (Common Language Runtime-CLR).

Trong .NET chương trình không biên dịch thành tập tin thực thi, chúng được biên dịch theo hai bước:

- Biên dịch mã nguồn thành IL (Intermediate Language).
- Dùng CLR để biên dịch IL thành mã máy theo từng nền tảng thích hợp.

Việc thực hiện như trên cung cấp nhiều thuận lợi cho .NET như:

- Độc lập nền tảng và phần cứng.
- Nâng cao hiệu suất.
- Giúp cho các ngôn ngữ phát triển trên các ngôn ngữ lập trình khác nhau có thể tương tác với nhau.

1. 3 Giới thiệu IL (Intermediate Language)

IL hoạt động như là bản chất của nền tảng .NET. Mã C# sẽ luôn được dịch sang IL trước khi nó được thực thi. bất kì ngôn ngữ nào hướng .NET cũng sẽ hỗ trợ các đặc tính chính của IL.

Sau đây là những đặc tính chính của IL:

- Hướng đối tượng và dùng giao tiếp.
- Sự tách biệt giữa kiểu giá trị và kiểu tham chiếu.
- Định kiểu mạnh.
- Quản lý thông qua các ngoại lệ.
- Sử dụng các thuộc tính.

1. 4 Thư viện (Assembly)

Một assembly là một tập tin chứa mã đã được biên dịch sang .NET. Nó có thể chứa trong nhiều tập tin. Nếu một assembly được lưu trong nhiều tập tin, thì sẽ có một tập tin chính chứa các con trỏ và các mô tả về các tập tin khác của assembly. Cấu trúc assembly được dùng chung cho cả mã thực thi và mã thư viện. Sự khác biệt duy nhất là assembly thực thi có chứa hàm main trong khi assembly thư viện thì không có.

Một điểm quan trọng trong các assembly là chúng chứa các siêu dữ liệu (metadata) dùng để mô tả các kiểu và phương thức được định nghĩa tương ứng trong mã. Một assembly cũng chứa siêu dữ liệu dùng để mô tả chính assembly đó. Siêu dữ liệu chứa trong một vùng được gọi là tập tin mô tả (**manifest**), nó cho phép kiểm tra phiên bản và tình trạng của assembly.

Với việc assembly chứa siêu dữ liệu, nó cho phép chương trình, ứng dụng hay các assembly khác có thể gọi mã trong một assembly mà không cần tham chiếu đến Registry, hoặc một dữ liệu nguồn khác.

1. 5 Các lớp trong .NET

Một trong những lợi ích lớn nhất của viết mã đó là việc sử dụng các thư viện lớp cơ sở sẵn có của .NET. Thư viện lớp cơ sở của .NET là một tập hợp rất nhiều các lớp mã đã được phát triển bởi Microsoft, những lớp này cho phép thao tác rất nhiều các tác vụ sẵn có trong Windows. Chúng ta có thể tạo các lớp của mình từ các lớp có sẵn trong thư viện lớp cơ sở của .NET thông qua sự kế thừa.

Thư viện lớp cơ sở .NET là kết hợp tính đơn giản của các thư viện Visual Basic và Java với hầu hết các đặc tính trong các thư viện hàm API. Có nhiều đặc tính lạ, ít sử dụng của Windows không được cung cấp trong các lớp của thư viện .NET. Những đặc tính thông dụng đều đã được hỗ trợ đầy đủ trong thư viện lớp của .NET. Nếu chúng ta muốn gọi một hàm API trong .NET, chúng ta thực hiện cơ chế "platform-invoke", cơ chế này luôn bảo đảm tính đúng đắn của kiểu dữ liệu khi gọi và hỗ trợ cho cả C#, C++, và VB.NET. Thao tác gọi này không khó hơn việc gọi trực tiếp API từ mã C++.

1. 6 Tạo ứng dụng .NET sử dụng C#

C# có thể tạo các ứng dụng dòng lệnh (console) cũng như các ứng dụng thuận văn bản chạy trên DOS hay Window. Tất nhiên, chúng ta có thể dùng C# để tạo các ứng dụng dùng cho các công nghệ tương thích với .NET.

Các ứng dụng có thể viết trên C#:

- Ứng dụng ASP.NET.
- Ứng dụng WinForm.
- Các dịch vụ dựa trên Windows.

1. 7 Vai trò của .NET trong kiến trúc .NET Enterprise

C# yêu cầu khi chạy phải có “.NET runtime”, do đó bắt buộc chúng ta phải cài đặt .Net runtime trước khi muốn chạy các ứng dụng được phát triển trên .Net. Tuy nhiên, trong một số phiên bản mới của hệ điều hành Windows, .Net đã được cài đặt mặc định. Thực vậy, C# được coi như là một cơ hội nổi bật cho các tổ chức để có thể tạo những ứng dụng mạnh mẽ, những ứng dụng client-server theo kiến trúc N-lớp.

Khi kết nối dữ liệu thông qua ADO.NET, C# có khả năng truy cập tới các cơ sở dữ liệu tổng quát và nhanh chóng như cơ sở dữ liệu SQL Server và Oracle. Dữ liệu trả về từ các thao tác dữ liệu thông qua DataSet giúp dễ dàng thao tác thông qua các đối tượng của ADO.NET. Kết nối dữ liệu tự động trả về kiểu XML giúp cho việc truyền thông trên mạng dễ dàng.

Chương 2: Căn bản C#

Mục đích của chương:

- Khai báo biến.
- Khởi tạo và phạm vi hoạt động của biến.
- Các kiểu dữ liệu cơ bản.
- Cách sử dụng các vòng lặp và câu lệnh.
- Gọi, hiển thị lớp và phương thức.
- Cách sử dụng mảng.
- Toán tử.
- An toàn kiểu và cách để chuyển kiểu dữ liệu.
- Kiểu liệt kê (enum).
- Không gian tên (namespace).
- Hàm Main().
- Biên dịch trong C#.
- Xuất nhập dùng System.Console.
- Sử dụng chú thích trong C#.
- Các định danh và từ khoá trong C#.

2. 1 Viết chương trình C# đầu tiên

Đầu tiên chúng ta viết một chương trình ứng dụng “Hello World” đơn giản sử dụng C#:

```
class HelloWorld
{
    static void Main( )
    {
        System.Console.WriteLine("Chuong Trinh Dau Tien");
        System.Console.ReadLine();

    }
}
```

Ứng dụng dòng lệnh là ứng dụng không có giao diện người dùng. Việc xuất nhập thông qua dòng lệnh chuẩn. Phương thức Main() trong ví dụ “Hello World” viết chuỗi “Chuong Trinh Dau Tien” lên màn hình. Màn hình được quản lý bởi một đối tượng tên Console. Đối tượng này có một phương thức WriteLine(), nhận một chuỗi và xuất chúng ra thiết bị xuất chuẩn (màn hình).

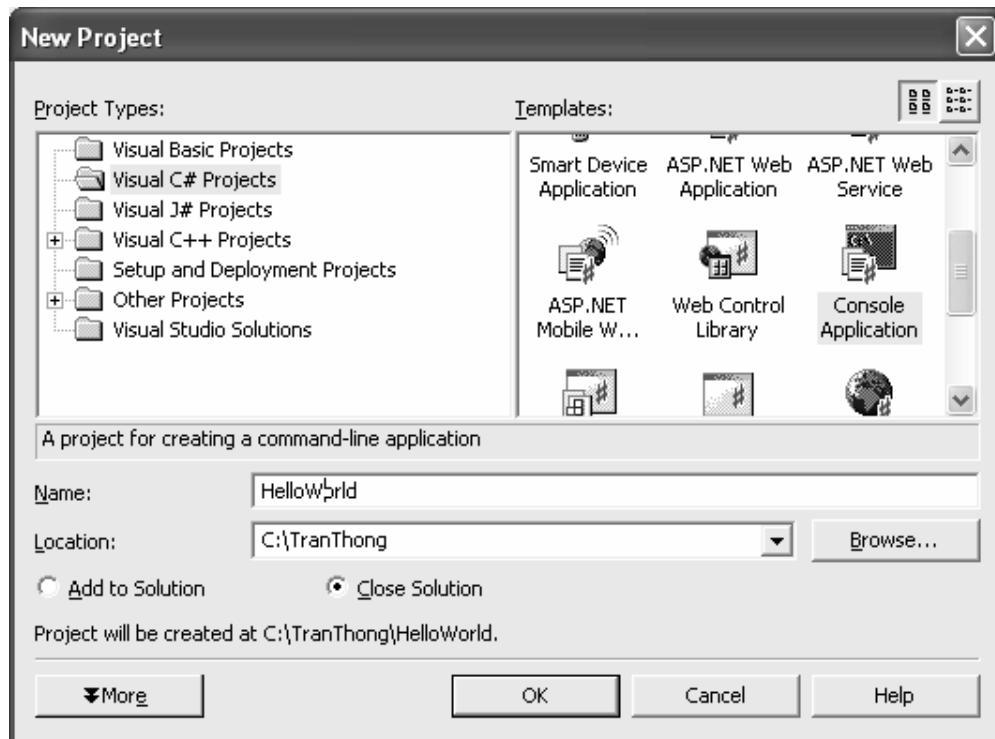
Cách chạy chương trình “Hello world”

Để thực hiện được chương trình chúng ta sử dụng “Visual Studio.NET Intergated Development Environment (IDE)” trong công cụ “Visual Studio.NET IDE”. Chúng cung cấp những công cụ rất mạnh cho việc dò lỗi và hỗ trợ một số tính năng khác.

Soạn thảo chương trình “Hello Wolrd”

- Chạy chương trình IDE. Chọn Visual Studio.NET từ thực đơn Start
- Chọn File→New→Project. Chọn kiểu dự án là Visual C# Project và dạng Console Application. Chúng ta có thể nhập vào tên dự án và đường dẫn để lưu trữ dự án. Sau khi chọn nút OK, một cửa sổ mới sẽ xuất hiện như hình 2.1

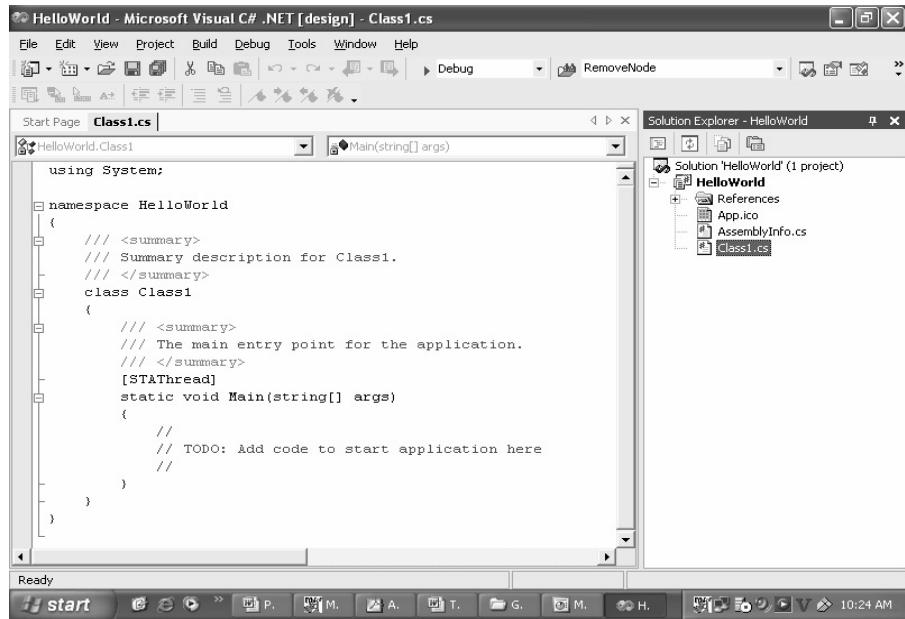
Hình 2.1:Tạo ứng dụng dòng lệnh trong Visual Studio.NET



- Sau đó đưa lệnh sau vào trong hàm Main()

```
System.Console.WriteLine("Chuong Trinh Dau Tien");
```

Hình 2.2: Cửa sổ soạn thảo cho một dự án mới



Biên dịch và chạy chương trình “Hello Wolrd”

Có nhiều cách để biên dịch và chạy chương trình trong Visual Studio.NET

- Chọn Ctrl+Shift+B hay Build→build từ thực đơn.
- Chọn nút Build như trong hình 2.3.

Hình 2.3: Nút build



Để chạy chương trình mà không thực hiện dò lỗi:

- Nhấn Ctrl + F5 hay Debug→Start Without Debugging từ thực đơn.
- Chọn nút Start Without Debugging như trong hình 2.4

Hình 2.4: Nút Start Without Debugging



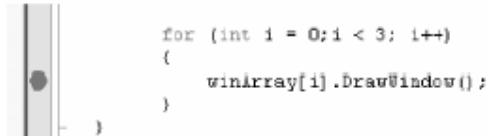
Sử dụng công cụ dò lỗi của Visual Studio.NET

3 kỹ năng quan trọng khi dò lỗi:

- Bằng cách nào đặt các điểm dừng (breakpoint) và chạy các điểm dừng như thế nào?
- Bằng cách nào chạy từng bước qua các lời gọi phương thức.
- Bằng cách nào kiểm tra và thay đổi giá trị của biến, dữ liệu thành viên của lớp.

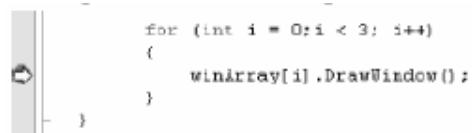
- Dò lỗi có thể được thực hiện theo nhiều cách. Thông thường qua thực đơn. Đơn giản nhất là đặt điểm dừng bên trước trái như trong hình 2.5

Hình 2.5: Một điểm dừng



Để chạy Debug chúng ta có thể nhấn F5 và sau đó chương trình sẽ chạy đến điểm dừng như trong hình 2.6.

Hình 2.6: Chọn điểm dừng



Bằng cách đặt con chuột vào vị trí các biến chúng ta có thể thấy được giá trị hiện tại của biến như trong hình 2.7.

Hình 2.7: Hiển thị một giá trị

```
for (int i = 0; i < 3; i++)
{
    winarray[i].DrawWindow();
}
```

Trình dò lỗi của Visual Studio.NET cũng cung cấp một số cửa sổ hữu dụng khác để dò lỗi như là cửa sổ Local để dò các biến cục bộ như trong hình 2.8

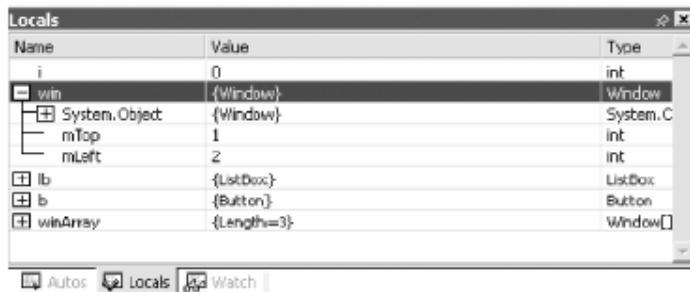
Hình 2.8: Cửa sổ Local

Locals		
Name	Type	Value
i	int	0
win	Window	{Window}
lb	ListBox	{ListBox}
b	Button	{Button}
winArray	Window[]	{Length=3}

Buttons at the bottom: Autos, Locals (highlighted), Watch.

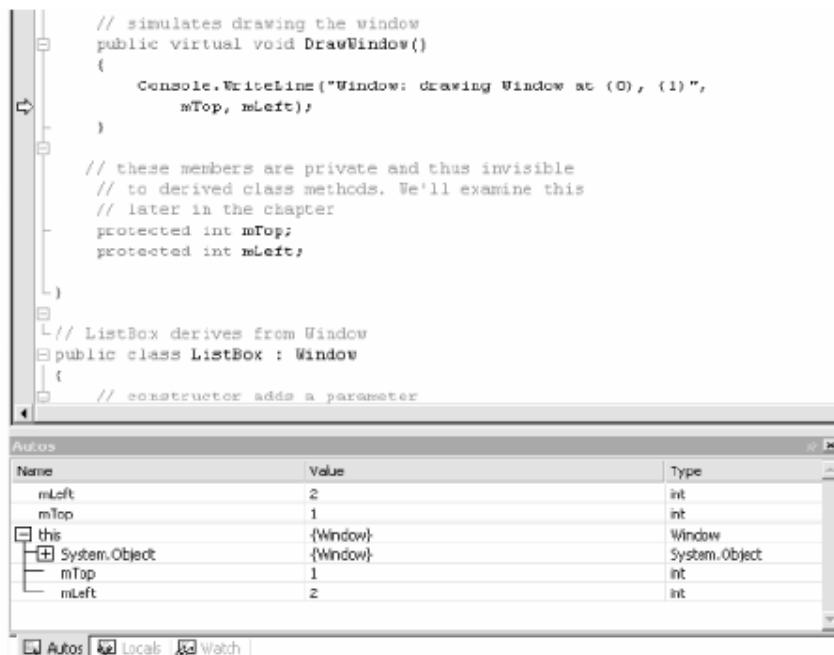
Chúng ta có thể mở rộng cửa sổ để xem chi tiết thông tin biến như trong hình 2.9.

Hình 2.9: Mở rộng cửa sổ Local



Chúng ta có thể lặp qua các phương thức bằng các nhấp F11. Ví dụ lặp qua phương thức DrawWindow() của lớp WindowClass như trong hình 2.10.

Hình 2.10: Lặp qua một phương thức



2. 2 Biến

Một biến dùng để lưu trữ giá trị của một kiểu dữ liệu nào đó.

Cú pháp C# sau đây để khai báo một biến:

[bổ_tù] kiểu_dữ_liệu định_danh;

Với **bổ_tù** là một trong những từ khoá: public, private, protected, ... còn **kiểu_dữ_liệu** là các kiểu dữ liệu như số nguyên (int), thực (float)... và **định_danh** là tên biến.

Ví dụ dưới đây một biến tên **i** kiểu nguyên và có thể được truy cập bởi bất cứ hàm nào.

```
public int i;
```

Ta có thể gán cho biến một giá trị bằng toán tử **"=**.

```
i = 10;
```

Ta cũng có thể khai báo biến và khởi tạo giá trị cho biến như sau:

```
int i = 10;
```

Chúng ta có thể khai báo nhiều biến có cùng kiểu dữ liệu như sau:

```
int x = 10, y = 20;
int x = 10;
bool y = true; // khai báo trên đúng
int x = 10, bool = true // khai báo trên có lỗi
```

Một hằng (constant) là một biến nhưng giá trị không thể thay đổi trong suốt thời gian thực thi chương trình. Đôi lúc ta cũng cần có những giá trị bao giờ cũng bất biến.

```
const int a = 100; // giá trị này không thể bị thay đổi
```

Hằng có những đặc điểm sau:

- Hằng bắt buộc phải được gán giá trị lúc khai báo. Một khi đã được khởi gán thì không thể viết đè lên.
- Trị của hằng có thể được tính toán vào lúc biên dịch. Do đó không thể gán một hằng từ một trị của một biến. Nếu muốn làm thế thì phải sử dụng “read-only field”.
- Hằng bao giờ cũng static, tuy nhiên ta không thể đưa từ khoá static vào khi khai báo hằng.

Có ba thuận lợi khi sử dụng hằng:

- Hằng làm cho chương trình đọc dễ dàng hơn, bằng cách thay thế những con số vô cảm bởi những tên mang đầy ý nghĩa hơn.
- Hằng làm cho chương trình dễ sửa hơn.
- Hằng làm cho việc tránh lỗi dễ dàng hơn, nếu bạn gán một trị khác cho một hằng đâu đó trong chương trình sau khi bạn đã gán giá trị cho hằng, thì trình biên dịch sẽ thông báo lỗi.

2. 3 Kiểu dữ liệu cơ bản

C# là một ngôn ngữ được kiểm soát chặt chẽ về mặt kiểu dữ liệu, ngoài ra C# còn chia các kiểu dữ liệu thành hai loại: **kiểu giá trị** (value type) và **kiểu tham chiếu** (reference type). Nghĩa là trên một chương trình C# dữ liệu được lưu trữ một hoặc hai nơi tùy theo đặc thù của kiểu dữ liệu.

- Chỗ thứ nhất là **stack**: một vùng bộ nhớ dành lưu trữ dữ liệu với chiều dài cố định, chẳng hạn số nguyên chiếm dụng 4 bytes. Mỗi chương trình khi đang thực thi đều được cấp phát riêng một stack mà các chương trình khác không được truy cập tới. Khi một hàm được gọi thực thi thì tất cả các biến cục bộ của hàm được đưa vào trong stack và sau khi gọi hàm hoàn thành thì những biến cục bộ của hàm đều bị đẩy ra khỏi stack.
- Chỗ thứ hai là **heap**: một vùng bộ nhớ dùng lưu trữ dữ liệu có dung lượng thay đổi như kiểu chuỗi chẳng hạn, hoặc dữ liệu có thời gian sống dài hơn phương thức của một đối tượng. Chẳng hạn, khi tạo thê hiện của một đối tượng, đối tượng được lưu trữ trên heap, và nó không bị tổng ra khi hàm hoàn thành giống như stack, mà ở nguyên tại chỗ và có thể trao cho các phương thức khác thông qua một tham chiếu.

Trên C#, heap này được gọi là **managed heap**, và được bộ dọn rác (garbage collector) chuyên lo thu hồi những vùng nhớ không được tham chiếu đến.

Kiểu giá trị được định nghĩa trước

Kiểu số nguyên(integer):

C# hỗ trợ 8 kiểu dữ liệu số nguyên sau:

Tên	Kiểu	Mô tả	Miền(min:max)
sbyte	System.SByte	Số nguyên có dấu 8-bit	-128:127 (-2 ⁷ :2 ⁷ -1)
short	System.Int16	Số nguyên có dấu 16-bit	-32,768:32,767 (-2 ¹⁵ :2 ¹⁵ -1)
int	System.Int32	Số nguyên có dấu 32-bit	2,147,483,648:2,147,483,647 (-2 ³¹ :2 ³¹ -1)
long	System.Int64	Số nguyên có dấu 64-bit	-9,223,372,036,854,775,808: 9,223,372,036,854,775,807 (-2 ⁶³ :2 ⁶³ -1)
byte	System.Byte	Số nguyên có dấu 8-bit	0:255 (0:2 ⁸ -1)
ushort	System.UInt16	Số nguyên có dấu 16-bit	0:65,35 (0:2 ¹⁶ -1)
uint	System.UInt32	Số nguyên có dấu 32-bit	0:4,294,967,295 (0:2 ³² -1)
ulong	System.UInt64	Số nguyên có dấu 64-bit	0:18,446,744,073,709,551,615(0:2 ⁶⁴ -1)

Ví dụ:

```
long x = 0x12ab; // ghi theo hexa
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

Kiểu dữ liệu số thực dấu chấm di động (Floating Point Types):

Tên	Kiểu	Mô tả	Miền
Float	System.Single	32-bit	$\pm 1.5 \times 10^{-45}$ đến $\pm 3.4 \times 10^{38}$
Double	System.Double	64-bit	$\pm 5.0 \times 10^{-324}$ đến $\pm 1.7 \times 10^{308}$

Ví dụ:

```
float f = 12.3F;
```

Kiểu dữ liệu số thập phân (Decimal Type):

Tên	Kiểu	Mô tả	Miền
decimal	System.Decimal	128-bit	$\pm 1.0 \times 10^{-28}$ đến $\pm 7.9 \times 10^{28}$

Ví dụ:

```
decimal d = 12.30M; //có thể viết decimal d = 12. 30m;
```

Kiểu Boolean:

Tên	Kiểu	Giá trị
Bool	System.Boolean	true hoặc false

Kiểu Character Type:

Tên	Kiểu	Giá trị
char	System.Char	Dùng unicode 16 bit

Kiểu tham chiếu định nghĩa trước:

C# hỗ trợ hai kiểu dữ liệu được định nghĩa trước:

Tên	Kiểu	Mô tả
object	System.Object	Kiểu cha của tất cả các kiểu trong CLR
string	System.String	Chuỗi kí tự unicode

Các ký tự escape thông dụng:

Thứ tự	Kí tự
\'	Nháy đơn
\\"	Nháy kép
\\\	Dấu xuyệt
\0	Null
\a	Cảnh báo
\b	Phím lui
\f	Form feed
\n	Xuống hàng
\r	Xuống hàng
\t	Tab
\v	Tab dọc

2. 4 Điều khiển luồng

Cú pháp:

```
if ( biểu thức)
    lệnh 1
else
    lệnh 2
```

Ví dụ minh họa lệnh rẽ nhánh

```
using System;
class Values
{
    static void Main( )
    {
        int valueOne = 10;
        int valueTwo = 20;
        if ( valueOne > valueTwo )
        {
            Console.WriteLine("Giá trị một: {0} lớn hơn giá trị hai: {1}", valueOne, valueTwo);
        }
        else
        {
            Console.WriteLine("Giá trị hai: {0} lớn hơn giá trị một: {1}", valueTwo, valueOne);
        }
        valueOne = 30; //gán lại giá trị mới
        if ( valueOne > valueTwo )
        {
            valueTwo = valueOne++;
            Console.WriteLine("\nGán giá trị một cho giá trị hai, ");
            Console.WriteLine("và tăng giá trị một lên hai. \n");
            Console.WriteLine("Giá trị một: {0}, Giá trị hai: {1}",
                valueOne, valueTwo);
        }
        else
        {
            valueOne = valueTwo;
            Console.WriteLine("Gán hai giá trị bằng nhau và bằng giá trị hai. ");
            Console.WriteLine("Giá trị một: {0} giá trị hai: {1}",
                valueOne, valueTwo);
        }
    }
}
```

```
}
```

Lệnh switch

Cú pháp:

```
switch (biểu thức)
{
    case biểu_thức_hằng:
        lệnh
        lệnh_nhảy
    [default: lệnh]
}
```

Ví dụ:

```
class Values
{
    static void Main( )
    {
        int chonThucDon;
        chonThucDon = 2;
        switch (chonThucDon)
        {
            case 1:
                System.Console.WriteLine(" Chọn món 1");
                break;
            case 2:
                System.Console.WriteLine(" Chọn món 1");
                break;
            default:
                System.Console.WriteLine(" Phải chọn món có trong thực đơn 1");
                break;
        }
    }
}
```

Lệnh lặp

Lệnh goto

Để sử dụng lệnh goto chúng ta cần:

- Tạo nhãn
- Goto trên nhãn

Ví dụ minh họa sử dụng lệnh goto

```
using System;
```

```

public class Tester
{
    public static int Main( )
    {
        int i = 0;
        repeat: // gán nhãn cho lệnh goto
            Console.WriteLine("i: {0}", i);
            i++;
            if (i < 10)
                goto repeat;
            return 0;
    }
}

```

Lệnh lặp while

Cú pháp:

```
while (biểu thức) lệnh;
```

Ví dụ dùng lệnh while

```

using System;
public class Tester
{
    public static int Main( )
    {
        int i = 0;
        while (i < 10)
        {
            Console.WriteLine("i: {0}", i);
            i++;
        }
        return 0;
    }
}

```

Lệnh do...while

Cú pháp:

```
do biểu_thức while lệnh;
```

```

using System;
public class Tester
{

```

```

public static int Main( )
{
    int i = 11;
    do
    {
        Console.WriteLine("i: {0}", i);
        i++;
    } while (i < 10);
    return 0;
}

```

Lệnh for

Cú pháp:

```

for (khởi tạo; biểu_thúc; lặp)
    lệnh;

```

Ví dụ:

```

using System;
public class Tester
{
    public static int Main( )
    {
        for (int i=0;i<100;i++)
        {
            Console.Ghi("{0} ", i);
            if (i%10 == 0)
            {
                Console.WriteLine("\t{0}", i);
            }
        }
        return 0;
    }
}

```

Lệnh continue và break

Thỉnh thoảng chúng ta muốn quay lại vòng lặp mà không cần thực hiện các lệnh còn lại trong vòng lặp, chúng ta có thể dùng lệnh continue. Ngược lại, nếu chúng ta muốn thoát ra khỏi vòng lặp ngay lập tức chúng ta có thể dùng lệnh break;

Ví dụ:

```
using System;
public class Tester
{
    public static int Main( )
    {
        string signal = "0";
        while (signal != "X")
        {
            Console.Ghi("Nhập vào tín hiệu: ");
            signal = Console.ReadLine( );
            Console.WriteLine("Tín hiệu vừa nhập: {0}", signal);
            if (signal == "A")
            {
                Console.WriteLine("Lỗi, bỏ qua\n");
                break;
            }
            if (signal == "0")
            {
                Console.WriteLine("Bình thường. \n");
                continue;
            }
            Console.WriteLine("{0}Tạo tín hiệu tiếp tục !\n",
signal);
        }
        return 0;
    }
}
```

2. 5 Kiểu liệt kê

Kiểu này bổ sung những tính năng mới thuận tiện hơn kiểu hằng. Kiểu liệt kê là một kiểu giá trị phân biệt bao gồm một tập các tên hằng. Ví dụ chúng ta tạo hai hằng liên quan nhau:

```
const int Nguoi = 1; // độ Fahrenheit
const int Soi = 2;
```

Chúng ta có thể bổ sung một số hằng khác vào trong danh sách như:

```
const int NguoiCoTheBoi = 3; // độ Fahrenheit
```

Quá trình này thực hiện rất cồng kềnh. Do đó, chúng ta có thể dùng danh sách liệt kê để giải quyết vấn đề:

```
enum Nguoi
```

```

{
    NheitDoDongLanh = 32;
    NheitDoSoi = 212;
    NheitDoCoTheBoi= 72;
}

```

Mỗi kiểu liệt kê phải có một kiểu bên dưới, chúng có thể là (int, short, long. ..) ngoại trừ kiểu char.

Ví dụ sau minh họa dùng kiểu enum để định nghĩa một thực đơn cho chương trình:

```

enum Menu
{
    Thoat=5,
    VeTamGiac,
    VeTamGiacLatNguoc,
    VeTamGiacGiuaManHinh,
    VeTamGiacRong
};

static void ThucDon()
{
    while(true)
    {
        int menu=0;

        Console.WriteLine("Nhập {0} để vẽ tam giác", (int)Menu.
VeTamGiac);

        Console.WriteLine("Nhập {0} để vẽ tam giác giữa man hinh",
(int)Menu. VeTamGiacGiuaManHinh);

        Console.WriteLine("Nhập {0} để vẽ tam giác lat nguoc ",
(int)Menu. VeTamGiacLatNguoc);

        Console.WriteLine("Nhập {0} để vẽ tam giác rong ", (int)Menu.
VeTamGiacRong);

        Console.WriteLine("Nhập {0} để thoát ", (int)Menu. Thoat);

        menu = int.Parse(Console.ReadLine());
        switch (menu)
        {
            case (int)Menu. VeTamGiac:
                VeTamGiac();
                break;
            case (int)Menu. VeTamGiacGiuaManHinh:
                VeTamGiacGiuaManHinh();
                break;
            case (int)Menu. VeTamGiacLatNguoc:
                VeTamGiacLatNguoc();
                break;
        }
    }
}

```

```

        break;

        case (int)Menu. VeTamGiacRong:
            VeTamGiacRong();
            break;
        default:
            return;
        }
    }
}

```

2. 6 Mảng

Mảng là một cấu trúc dữ liệu cấu tạo bởi một số biến được gọi là những phần tử mảng. Tất cả các phần tử này đều thuộc một kiểu dữ liệu. Bạn có thể truy xuất phần tử thông qua chỉ số . Chỉ số bắt đầu bằng 0.

Có nhiều loại mảng (array): mảng một chiều, mảng nhiều chiều...

Cú pháp:

```
kiểu[ ] tên+mảng;
```

Ví dụ:

```

int[] myIntegers; // mảng kiểu số nguyên
string[] myString ; // mảng kiểu chuỗi chữ

```

Bạn khai báo mảng có chiều dài xác định với từ khoá **new** như sau:

```

int[] integers = new int[32];
integers[0] = 35;// phần tử đầu tiên có giá trị 35
integers[31] = 432;// phần tử 32 có giá trị 432
Bạn cũng có thể khai báo như sau:
int[] integers;
integers = new int[32];
string[] myArray = {"phần tử 1", " phần tử 2", " phần tử 3"};

```

Chương trình minh họa dùng mảng 1 chiều:

```

using System;
using System.Collections.Generic;
using System.Text;
namespace Mang1Chieu
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a = new int[10];

```

```

        int len =0;
        Console.WriteLine("Nhập vào chiều dài mảng");
        len = int.Parse(Console.ReadLine());
        for(int i=0; i< len; i++)
        {
            Console.Write("Nhập a[{0}] = ", i);
            a[i] = int.Parse(Console.ReadLine());
        }
        for (int i = 0; i < len; i++)
            Console.WriteLine("\t{0}", a[i]);
    }
}

```

2. 7 Không gian tên (Namespace)

Namespace cung cấp cho ta cách tổ chức quan hệ giữa các lớp và các kiểu khác nhau. Đây là kĩ thuật cho phép .NET tránh việc các tên lớp, tên biến, tên hàm... đụng độ vì trùng tên với nhau giữa các lớp.

Để khai báo không gian tên chúng ta sử dụng từ khóa namespace. Ví dụ:

```

namespace Programming_C_Sharp
{
    using System;
    public class Tester
    {
        public static int Main( )
        {
            for (int i=0;i<10;i++)
            {
                Console.WriteLine("i: {0}", i);
            }
            return 0;
        }
    }
}

```

Ví dụ sau khai báo không gian tên lồng nhau:

```

namespace LapTrinh_C_Sharp
{
    namespace LapTrinh _C_Sharp_Test
    {
    }
}

```

```

{
    using System;
    public class Tester
    {
        public static int Main( )
        {
            for (int i=0;i<10;i++)
            {
                Console.WriteLine("i: {0}", i);
            }
            return 0;
        }
    }
}

```

Câu lệnh using

Từ khoá using giúp bạn giảm thiểu việc phải gõ những namespace trước các hàm hành sự hoặc thuộc tính. Ví dụ sau sử dụng Console.WriteLine thay vì phải gõ đầy đủ đường dẫn System. Console.WriteLine:

```

using System;
class Test
{
    public static int Main()
    {
        Console.WriteLine("Khong gian ten ");
        return 0;
    }
}

```

2. 8 Phương thức Main()

Khi một ứng dụng dòng lệnh hoặc ứng dụng Windows được biên dịch, theo mặc định trình biên dịch nhìn vào phương thức Main() như là điểm bắt đầu của chương trình. Nếu có nhiều hơn một phương thức Main(), trình biên dịch sẽ trả về thông báo lỗi. Do đó, mọi chương trình C# phải chứa một phương thức Main().

2. 9 Biên dịch nhiều tập tin C#

Tùy chọn	Xuất
/t:exe	Ứng dụng console mặc định
/t:library	Lớp thư viện với manifest
/t:module	Thành phần không có manifest

Tùy chọn	Xuất
/t:winexe	Một cửa sổ ứng dụng

Ví dụ tập tin MathLibrary.cs có nội dung như sau:

```
public class MathLib
{
    public int Cong(int x, int y)
    {
        return x + y;
    }
}
```

Chúng ta biên dịch file C# trên thành thư viện liên kết động (DLL) .NET bằng cách sử dụng câu lệnh sau:

```
csc /t:library MathLibrary.cs
```

Sử dụng thư viện trong các chương trình MathClient.cs như sau:

```
using System;
class Client
{
    public static void Main()
    {
        MathLib mathObj = new MathLib();
        Console.WriteLine(mathObj.Cong(7, 8));
    }
}
```

biên dịch chương trình

```
csc MathClient.cs /r:MathLibrary.dll
kết quả là 15
```

2. 10 Xuất nhập qua Console

Ứng dụng dòng lệnh là ứng dụng không có giao diện người dùng. Việc xuất nhập thông qua dòng lệnh chuẩn. Phương thức Main() trong ví dụ “Hello World” viết chuỗi “Chuong Trinh Dau Tien” lên màn hình. Màn hình được quản lý bởi một đối tượng tên Console. Đối tượng này có một phương thức WriteLine() nhận một chuỗi và xuất chúng ra thiết bị xuất chuẩn (màn hình).

Để đọc một ký tự văn bản từ cửa sổ console, chúng ta dùng phương thức:

- Console.Read(): giá trị trả về sẽ là kiểu int hoặc kiểu string.

Hai phương thức dùng để xuất chuỗi ký tự:

- Console.Write() - Viết một giá trị ra của sổ.

- `Console.WriteLine()` - tương tự như trên nhưng sẽ tự động xuống hàng khi kết thúc lệnh.

Ví dụ sau sẽ cho giá trị nhập kiểu int và giá trị xuất ra kiểu chuỗi

```
int x = Console.Read();
Console.WriteLine((char)x);
```

Giá trị trả về kiểu string:

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

Giả sử có đoạn mã như sau:

```
int i = 10;
int j = 20;
Console.WriteLine("{0} cộng {1} bằng {2}", i, j, i + j);
```

Kết quả hiển thị như sau:

```
10 cộng 20 bằng 30
int i = 940;
int j = 73;
Console.WriteLine(" {0, 4}\n+{1, 4}\n----\n {2, 4}", i, j, i + j);
```

Kết quả:

```
940
+
73
-----
1013
```

2. 11 Sử dụng chú thích

Như chúng ta đã lưu ý lúc đầu, C# sử dụng kiểu truyền thống của C cho việc chú thích. Dùng `//` cho hàng đơn và `/*...*/` cho một khối lệnh. Các đoạn mã lệnh trong chương trình C# cũng có thể chứa những dòng chú thích. Ví dụ:

```
// Chú thích trên 1 dòng
/* Chú thích
   trên 2 dòng */
Console.WriteLine(/* Kiểm tra chú thích! */ "Biên dịch bình thường");
DoSomething(Width, /*Height*/ 100);
string s = /* Đây là chuỗi không phải chú thích*/;
```

2. 12 Chỉ dẫn tiền xử lý trong C#

Từ định danh là tên dùng cho việc đặt tên biến cũng như để định nghĩa kiểu sử dụng như các lớp, cấu trúc, và các thành phần của kiểu này. C# có một số quy tắc để định rõ các từ định danh như sau:

- Chúng phải bắt đầu bằng ký tự không bị gạch dưới.

- Chúng ta không được sử dụng từ khoá làm từ định danh.

Trong C# có sẵn một số từ khoá (keyword).

abstract	do	implicit	params	switch
as	double	in	private	this
base	else	int	protected	throw
bool	enum	interface	public	true
break	event	internal	readonly	try
byte	explicit	is	ref	typeof
case	extern	lock	return	uint
catch	false	long	sbyte	ulong
char	finally	namespace	sealed	unchecked
checked	fixed	new	short	unsafe
class	float	null	sizeof	ushort
const	for	object	stackalloc	using
continue	foreach	operator	static	virtual
decimal	goto	out	string	volatile
default	if	override	struct	void
delegate				while

Chương 3: Đối tượng và kiểu

Mục đích của chương:

- Sử dụng kế thừa, phương thức ảo.
- Sử dụng nạp chồng phương thức: C# cho phép bạn định nghĩa những dạng khác nhau của một phương thức trong một lớp. Trình biên dịch sẽ tự động chọn phương thức nào thích hợp nhất dựa vào tham số truyền vào của nó.
- Phương thức tạo lập và phương thức hủy: chỉ rõ một số hành động tự động kèm theo khi khởi tạo đối tượng và tự động giải phóng khi kết thúc đối tượng.
- Cấu trúc (struct): là những kiểu giá trị cung cấp những tiện ích khi chúng ta cần một số tính năng của lớp nhưng không cần vất vả tạo ra một thực thể lớp.
- Nạp chồng toán hạng : kiểm tra cách để định nghĩa những toán hạng cho lớp.
- Chỉ mục: cho phép một lớp được xử lý chỉ mục khi nó là một mảng và đơn giản hóa cách sử dụng những lớp chứa các tập đối tượng.
- Giao diện : C# hỗ trợ kế thừa thông qua giao diện.

3. 1 Lớp và cấu trúc

Khả năng tạo các kiểu dữ liệu mới là đặc trưng của ngôn ngữ lập trình hướng đối tượng. Chúng ta có thể tạo ra kiểu mới bằng cách khai báo và định nghĩa lớp. Thể hiện của một lớp gọi là một đối tượng. Đối tượng được tạo trong bộ nhớ khi chương thực thi.

Một thuận lợi lớn nhất của các lớp trong ngôn ngữ lập trình hướng đối tượng là chúng có khả năng đóng gói các đặc trưng và khả năng của một thực thể vào trong một đơn vị mã chương trình.

Định nghĩa lớp (class)

Để định nghĩa một kiểu mới hay một lớp chúng ta đầu tiên khai báo nó và sau đó định nghĩa các phương thức và trường của nó. Chúng ta khai báo lớp sử dụng từ khóa class.

Cú pháp:

```
[thuộc_tính] [bổ_tù_truy_xuất] class định_danh [:lớp_có_sở ]  
{  
    Nội dung lớp  
}
```

Thông thường một lớp sẽ dùng bổ_tù_truy_xuất là từ khóa public

Định_danh là tên của lớp. Nội dung của lớp được định nghĩa trong phần {}.

Trong C# mọi thứ xảy ra trong một lớp. Ví dụ sau định nghĩa một lớp Tester:

```
public class Tester  
{
```

```

public static int Main( )
{
/
}
}

```

Lúc khai báo một lớp mới, chúng ta định nghĩa thuộc tính của tất cả đối tượng cũng như những hành vi của lớp đó. Ví dụ: một lớp SinhVien có các thuộc tính: maSV, tenSV, ngaySinh, tuoi và các hành vi như sau: XemDiem, XemThongTinHocPhi. Ngôn ngữ lập trình hướng đối tượng phép chúng ta tạo một kiểu mới SinhVien, đồng thời đóng gói những đặc trưng (thuộc tính) và khả năng (phương thức) của chúng. Lúc đó, lớp SinhVien có thể có các biến thành viên(trường) như maSV, tenSV, ngaySinh, tuoi và các hàm thành viên XemDiem(), XemThongTinHocPhi().

Lớp SinhVien có thể được khai báo như sau:

```

public class SinhVien
{
    public string maSV;
    public string tenSV;
    public DateTime ngaySinh;
    public int tuoi;
    public void XemDiem()
    {
        //Nội dung ham
    }
    public void XemThongTinHocPhi()
    {
        //Nội dung ham
    }
}

```

Chúng ta không thể gán dữ liệu đến kiểu SinhVien, đầu tiên chúng ta phải khai báo một đối tượng kiểu SinhVien theo đoạn mã sau.

```
SinhVien sv1;
```

Một khi chúng ta tạo một thê hiện của lớp SinhVien chúng ta có thể gán dữ liệu đến các trường của nó. Dùng từ khóa new để tạo đối tượng.

```

sv1 = new SinhVien();
sv1.maSV ="012345";

```

Xét một lớp theo dõi và hiển thị thời gian của ngày. Trạng thái bên trong của lớp phải có khả năng biểu diễn năm, tháng, ngày, giờ, phút và giây hiện tại. Chúng ta muốn thời gian hiển thị ở nhiều dạng khác nhau. Chúng ta có thể thực hiện được yêu cầu của lớp trên bằng cách khai báo một lớp định nghĩa một phương thức và sáu biến như sau:

```
using System;
```

```

public class Time
{
    // phương thức public
    public void HienThi( )
    {
        Console.WriteLine("Hiển thị thời gian hiện tại");
    }

    // biến private
    int Nam;
    int Thang;
    int Ngay;
    int Gio;
    int Phut;
    int Giay;
}

public class Tester
{
    static void Main( )
    {
        Time t = new Time( );
        t. HienThi( );
    }
}

```

Phương thức HienThi() được định nghĩa trả về kiểu void. Nó không trả về giá trị cho phương thức gọi nó. Trong hàm Main() thể hiện của lớp Time được tạo và địa chỉ của nó được gán đến đối tượng t. Bởi vì t là thể hiện của đối tượng Time nên nó có thể sử dụng phương thức HienThi() để gọi phương thức hiển thị thời gian:

t. HienThi ();

3.2 Thành viên của lớp

Kiểu truy xuất

Kiểu truy xuất xác định các thành viên (bao gồm trường, thuộc tính, phương thức, sự kiện, ủy nhiệm) của lớp có thể được nhìn thấy và sử dụng tại các thành viên bên trong hay bên ngoài lớp hay không. Bảng sau liệt kê các kiểu truy xuất và phạm vi của chúng:

Kiểu truy xuất	Hạn chế
Public	Không hạn chế. Thành viên được đánh dấu public được nhìn thấy bởi bất kỳ phương thức của bất kỳ lớp

private	Các thành viên trong lớp A được đánh dấu private được truy xuất chỉ trong các phương thức của lớp A. Mặc định các thành viên được khai báo private.
protected	Các thành viên trong lớp A được đánh dấu protected được truy xuất trong các phương thức của lớp A và các lớp dẫn xuất từ A
internal	Các thành viên trong lớp A được đánh dấu internal được truy xuất trong các phương thức của bất kỳ lớp trong assembly của A
protected internal	Tương đương với protected or internal

Biến thành viên của lớp nên được khai báo như sau:

```
int Nam;
int Thang;
int Ngay;
int Gio;
int Phut;
int Giay;
```

Tham số của phương thức

Phương thức có thể có số tham số bất kỳ. Tham số khai báo theo sau tên phương thức, mỗi tham số phải được gán một kiểu dữ liệu. Ví dụ sau định nghĩa một tên phương thức “PhuongThuc” và trả về giá trị void, nó nhận hai tham số kiểu int và Button:

```
void PhuongThuc (int thamsol, Button thamso2)
{
// ...
}
```

Bên trong phần thân của phương thức, tham số hoạt động như là một biến cục bộ. Ví dụ sau minh họa cách truyền giá trị vào một phương thức, trong trường hợp này giá trị của tham số có kiểu int và float

```
using System;
public class MyClass
{
    public void PhuongThuc(int ts1, float ts2)
    {
        Console.WriteLine("Các tham số nhận được: {0}, {1}", ts1, ts2);
    }
}
```

```

}
public class Tester
{
    static void Main( )
    {
        int n = 5;
        float pi = 3.14f;
        MyClass mc = new MyClass( );
        mc.PhuongThuc( n, pi );
    }
}

```

Tạo đối tượng

Trong chương trước chúng ta phân biệt giữa kiểu giá trị và kiểu tham chiếu. Các kiểu cơ bản trong C# là kiểu giá trị và chúng được tạo trên stack. Đối tượng vì nó là một kiểu tham chiếu nên được tạo trên heap. Ví dụ:

```
Time t = new Time();
```

Trong trường hợp này t không thật sự chứa đối tượng Time, nó chỉ chứa địa chỉ của đối tượng Time được tạo trên Heap. t chỉ thật sự là một tham chiếu của đối tượng đó.

Phương thức tạo lập

Trong lệnh sau:

```
Time t = new Time();
```

Một phương thức được gọi bất cứ lúc nào chúng ta tạo một thể hiện cho một đối tượng gọi là phương thức tạo lập. Nếu chúng ta không định nghĩa nó trong phần khai báo lớp, CLR sẽ cung cấp một phương thức mặc định đại diện cho nó. Nhiệm vụ của phương thức tạo lập là tạo đối tượng chỉ bởi lớp và đặt đối tượng vào trong trạng thái sẵn sàng. Trước khi phương thức tạo lập chạy, đối tượng chưa tồn tại trong bộ nhớ, sau khi phương thức tạo lập thực hiện hoàn thành, bộ nhớ sẽ lưu trữ một thể hiện hợp lệ của một lớp.

Trong ví dụ lớp Time, chúng ta không định nghĩa một phương thức tạo lập, trình biên dịch sẽ tự động tạo một phương thức tạo lập mặc định cho nó. Khi sử dụng phương thức tạo lập mặc định các biến được khởi tạo giá trị mặc định như sau:

Kiểu	Giá trị mặc định
Kiểu số (int, long, . .)	0
Bool	False
Char	'\0'

Enum	0
Reference	null

Thông thường chúng ta định nghĩa riêng một phương thức tạo lập và cung cấp tham số cho phương thức tạo lập để khởi tạo các biến cho đối tượng của chúng ta.

Để định nghĩa một phương thức tạo lập, chúng ta khai báo một phương thức có tên trùng với tên lớp mà chúng ta khai báo. Phương thức này không trả về bất cứ giá trị nào và thông thường được khai báo public. Tham số sử dụng trong phương thức tạo lập cũng như những tham số trong phương thức khác.

Ví dụ sau khai báo một phương thức tạo lập cho lớp SinhVien và chấp nhận các tham số có các kiểu dữ liệu khác nhau:

```
using System;
namespace QuanLySinhVien
{
    public class SinhVien
    {
        private float _diem;
        public string maso;
        public string ten;
        public bool gioitinh;
        public float diem;

        public SinhVien(string maso, string ten, bool gioitinh, float diem)
        {
            this.maso = maso;
            this.ten = ten;
            this.gioitinh = gioitinh;
            this.diem = diem;
        }

        private string GT(bool gt)
        {
            return gt?"Nam":"Nu";
        }

        public override string ToString()
        {
            return maso + "\t" + ten + "\t" + GT(this.gioitinh) +
"\t" + diem.ToString();
        }
    }
}
```

```

        static void Main(string[] args)
    {
        //Khởi tạo một biến s thuộc kiểu sinh viên
        SinhVien s = new SinhVien("001", "Thanh", false, 8);
        //Xuất thông tin sinh viên ra màn hình
        Console.WriteLine(s);
    }
}

```

Khởi tạo giá trị cho các biến

Thay vì khởi tạo giá trị các biến thành viên thông qua phương thức tạo lập chúng ta có thể thực hiện gán giá trị trực tiếp cho biến. Phương thức tạo lập của chương trình trên có thể được viết như sau:

```

public void SinhVien()
{
    Console.Ghi("Nhập maso ");
    maso = Console.ReadLine();
    Console.Ghi("Nhập tên ");
    ten = Console.ReadLine();
    Console.Ghi("Nhập giới tính 1 cho Nam, 0 cho Nữ ");

    int i = int.Parse(Console.ReadLine());
    if(i==1)
        this.gioitinh = true;
    else
        this.gioitinh = false;
    Console.Ghi("Nhập điểm ");
    diem = float.Parse(Console.ReadLine());
}

```

Phương thức tạo lập sao chép

Phương thức tạo lập sao chép tạo một đối tượng mới bằng cách chép các biến từ đối tượng hiện tại đến đối tượng mới cùng kiểu. Ví dụ chúng ta muốn truyền một đối tượng SinhVien b đến một đối tượng SinhVien a của phương thức tạo lập nhằm tạo ra đối tượng mới có cùng giá trị với đối tượng cũ. C# không cung cấp phương thức tạo lập sao chép, do đó chúng ta phải tự tạo. Ví dụ:

```

public SinhVien(SinhVien sv)
{
    this.maso = sv.maso;
    this.ten = sv.ten;
    this.gioitinh = sv.gioitinh;
    this.diem = sv.diem;
}

```

```
}
```

Dựa trên việc khai báo phương thức tạo lập bên trên ta có thể tạo một đối tượng sinh viên b từ một đối tượng sinh viên a như sau:

```
SinhVien a = new SinhVien("001", "Thanh", false, 8);
SinhVien b = new SinhVien(a)
```

Tùy khóa this

Tùy khóa this chỉ ra trạng thái hiện tại của đối tượng. Tham chiếu this (con trỏ this) là một con trỏ ẩn đến các hàm không tĩnh (không khai báo từ khóa static) của lớp. Mỗi phương thức có thể tham chiếu đến các phương thức và các biến khác dựa trên tham chiếu this.

Tham chiếu this có thể sử dụng theo ba trường hợp sau:

- Tránh xung đột tên.
- Truyền đối tượng hiện tại là tham số của một phương thức khác.
- Dùng với chỉ mục.

Sử dụng các thành viên tĩnh

Thuộc tính và phương thức của một lớp có thể là thành viên của thể hiện hay thành viên tĩnh. Thành viên thể hiện được kết hợp với thể hiện của kiểu, trong khi thành viên tĩnh được coi là phần của lớp. Chúng ta truy xuất các thành viên tĩnh thông qua tên lớp chúng ta khai báo.

```
using System;
namespace Test
{
    class Mang1Chieu
    {
        static int []a = new int[100];
        static int n = 0;
        static void Main(string[] args)
        {
            Nhap();
            //Chúng ta cũng có thể gọi phương thức tĩnh như sau
            Mang1Chieu.Xuat();
            Console.WriteLine("Tong cac phan tu mang la " +
TinhTong().ToString());
            Console.ReadLine();
        }
        void NhapNgauNhien()
        {
            Console.Ghi("Nhập vào chiều dài của mảng n = ");
            n = int.Parse(Console.ReadLine());
        }
        void Xuat()
        {
            Console.WriteLine("Mảng có " + n + " phần tử là:");
            for (int i = 0; i < n; i++)
                Console.WriteLine(a[i]);
        }
        int TinhTong()
        {
            int sum = 0;
            for (int i = 0; i < n; i++)
                sum += a[i];
            return sum;
        }
    }
}
```

```

        Random r = new Random();
        for(int i=0; i < n; i++)
            a[i] = r.Next(10);
    }

    static void Nhap()
    {
        Console.Ghi("Nhập vào chiều dài của mảng n = ");
        n = int.Parse(Console.ReadLine());
        for(int i=0; i < n; i++)
        {
            Console.Ghi("a[{0}] = ", i);
            a[i] = int.Parse(Console.ReadLine());
        }
    }

    static int TinhTong()
    {
        int sum = 0;
        int i = 0;
        while(i<n)
            sum += a[i++];
        return sum;
    }

    static void Xuat()
    {
        Console.Ghi("Mảng vừa nhập là ");
        for(int i=0; i < n; i++)
            Console.Ghi("{0} ", a[i]);
    }
}

```

Trong ví dụ trên chúng ta khai báo lớp Mang1Chieu và có các phương thức tĩnh như Nhập, Xuất. Chúng ta có thể gọi thực hiện các phương thức này trực tiếp mà không cần thông qua đối tượng hay thể hiện của lớp.

Trong C#, sẽ không hợp lệ nếu chúng ta truy xuất thành viên tĩnh qua thể hiện.

Gọi phương thức tĩnh

Hàm Main() là một phương thức tĩnh. Phương thức tĩnh hoạt động trên lớp hơn là trên thể hiện của lớp. Chúng không có tham chiếu this để trỏ đến phương thức tĩnh.

Phương thức tĩnh không thể truy xuất trực tiếp từ các thành viên không tĩnh.

Trong ví dụ trên phương thức NhapNgauNhien là phương thức không tĩnh, do đó phải được truy cập thông qua đối tượng như sau:

Ví dụ:

```
static void Main(string[] args)
{
    Mang1Chieu a = new Mang1Chieu();
    a.NhapNgauNhien();
    Xuat();
    Console.WriteLine("Tong cac phan tu mang la " + TinhTong().
ToString());
    Console.ReadLine();
}
```

Sử dụng phương thức tạo lập tĩnh

Nếu chúng ta khai báo một phương thức tạo lập tĩnh, chúng ta phải đảm bảo phương thức tạo lập tĩnh sẽ chạy trước khi thể hiện của lớp được tạo.

Ví dụ, chúng ta có thể thêm phương thức tạo lập tĩnh vào lớp Mang1Chieu

```
static Mang1Chieu( )
{
}
```

Chúng ta không được khai báo kiểu truy xuất trước phương thức tạo lập tĩnh. Hơn nữa, vì đây là một phương thức thành viên tĩnh, chúng ta không thể truy xuất vào biến thành viên không tĩnh, do đó tất cả các biến muốn được truy cập thông qua các thành viên tĩnh phải được khai báo static.

Sử dụng trường tĩnh

Dùng biến thành viên tĩnh là cách phổ biến để theo dõi số thể hiện hiện tại của lớp. Ví dụ:

```
using System;
public class Meo
{
    public Meo( )
    {
        thehien++;
    }
    public static void SoMeo( )
    {
        Console.WriteLine("Số thể hiện của mèo là{0} ", thehien);
    }
    private static int thehien = 0;
```

```

}
public class Tester
{
    static void Main( )
    {
        Meo.SoMeo( );
        Meo Tom= new Cat( );
        Meo.SoMeo( );
        Meo Linda = new Cat( );
        Meo.SoMeo( );
    }
}

```

Hủy đối tượng

C# cung cấp một cơ chế dọn rác để hủy các đối tượng một cách tự động và do đó chúng ta không cần một phương thức hủy rõ ràng để xóa các đối tượng trong bộ nhớ. Phương thức `Finalize()` được gọi bởi bộ dọn rác khi đối tượng của chúng ta bị hủy. Hàm này chỉ giải phóng các tài nguyên mà đối tượng này đang giữ và nó không tham chiếu đến các đối tượng khác.

Ví dụ khai báo phương thức hủy:

```

~ Mang1Chieu ()
{
    //Mã
}

```

Hay

```

Mang1Chieu.Finalize()
{
    //Mã
}

```

Truyền tham số

Mặc định các kiểu giá trị được truyền vào trong các phương thức bởi giá trị. Tuy nhiên trong nhiều trường hợp chúng ta muốn truyền giá trị đối tượng theo tham chiếu. C# cung cấp 2 cách truyền tham chiếu đối tượng vào trong phương thức:

- Dùng từ khóa `ref`.
- Dùng từ khóa `out`, đối tượng truyền vào không cần khởi gán.

Ngoài ra, C# còn cung cấp từ khóa `params` để cho phép một phương thức truyền số tham số tùy ý.

Truyền tham chiếu

Một phương thức chỉ có thể trả về một giá trị, do đó khi muốn phương thức trả về nhiều hơn một giá trị, chúng ta dùng cách thức truyền tham chiếu.

Ví dụ, thực hiện đoạn chương trình sau:

```
using System;
namespace Bien
{
    class Class1
    {
        static int a = 1;
        static int b = 2;
        static void GiaTri(int a, int b)
        {
            a = 10;
            b = 20;
        }
        static void Main(string[] args)
        {
            int a = 1;
            Console.WriteLine("a = {0}, b = {1}", a, b);
            GiaTri(a, b);
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.ReadLine();
        }
    }
}
```

Kết quả thực hiện chương trình là a =1, b =2 và a=1, b=2

Nhưng nếu chúng ta khai báo phương thức GiaTri như sau:

```
static void GiaTri(ref int a, int b)
{
    a = 10;
    b = 20;
}
```

thì kết quả thực hiện chương trình sẽ là a =1, b =2 và a=10, b=2. Ta thấy giá trị của biến a thay đổi nhưng giá trị của biến b không thay đổi sau khi chúng ta thực hiện gọi phương thức GiaTri vì biến a được truyền theo kiểu tham biến còn biến b được truyền theo kiểu tham trị.

Truyền tham số dùng từ khóa out

Mặc định C# quy định tất cả các biến phải được gán tham số trước khi sử dụng. Trong ví dụ trên nếu chúng ta không khởi tạo biến a trong hàm Main, chương trình thông dịch sẽ thông báo lỗi biến chưa được khởi gán như sau “Use of unassigned local variable 'a'”. Chúng ta có thể tránh bằng cách dùng từ khóa out và khai báo như sau:

```
using System;
namespace Bien
{
    class Class1
    {
        static int a = 1;
        static int b = 2;
        static void GiaTri(out int a, int b)
        {
            a = 10;
            b = 20;
        }
        static void Main(string[] args)
        {
            int a = 1;
            GiaTri(out a, b);
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.ReadLine();
        }
    }
}
```

Nạp chồng phương thức

Thông thường chúng ta muốn khai báo nhiều hàm với cùng tên. C# cho phép khai báo các hàm trùng tên nhưng dấu hiệu phải khác nhau. Ví dụ chúng ta muốn dùng nhiều phương thức tạo lập khác nhau để khởi tạo các kiểu đối tượng khác nhau. Dấu hiệu của một phương thức được định nghĩa bởi tên và danh sách tham số. Hai phương thức có dấu hiệu khác nhau nếu có tên và danh sách tham số khác nhau.

Tham số khác nhau bởi số tham số hay kiểu khác nhau. Ví dụ các hàm sau có các dấu hiệu khác nhau:

```
void PhuongThuc(int p1);
void PhuongThuc (int p1, int p2);
void PhuongThuc (int p1, string s1);
```

Ví dụ minh họa sử dụng quá tải hàm cho phương thức tạo lập của lớp MaTran

```

using System;
namespace MaTran
{
    public class MaTran
    {
        public double[, ] mt;
        public int hang;
        public int cot;
        #region Phuong thuc tao lap
        public MaTran()
        {
            //Phuong thuc tao lap mac dinh
        }
        public MaTran(int h, int c)
        {
            this.hang = h;
            this.cot = c;
            mt = new double[h, c];
        }
        public MaTran(MaTran a)
        {
            mt = new double[a. hang, a. cot];
            this.hang = a.hang;
            this.cot = a.cot;
            for (int i = 0; i < a.hang; i++)
                for (int j = 0; j < a.cot; j++)
                    this.mt[i, j] = a.mt[i, j];
        }
        public MaTran(MaTran a, int h, int c)
        {
            mt = new double[h, c];
            this.hang = h;
            this.cot = c;
            for (int i = 0; i < h; i++)
                for (int j = 0; j < c; j++)
                    this.mt[i, j] = a.mt[i, j];
        }
        public void Nhap()
        {
    
```

```

        Console.WriteLine("Nhập mảng ma trận cấp {0}x{1} ",
this.hang, this.cot);
        for (int i = 0; i < this.hang; i++)
            for (int j = 0; j < this.cot; j++)
            {
                //Console.Ghi("Nhập phần tử thứ [{0}, {1}] = ", i,
j);
                this.mt[i, j] = i + j;
            }
        }
    public void Nhap(int h, int c)
    {
        mt = new double[h, c];
        for (int i = 0; i < h; i++)
            for (int j = 0; j < c; j++)
            {
                Console.Ghi("Nhập phần tử thứ [{0}, {1}] = ", i,
j);
                this.mt[i, j] = double.Parse(Console.ReadLine());
            }
        }
    public override string ToString()
    {
        string kq = "";
        for (int i = 0; i < this.hang; i++)
        {
            kq += "\n";
            for (int j = 0; j < this.cot; j++)
            {
                kq += "\t" + this.mt[i, j].ToString();
            }
        }
        return kq;
    }
    public MaTran Tong(MaTran a, MaTran b)
    {
        MaTran kq = new MaTran(a. hang, a. cot);
        for (int i = 0; i < b.hang; i++)
            for (int j = 0; j < b.cot; j++)
                kq.mt[i, j] = a.mt[i, j] + b.mt[i, j];
        return kq;
    }
}

```

```

        }

        public void Tong(MaTran b)
        {
            for (int i = 0; i < b.hang; i++)
                for (int j = 0; j < b.cot; j++)
                    this.mt[i, j] = this.mt[i, j] + b.mt[i, j];
        }
    }
}

```

Trong ví trên chúng ta khai báo ba phương thức tạo lập khác nhau cho việc khởi tạo một đối tượng kiểu MaTran. Tương tự chúng ta cũng định nghĩa nhiều phương thức Tong với các tham số khác nhau cho việc cộng hai ma trận.

Đóng gói dữ liệu với thuộc tính

Các thuộc tính cho phép các client truy xuất tới trạng thái của lớp như là truy xuất các trường thành viên trực tiếp của lớp. Truy xuất thuộc tính tương tự như truy xuất phương thức của lớp. Nghĩa là các client có thể truy xuất trực tiếp đến trạng thái của đối tượng (thuộc tính) mà không cần thông qua việc gọi thực thi các phương thức. Tuy nhiên, những người thiết kế lớp hay muốn che dấu trạng thái bên trong của các thành viên lớp và chỉ muốn các thuộc tính chỉ truy xuất gián tiếp thông qua phương thức của lớp.

Bằng cách phân tách thuộc tính với các phương thức của lớp, các nhà thiết kế có thể tự do thay đổi giá trị hay trạng thái của các thuộc tính bên trong của đối tượng khi cần. Cho ví dụ như sau:

```

using System;
namespace DoiTuongDoiTuongHinhHoc
{
    public class HinhTron
    {
        private double r;
        public HinhTron()
        {
        }
        public HinhTron(double bankinh)
        {
            this.R = bankinh;
        }
        public double DienTich()
        {
            return Math.Round(Math.PI*R*R, 2);
        }
        public override string ToString()
    }
}

```

```

    {
        return "Duong tron co ban kinh " + R + " dien tich " +
this.DienTich();
    }
}
}

```

Lúc lớp `HinhTron` được tạo, giá trị của thuộc tính bán kính `r` có thể được lưu trữ là một biến thành viên. Lúc lớp được thiết kế lại, giá trị của thuộc tính `r` có thể được tính toán lại hay lấy từ cơ sở dữ liệu. Nếu client truy xuất trực tiếp biến thành viên `r`, những thay đổi giá trị tính toán có thể làm hỏng ứng dụng client. Bằng cách phân tách và bắt buộc client muốn truy cập thuộc tính `r` phải sử dụng thông qua phương thức. Ta có thể khai báo như sau để cho phép đổi tượng có thể truy cập biến `r` thông qua phương thức:

```

using System;
namespace DoiTuongDoiTuongHinhHoc
{
    public class HinhTron
    {
        private double r;
        public double R
        {
            get
            {
                return r;
            }
            set
            {
                r = value;
            }
        }
        public HinhTron()
        {
        }
        public HinhTron(double bankinh)
        {
            this.R = bankinh;
        }
        public double DienTich()
        {
            return Math.Round(Math.PI*R*R, 2);
        }
        public override string ToString()
        {
        }
    }
}

```

```

    {
        return "Duong tron co ban kinh " + R + " dien tich " +
this.DienTich();
    }
}
}

```

Thuộc tính đáp ứng cả hai mục đích:

- Chúng cung cấp một giao tiếp đơn giản với client, xuất hiện như là biến thành viên.
- Chúng thực hiện như là một phương thức. Tuy nhiên, chúng cung cấp cơ chế che dấu dữ liệu bởi một thiết kế hướng đối tượng.

Để khai báo thuộc tính, chúng ta viết kiểu thuộc tính và tên theo sau bởi {}. Bên trong {} chúng ta có thể khai báo cách thức truy xuất get hay set. Qua phương thức set chúng ta có một tham số ẩn value.

Trong ví dụ trên R là một thuộc tính, nó khai báo 2 kiểu truy xuất

```

public int R
{
    get
    {
        return r;
    }
    set
    {
        r = value;
    }
}

```

Giá trị của thuộc tính R có thể được lưu trữ trong cơ sở dữ liệu, trong biến thành viên private thậm chí lưu trữ tại một máy tính hay một dịch vụ nào đó trong mạng.

Truy xuất get

Phần thân của truy xuất get thì tương tự như phần thân của phương thức lớp. Chúng trả về một đối tượng có kiểu là kiểu của thuộc tính. Trong ví dụ trên, truy xuất get của R trả về giá trị double. Nó trả về giá trị của biến thành viên private r.

Bất cứ lúc nào chúng ta tham chiếu đến thuộc tính, truy xuất get được yêu cầu để đọc giá trị của thuộc tính.

```

HinhTron t = new HinhTron (1, 5);
double bankinh = t.R;

```

Truy xuất set

Truy xuất set dùng để gán giá trị cho thuộc tính, nó tương tự như một phương thức lớp trả về kiểu void. Lúc chúng ta định nghĩa một truy xuất set, chúng ta phải sử dụng từ

khóa value (tham số ẩn) để biểu diễn tham số nơi mà giá trị được truyền và lưu trữ trong thuộc tính:

```
set
{
    r = value;
}
```

Lúc chúng ta gán một giá trị đến một thuộc tính, truy xuất set tự động được yêu cầu và tham số ẩn value được đặt giá trị mà chúng ta cần gán cho thuộc tính:

Ví dụ ta có thể dùng lệnh sau trong một phương thức nào đó của lớp HinhTron để tăng giá trị của thuộc tính r lên 1:

```
r++;
```

Thuận lợi của các tiếp cận này là client có thể truy xuất trực tiếp thuộc tính mà không cần quan tâm tới các dữ liệu ẩn bên trong.

Các trường chỉ đọc (readonly)

Nếu chúng ta muốn tạo một phiên bản của lớp ThoiGian sau có nhiệm vụ cung cấp một giá trị tĩnh, biểu diễn ngày và thời gian hiện tại. Chúng ta có thể dùng khai báo readonly để khai báo các thuộc tính:

```
public class ThoiGian
{
    static ThoiGian()
    {
        System.DateTime dt = System.DateTime.Now;
        Nam = dt.Year;
        Thang = dt.Month;
        Ngay = dt.Day;
        Gio = dt.Hour;
        Phut = dt.Minute;
        Giay = dt.Second;
    }

    public static int Nam;
    public static int Thang;
    public static int Ngay;
    public static int Gio;
    public static int Phut;
    public static int Giay;
}

public class Tester
{
    static void Main()
```

```

{
    System.Console.WriteLine("This Nam: {0}",
        ThoiGian.Nam.ToString());
    ThoiGian.Nam = 2002;
    System.Console.WriteLine("This Nam: {0}",
        ThoiGian.Nam.ToString());
}
}

```

Trong trường hợp này, giá trị của biến Nam sẽ vẫn thay đổi khi chúng ta thực hiện phép gán Nam = 2002. Chúng ta muốn đánh dấu các giá trị tĩnh là hằng nhưng không thể vì chúng ta không khởi gán giá trị cho chúng cho tới khi phương thức tạo lập tĩnh được thực hiện. C# cung cấp từ khóa readonly dùng cho mục đích này. Nếu chúng ta thay đổi khai báo biến thành viên của lớp như sau:

```

public static readonly int Nam;
public static readonly int Thang;
public static readonly int Ngay;
public static readonly int Gio;
public static readonly int Phut;
public static readonly int Giay;;

```

và không dùng lệnh:

```
// ThoiGian. Year = 2002; // Thực hiện lệnh này sẽ gây ra lỗi
```

Trình biên dịch thực hiện và chạy theo đúng yêu cầu của chúng ta.

3.3 Cấu trúc (Struct)

Làm việc với kiểu cấu trúc

Kiểu dữ liệu tham chiếu luôn được tạo trên heap. Trong một số trường hợp, một lớp chứa quá ít dữ liệu để cần sự quản lý của heap. Tốt nhất trong trường hợp này bạn dùng cấu trúc. Bởi vì cấu trúc được lưu trữ trong stack, điều này giảm bớt chức năng quản lý bộ nhớ.

Cấu trúc cũng có riêng trường, phương thức tạo lập và phương thức giống lớp (nhưng không giống kiểu liệt kê). Tuy nhiên nó là kiểu giá trị không phải kiểu tham chiếu.

Các kiểu cấu trúc phổ biến

Trong C#, các kiểu số cơ bản như int, long, float là tên hiệu của cấu trúc System.Int32, System.Int64, và System.Single. Nghĩa là chúng ta có thể gọi phương thức trên biến thuộc kiểu này. Ví dụ, tất cả cấu trúc này cung cấp phương thức ToString để chuyển số sang chuỗi. Các lệnh sau là hợp lệ trong C#:

```

int i = 99;
Console.WriteLine(i. ToString());
Console.WriteLine(55. ToString());
float f = 98.765F;

```

```

Console.WriteLine(f. ToString());
Console.WriteLine(98.765F. ToString());

```

Console.WriteLine sẽ tự động gọi phương thức ToString khi cần. Sử dụng phương thức tĩnh trong cấu trúc rất phổ biến. Ví dụ phương thức tĩnh Int32.Parse thường được dùng chuyển một chuỗi sang số:

```

string s = "42";
int i = Int32.Parse(s);

```

Trong những cấu trúc trên cũng chứa những trường tĩnh hữu ích như Int32.MaxValue để lấy giá trị lớn nhất của biến int và Int32.MinValue lấy giá trị nhỏ nhất của biến int.

Bảng sau hiển thị kiểu dữ liệu cơ bản trong C#, nó có thể tương đương kiểu lớp hay cấu trúc:

Tùy khóa	Kiểu tương đương	Lớp hoặc cấu trúc
bool	System.Boolean	Cấu trúc
byte	System.Byte	Cấu trúc
decimal	System.Decimal	Cấu trúc
Double	System.Double	Cấu trúc
Float	System.Single	Cấu trúc
Int	System.Int32	Cấu trúc
Long	System.Int64	Cấu trúc
Object	System.Object	Lớp
Sbyte	System.SByte	Cấu trúc
Short	System.Int16	Cấu trúc
String	System.String	Lớp
Uint	System.UInt32	Cấu trúc
Ulong	System.UInt64	Cấu trúc
Ushort	System.UInt16	Cấu trúc

Khai báo kiểu cấu trúc

Để khai báo kiểu cấu trúc, bạn khai báo từ khóa struct theo sau bởi tên và phần thân trong dấu “{}”. Ví dụ, đây là cấu trúc tên ThoiGian chứa 3 trường public: Gio, Phut, và Giay.

```

struct Time
{
    public int Gio, Phut, Giay;
}

```

Tương tự như lớp, đánh dấu public cho các trường không được khuyến khích trong hầu hết trường hợp vì không có cách để đảm bảo trường public chứa giá trị hợp lệ. Ví dụ, bạn có thể gán trị của Phut hay Giay > 60. Cách tốt hơn là đánh dấu trường private và cung cấp truy xuất cấu trúc của bạn qua phương thức tạo lập hay phương thức như sau:

```

struct ThoiGian
{
    public ThoiGian (int hh, int mm, int ss)
    {
        Gio= hh % 24;
        phut = mm % 60;
        giay = ss % 60;
    }

    public int Gio()
    {
        return gio;
    }

    ...

    private int gio, phut, giay;
}

```

Khi bạn sao chép một biến kiểu giá trị, bạn có hai bản sao của giá trị. Ngược lại, lúc bạn sao chép một biến kiểu tham chiếu, bạn có hai tham chiếu đến cùng đối tượng.

Tìm hiểu sự khác nhau giữa lớp và cấu trúc

- Bạn không thể khai báo phương thức tạo lập mặc định (phương thức tạo lập không có tham số) cho cấu trúc. Ví dụ sau sẽ biên dịch nếu ThoiGian là lớp nhưng với cấu trúc thì không:

```

struct ThoiGian
{
    public ThoiGian () {...} // Lỗi biên dịch
    ...
}

```

Bởi vì trình biên dịch luôn tạo phương thức tạo lập mặc định cho cấu trúc. Trong khi trong lớp, trình biên dịch chỉ tạo nó khi không có khai báo phương thức tạo lập trong lớp.

Trình biên dịch tạo phương thức tạo lập mặc định cho một cấu trúc luôn gán 0, false, null giống như lớp cho các kiểu dữ liệu tương ứng. Do đó bạn muốn khởi tạo các giá

trị khác cho trường bạn phải dùng phương thức tạo lập không mặc định của bạn. Tuy nhiên bạn phải khởi tạo tất cả các trường trong cấu trúc, nếu không trình biên dịch sẽ thông báo lỗi. Ví dụ sau sẽ được biên dịch nếu ThoiGian là một lớp và giây được gán bằng 0 nhưng vì ThoiGian là cấu trúc nên lỗi khi biên dịch:

```
struct ThoiGian
{
    public ThoiGian (int hh, int mm)
    {
        gio = hh;
        phut = mm;
    } // lỗi thời gian biên dịch: giây chưa được khởi tạo
    ...
    private int gio, phut, giay;
}
```

- Trong một lớp bạn có thể khởi tạo trường lúc khai báo nhưng trong cấu trúc bạn không thể. Ví dụ sau sẽ được biên dịch nếu ThoiGian là một lớp nhưng vì ThoiGian là cấu trúc nên lỗi khi biên dịch:

```
struct ThoiGian
{
    ...
    private int gio = 0; // lỗi biên dịch
    private int phut;
    private int giay;
}
```

- Bảng sau tóm tắt sự khác nhau giữa lớp và cấu trúc

Câu hỏi	Cấu trúc	Lớp
Kiểu của nó là gì?	Cấu trúc là kiểu giá trị	Lớp là kiểu tham chiếu
Thể hiện của nó lưu trên stack hay heap?	Thể hiện của cấu trúc được gọi là giá trị và lưu trên stack	Thể hiện của lớp là đối tượng được lưu trên heap
Bạn có thể khai báo phương thức tạo lập mặc định?	Không	Có
Nếu bạn khai báo phương thức tạo lập riêng bạn, trình biên dịch sẽ tạo phương thức tạo lập mặc định?	Có	Không
Nếu bạn không khởi tạo trường trong phương thức tạo lập riêng bạn, trình biên	Không	Có

Câu hỏi	Cấu trúc	Lớp
dịch sẽ tự động khởi tạo cho bạn?		
Bạn được phép khởi tạo trường thẻ hiện lúc khai báo?	Không	Có

Còn một sự khác nhau khác giữa lớp và cấu trúc là lớp có thể kế thừa từ lớp cơ sở nhưng cấu trúc thì không.

Khai báo biến cấu trúc

Sau khi bạn định nghĩa một kiểu cấu trúc, bạn có thể dùng nó như các kiểu khác. Ví dụ, nếu bạn định nghĩa cấu trúc ThoiGian bạn có thể tạo biến, trường, tham số kiểu ThoiGian như sau:

```
struct ThoiGian
{
    ...
    private int gio, phut, giay;
}

class ViDu
{
    public void PhuongThuc(ThoiGian para)
    {
        ThoiGian bienCucBo;
        ...
    }
    private ThoiGian thoiGianHienTai;
}
```

Khởi tạo cấu trúc

Giả sử chúng ta định nghĩa lớp như sau:

```
struct Time
{
    ...
    private int hours, minutes, seconds;
```

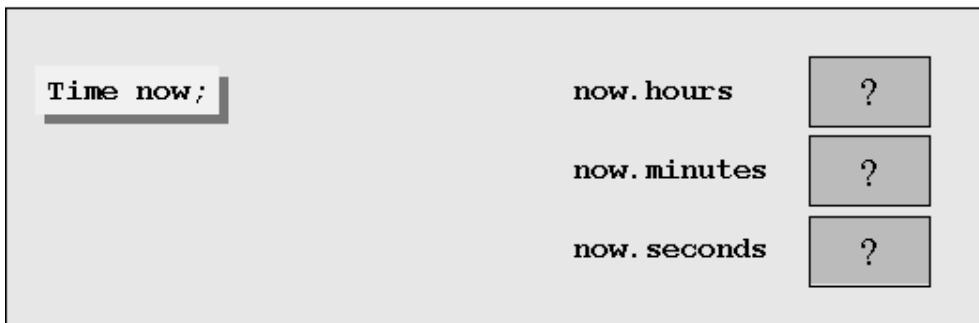
}

Vì cấu trúc là kiểu dữ liệu nên bạn có thể tạo biến cấu trúc mà không cần gọi phương thức tạo lập như ví dụ sau:

```
Time now;
```

Trong ví dụ này, biến được tạo nhưng trường bên trái của nó ở trạng thái chưa được tạo. Nếu bạn truy xuất giá trị của trường này sẽ gây ra lỗi biên dịch. Hình sau mô tả trạng thái của trường trong biến now:

STACK

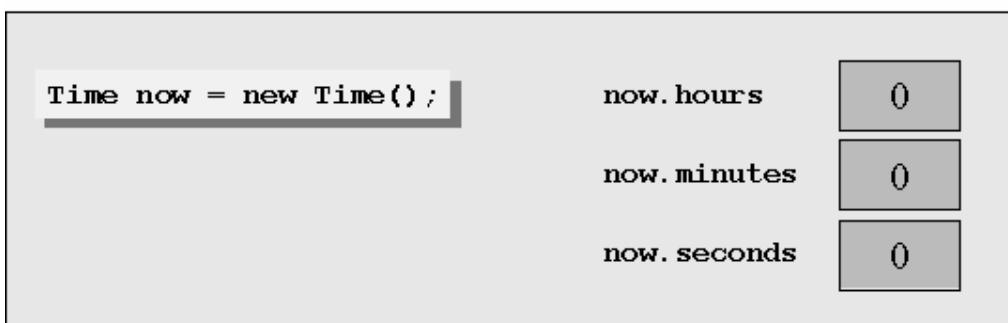


Nếu bạn gọi một phương thức tạo lập, quy tắc khởi tạo đảm bảo tất cả các biến được khởi tạo:

```
Time now = new Time();
```

Lúc này phương thức tạo lập mặc định được gọi khởi tạo tất cả các trường trong cấu trúc như hình sau:

STACK



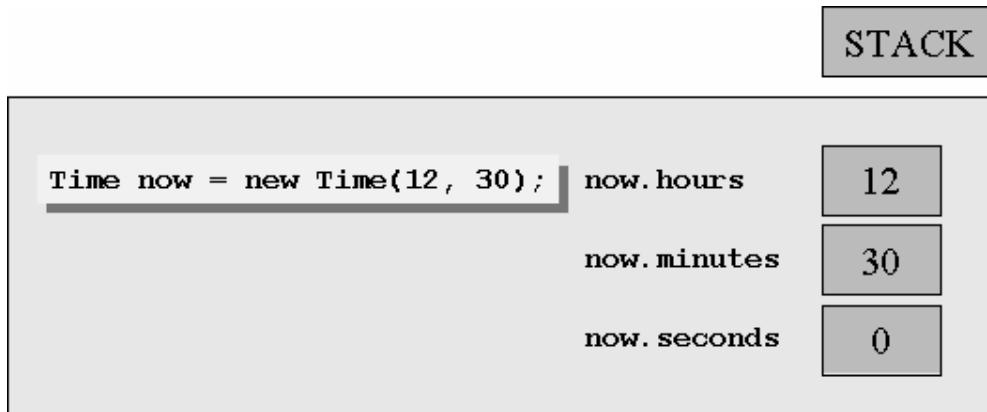
Nếu bạn muốn viết phương thức tạo lập riêng, bạn có thể dùng nó để khởi tạo biến cấu trúc. Một phương thức tạo lập của cấu trúc phải luôn khởi tạo tất cả các trường của nó. Ví dụ:

```
struct Time
{
    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
        seconds = 0;
    }
    ...
    private int hours, minutes, seconds;
}
```

Trong ví dụ sau khởi tạo `now` bởi gọi phương thức tạo lập người dùng định nghĩa:

```
Time now = new Time(12, 30);
```

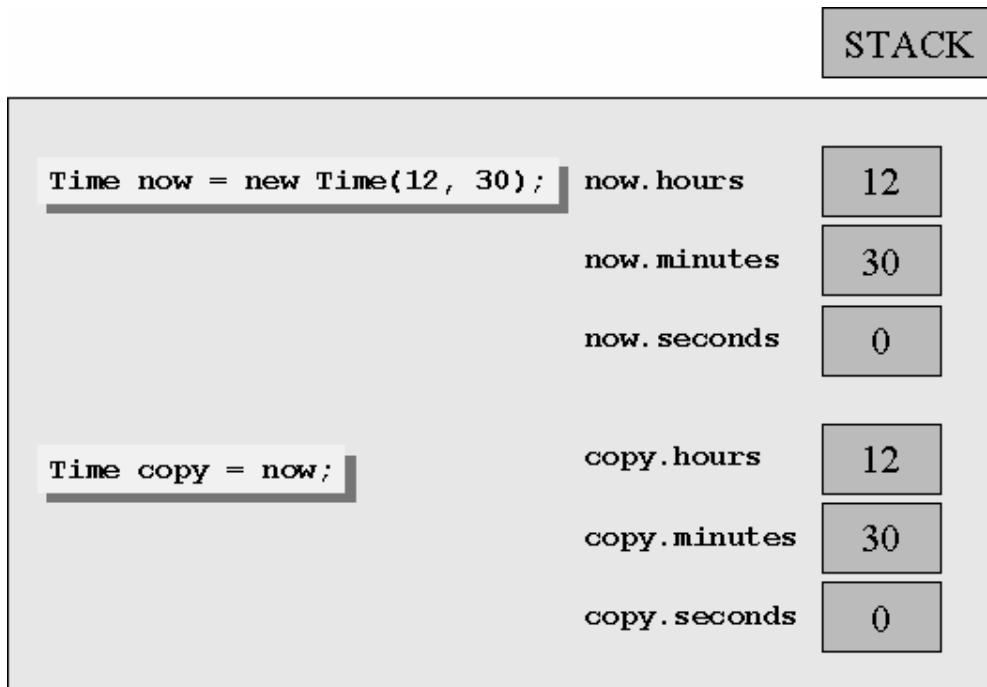
Kết quả trong stack được lưu trữ như trong hình sau:



Sao chép biến cấu trúc

Bạn được cho phép khởi gán hay gán một biến cấu trúc cho một biến cấu trúc khác nhưng chỉ nếu biến bên phải được khởi tạo hoàn chỉnh (tất cả các trường được khởi tạo). Ví dụ sau sẽ được biên dịch vì now được khởi tạo hoàn toàn:

```
Time now = new Time(12, 30);  
Time copy = now;
```



Trong ví dụ sau sẽ lỗi khi biên dịch vì now chưa được khởi tạo:

```
Time now;  
Time copy = now; // lỗi biên dịch: now chưa được gán
```

Khi bạn sao chép một cấu trúc, các trường bên trái được sao chép trực tiếp từ các trường bên phải.

3.4 Lớp Object

Trong C#, nếu bạn không chỉ rõ một lớp kế thừa từ một lớp khác thì trình biên dịch sẽ tự động hiểu rằng lớp của bạn kế thừa từ lớp Object. Nghĩa là ngoài những thuộc tính và phương thức mà bạn định nghĩa thì bạn có thể truy cập đến những phương thức protected và public của lớp Object và những phương thức này cũng có trong tất cả lớp mà bạn định nghĩa.

Một số phương thức được định nghĩa trong lớp Object là:

Phương thức	Bổ túc truy cập	Chức năng
string ToString()	public virtual	Trả về một chuỗi mô tả của đối tượng
int GetHashCode()	public virtual	Được sử dụng nếu thực thi từ điển
bool Equals(object obj)	public virtual	So sánh các thực thể của đối tượng
bool Equals(object objA, object objB)	public static	So sánh các thực thể của đối tượng
bool ReferenceEquals(object objA, object objB)	public static	So sánh hai sự tham khảo đến một đối tượng
Type GetType()	public	Trả về chi tiết kiểu của một đối tượng.
object MemberwiseClone()	protected	tạo ra một bản copy của đối tượng
void Finalize()	protected virtual	Đây là một dạng Destructor của.NET

Phương thức ToString():

ToString() là một cách tiện lợi để lấy một chuỗi mô tả đối tượng. Ví dụ:

```
int i = -50;
string str = i.ToString(); // returns "-50"
```

Thêm một ví dụ khác:

```
enum Mau {Do, Cam, Vang};
Mau m = Mau.Cam;
string str = m.ToString(); // trả về "Cam"
```

Object.ToString() được khai báo là hàm ảo. Trong C#, các kiểu dữ liệu có sẵn đã ch่อง hàm (override) phương thức ToString() giúp cho chúng ta có thể lấy được chuỗi mô tả của các kiểu đó. Nếu không định nghĩa ch่อง hàm phương thức ToString() trong lớp thì lớp đó sẽ kế thừa sự thực thi phương thức ToString() của lớp System.Object và trả về tên lớp. Nếu muốn phương thức ToString() trả về một chuỗi

chứa nội dung thông tin về giá trị của đối tượng của một lớp, chúng ta phải ghi chèn phương thức `ToString()`. Xem ví dụ sau:

```
using System;
class ViDu
{
    static void Main(string[] args)
    {
        Tien tienMat = new Tien();
        tienMat.SoLuong = 40M;
        Console.WriteLine("tienMat.ToString() trả về: " +
tienMat.ToString());
        tienMat = new TienKhac();
        tienMat.SoLuong = 40M;
        Console.WriteLine("tienMat.ToString() trả về: " +
tienMat.ToString());
        Console.ReadLine();
    }
}
class Tien
{
    private decimal soluong;

    public decimal SoLuong
    {
        get
        {
            return soluong;
        }
        set
        {
            soluong = value;
        }
    }
}
class TienKhac: Tien
{
    public override string ToString()
    {
        return "$" + SoLuong.ToString();
    }
}
```

Trong phương thức Main(), chúng ta khởi tạo đầu tiên là đối tượng Tien(), sau đó là đối tượng TienKhac(). Trong cả hai trường hợp ta đều gọi ToString() nhưng với đối tượng Tien() thì sẽ thực thi phương thức của System.Object còn với đối tượng TienKhac() thì sẽ thực thi phương thức mà chúng ta override. Kết quả sau khi chạy đoạn mã trên là:

```
tienMat. ToString() tra ve: Tien  
tienMat. ToString() tra ve: $40
```

Chương 4: Sự kế thừa

Mục đích của chương:

- Cung cấp kiến thức về sử dụng tính đóng gói trong chương trình.
- Sử dụng tính kế thừa và đa hình trong chương trình.
- Sử dụng giao tiếp (interface) trong quá trình phát triển ứng dụng.

4. 1 Các kiểu kế thừa

Sự kế thừa (Inheritance) là gì?

Kế thừa trong ngôn ngữ lập trình liên quan sự phân lớp, đó chính là mối quan hệ giữa các lớp. Ví dụ, cá voi hay ngựa là những động vật có vú. Cá voi và ngựa có mọi đặc trưng mà động vật có vú có nhưng nó cũng có riêng những đặc trưng của từng loài (ngựa có móng còn cá voi thì không).

Trong C#, bạn có thể mô hình điều này bằng cách tạo hai lớp, một lớp tên DongVat và một lớp tên Ngua và khai báo Ngua kế thừa từ DongVatCoVu. Sự kế thừa sẽ mô hình rằng có một quan hệ và mô tả sự kiện rằng tất cả ngựa là động vật có vú. Tương tự, bạn có thể khai báo một lớp tên CaHeo cũng kế thừa từ DongVatCoVu. Những đặc trưng chung (như ngày sinh, cân nặng) được đặt trong lớp DongVatCoVu. Thuộc tính như là móng hay vây được đặt tương ứng trong lớp Ngua và CaHeo.

Lớp cơ sở (base) và lớp dẫn xuất (Derived)

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class Lớp_Dẫn_Xuất: Lớp_Cơ_Sở {  
    ...  
}
```

Lớp dẫn xuất kế thừa từ lớp cơ sở. Trong C# chỉ cho một lớp kế thừa từ một lớp khác duy nhất, nó không cho dẫn xuất từ hai hay nhiều lớp. Trừ khi lớp dẫn xuất được khai báo là sealed, bạn vẫn có thể khai báo một lớp kế thừa từ lớp dẫn xuất với cùng cú pháp:

```
class Lớp_Con_Dẫn_Xuất: Lớp_Dẫn_Xuất {  
    ...  
}
```

Theo cách này bạn có thể tạo một phân cấp kế thừa.

Giả sử chúng ta viết một trình quản lý các đối tượng hình học. Trong đó chúng ta quản lý các đối tượng như là hình tam giác, hình vuông, hình tròn... Chúng ta có thể khai báo một lớp DoiTuongHinhHoc trong đó định nghĩa phương thức TinhDienTich và Ve như sau:

```
public class DoiTuongHinhHoc  
{
```

```

        public float X;
        public float Y;
        public DoiTuongHinhHoc() {}
        public DoiTuongHinhHoc(float x, float y) {}
        public double TinhDienTich();
        public void Ve();
    }

```

Chúng ta có thể định nghĩa các lớp HinTron, HinVuong, HinTamGiac kế thừa từ lớp DoiTuongHinhHoc như sau:

```

public class HinTron: DoiTuongHinhHoc
{
...
}

public class HinTamGiac: DoiTuongHinhHoc
{
...
}

public class HinVuong: DoiTuongHinhHoc
{
...
}

```

Đối tượng System.Object là lớp gốc của tất cả các lớp. Tất cả các lớp dẫn xuất ngầm định từ lớp System.Object. Nếu bạn thực thi lớp DoiTuongHinhHoc như trên:

Trình biên dịch sẽ tự động viết lại theo mã sau:

```

class DoiTuongHinhHoc: System.Object
{
    public DoiTuongHinhHoc()
    {
        ...
    }
    ...
}

```

Nghĩa là tất cả các lớp bạn định nghĩa kế thừa tất cả các đặc trưng của lớp System.Object. Điều này bao gồm phương thức ToString để chuyển một đối tượng sang một chuỗi.

Trong C#, quan hệ chuyên biệt hóa thông thường được thực hiện thông qua sự kế thừa. Đây không phải là cách duy nhất để thực hiện sự kế thừa. Tuy nhiên nó là cách phổ biến và tự nhiên để thực hiện quan hệ này.

Chúng ta nói HinTron kế thừa từ DoiTuongHinhHoc có nghĩa nó là một DoiTuongHinhHoc chuyên biệt. DoiTuongHinhHoc được gọi là một lớp cơ sở (base)

và HinTron gọi là lớp dẫn xuất(derived). Thật vậy, HinTron kế thừa những đặc trưng và hành vi của lớp DoiTuongHinhHoc và rồi chuyên biệt hóa những nhu cầu riêng của nó.

4. 2 Thực thi sự kế thừa

Trong C# chúng ta có thể tạo một lớp dẫn xuất bằng cách thêm dấu hai chấm theo sau tên của lớp dẫn xuất cùng với tên của lớp cơ sở.

```
public class HinTron: DoiTuongHinhHoc
```

Trong khai báo trên, khai báo HinTron là lớp dẫn xuất từ DoiTuongHinhHoc. Lớp dẫn xuất kế thừa tất cả thành viên của lớp cơ sở. Lớp dẫn xuất có thể tự do thực hiện những phiên bản riêng của mình đối với các phương thức lớp cơ sở. Nó thực hiện điều này bằng cách đánh dấu phương thức mới bằng từ khóa new. Dùng từ khóa này để chỉ ra lớp dẫn xuất muốn che dấu và thay thế phương thức ở lớp cơ sở.

Ví dụ sau minh họa một lớp dẫn xuất

```
public class DoiTuongHinhHoc
{
    public float X;
    public float Y;
    public DoiTuongHinhHoc() { }
    public DoiTuongHinhHoc(float x, float y) {
        this.X = x;
        this.Y = y;
    }
    public double TinhDienTich()
    {
        return 0;
    }
    public void Ve()
    {
        Console.WriteLine("Ve tai vi tri {0} {1} ", X, Y);
    }
}
public class HinTron:DoiTuongHinhHoc
{
    public HinTron() {}
    public HinTron(float x,float y):base(x,y)
    {
    }
    public new void Ve()
    {
        base.Ve();
    }
}
```

```

    }
}

class ViDu
{
    static void Main()
    {
        DoiTuongHinhHoc h = new DoiTuongHinhHoc(10, 10);
        h.Ve();
        HinhTron r = new HinhTron(20, 20);
        r.Ve();
    }
}

```

Gọi phương thức tạo lập của lớp cơ sở

Trong ví dụ trên lớp HinhTron dẫn xuất từ lớp DoiTuongHinhHoc và có phương thức tạo lập riêng của nó với hai tham số. Phương thức tạo lập của lớp HinhTron gọi phương thức tạo lập của lớp cha bằng cách đặt dấu “:” sau danh sách tham số và gọi lớp cơ sở với từ khóa base:

```

public HinhTron(float x, float y) : base(x, y) // một cách khác để gọi
phương thức tạo lập của lớp cơ sở

```

Bởi vì các lớp không thể thừa phương thức tạo lập của lớp cơ sở nên một lớp dẫn xuất phải thực thi phương thức tạo lập riêng của mình và chúng có thể sử dụng phương thức tạo lập của lớp cơ sở theo cách rõ ràng như trên

Chú ý trong ví dụ trên lớp dẫn xuất HinhTron thực thi một phiên bản mới của phương thức Ve();

```

public new void Ve();

```

Từ khoá new chỉ ra rằng người lập trình muốn tạo ra một phiên bản mới của phương thức này trong lớp dẫn xuất.

Nếu lớp cơ sở có một phương thức tạo lập có thể truy xuất mặc định. Khi phương thức tạo lập của lớp dẫn xuất không yêu cầu phương thức tạo lập của lớp cơ sở, khi đó phương thức tạo lập mặc định của lớp cơ sở được gọi theo kiểu không tường minh. Tuy nhiên nếu lớp cơ sở không có một phương thức tạo lập mặc định, các lớp dẫn xuất phải gọi rõ ràng một phương thức tạo lập cơ sở sử dụng từ khóa base.

Gọi phương thức lớp cơ sở

Trong ví dụ trên phương thức Ve() của lớp HinhTron ẩn và thay thế phương thức Ve() của lớp cơ sở. Lúc chúng ta gọi phương thức Ve() trong một đối tượng kiểu HinhTron thì phương thức HinhTron.Ve() được gọi chứ không phải phương thức HinhTron () trong lớp DoiTuongHinhHoc. Tuy nhiên, trong lớp HinhTron ta vẫn có thể gọi phương thức Ve() của lớp DoiTuongHinhHoc thông qua từ khóa base.

4. 3 Từ khóa hỗ trợ

Hiển thị của lớp và các thành viên của lớp có thể được hạn chế bằng cách khai báo kiểu truy xuất như public, private, protected, internal, protected internal.

Các lớp và các thành viên của nó có thể được thiết kế theo nhiều mức truy xuất khác nhau. Nếu một thành viên của lớp có nhiều mức truy xuất khác nhau, mức hạn chế nhất được chọn. Thật vậy, nếu chúng ta định nghĩa một lớp ViDu như sau:

```
public class ViDu
{
    ...
    protected int giaTri;
}
```

Thì truy xuất đối với biến giaTri từ các đối tượng là protected mặc dù mức truy xuất của lớp ViDu là public.

4. 4 Đa hình (polymorphism)

Hai khía cạnh mạnh nhất của kế thừa đó là khả năng sử dụng lại mã và đa hình (polymorphism). Poly có nghĩa là nhiều và morph nghĩa là hình thức. Thật vậy đa hình chính là khả năng sử dụng nhiều hình thức của một kiểu mà không cần quan đến chi tiết của nó.

Tạo kiểu đa hình

Bởi vì HìnhTron là-một DoiTuongHinhHoc và HìnhVuong là-một DoiTuongHinhHoc. Đây là những quan hệ kế thừa. Chúng ta mong muốn có thể sử dụng một trong hai kiểu này trong tình huống chúng ta gọi DoiTuongHinhHoc. Ví dụ, một mảng giữ một tập tất cả các thể hiện của DoiTuongHinhHoc. Chúng ta muốn tính tổng diện tích của tất cả các hình có trong mảng bằng cách lấy từng phần tử trong mảng và cộng vào tổng diện tích. Trong trường hợp này chúng ta không cần biết đối tượng trong danh sách là HìnhTron hay HìnhVuong, chúng ta chỉ lấy ra một phần tử và yêu cầu nó tự tính diện tích. Chúng ta sẽ xử lý tất cả các đối tượng DoiTuongHinhHoc theo đặc trưng đa hình.

Tạo phương thức đa hình

Để tạo phương thức hỗ trợ đa hình, chúng ta cần đánh dấu dùng từ khóa virtual trong lớp cơ sở của nó. Ví dụ để chỉ ra phương thức Ve() trong lớp DoiTuongHinhHoc là đa hình chúng ta dùng từ khóa theo khai báo sau:

```
public virtual void Ve( )
```

Bây giờ các lớp dẫn xuất tự do thực hiện các phiên bản riêng của lớp Ve(). Để thực hiện điều này chúng ta đơn giản viết đè lên phương thức ảo của lớp cơ sở dùng từ khóa override trong lớp dẫn xuất HìnhTron. Ví dụ:

```
public override void Ve( )
{
    base.Ve( ); // gọi hàm vẽ của sổ của lớp cha
    Console.WriteLine ("Hình Tron");
}
```

Tương tự chúng ta có thể thực hiện điều này cho lớp HìnhVuong.

Hàm Main() của chương trình có thể được viết lại như sau

```
DoiTuongHinhHoc h = new DoiTuongHinhHoc (1, 2);
HinhTron r = new HinhTron (3, 4);
HinhVuong b = new HinhVuong (5, 6);
h.Ve();
r.Ve();
b.Ve();
```

Chương trình có thể thực hiện như ta mong muốn. Đối tượng Ve được gọi cho mỗi đối tượng. Tuy nhiên không có tính đa hình được sử dụng ở đây. Muốn sử dụng đa hình chúng ta phải khai báo sau:

```
DoiTuongHinhHoc [] a = new DoiTuongHinhHoc [3];
a[0] = new DoiTuongHinhHoc (1, 2);
a[1] = new HinhVuong(3, 4);
a[2] = new HinhTron(5, 6);
```

Điều gì xảy ra khi ta gọi phương thức DrawWindow()?

Ví dụ hoàn chỉnh như sau:

```
using System;
using System.Text;
namespace ConsoleApplication3
{
    public class DoiTuongHinhHoc
    {
        public float X;
        public float Y;
        public DoiTuongHinhHoc() { }
        public DoiTuongHinhHoc(float x, float y)
        {
            this.X = x;
            this.Y = y;
        }
        public virtual float TinhDienTich()
        {
            return 0;
        }
        public virtual void Ve()
        {
            Console.WriteLine("Ve tai vi tri {0} {1} ", X, Y);
        }
    }
}
```

```

public class HinhTron : DoiTuongHinhHoc
{
    float bk = 0;
    public HinhTron() { }
    public HinhTron(float x, float y, float r)
        : base(x, y)
    {
        bk = r;
    }
    public override void Ve()
    {
        base.Ve();
        Console.WriteLine("Duong tron co ban kinh " + bk);
    }
    public override float TinhDienTich()
    {
        return (float)Math.PI * bk * bk;
    }
}
public class HinhVuong : DoiTuongHinhHoc
{
    float canh = 0;
    public HinhVuong() { }
    public HinhVuong(float x, float y, float r)
        : base(x, y)
    {
        canh = r;
    }
    public override void Ve()
    {
        base.Ve();
        Console.WriteLine("Hinh vuong co canh " + canh);
    }
    public override float TinhDienTich()
    {
        return canh * canh;
    }
}
class ViDu
{

```

```

static void Main()
{
    DoiTuongHinhHoc[] a = { new HinhTron(1, 1, 2), new HinhVuong(10,
10, 4), new DoiTuongHinhHoc(5, 5) };

    float tongDT = 0;
    for (int i = 0; i < a.Length; i++)
        tongDT += a[i].TinhDienTich();
    Console.WriteLine("Tong dien tich cac hinh la " + tongDT);
    for (int i = 0; i < a.Length; i++)
        a[i].Ve();
}
}
}

```

Phân biệt giữa từ khóa new và override

Trong C#, quyết định của lập trình viên nhằm ghi đè một phương thức ảo của lớp cơ bản được thực hiện rõ ràng qua từ khóa override. Điều này cho phép chúng ta tạo ra một phiên bản mới trong mã chúng ta. Những thay đổi trong mã của lớp cơ sở không phá vỡ những đoạn mã trong lớp dẫn xuất.

Trong C# một hàm ảo luôn được xem là gốc của các hàm ảo gọi đi. Thật vậy, khi C# tìm một phương thức ảo, nó không tìm kiếm trong phân cấp kế thừa. Nếu một hàm ảo được đưa vào trong lớp DoiTuongHinhHoc, hành vi thực thi của lớp HinhTron vẫn không thay đổi.

Để tránh việc khai báo phương thức ảo bị chồng chéo, chúng ta có thể sử dụng từ khóa new để chỉ ra không có phương thức ghi đè lên phương thức ảo của lớp cơ sở.

Lớp trừu tượng (abstract)

Mỗi lớp con của DoiTuongHinhHoc nên thực thi phương pháp Ve() của riêng nó, nhưng các lớp dẫn xuất không bắt buộc thực hiện điều này trong đoạn chương trình trên. Để yêu cầu tất cả các lớp dẫn xuất thực thi một phương pháp của lớp cơ sở chúng ta cần chỉ ra phương thức đó là trừu tượng (abstract).

Phương thức được khai báo trừu tượng không cần định nghĩa hay khai báo nội dung thực thi của nó. Nó chỉ nhằm tạo một tên phương thức và đánh dấu rằng nó phải được thực thi trong tất cả các lớp dẫn xuất từ nó.

Chúng ta dùng từ khóa abstract để định nghĩa phương thức trừu tượng:

```
abstract public void Ve();
```

Ví dụ sử dụng phương pháp và lớp trừu tượng

```

using System;
using System.Text;
namespace ConsoleApplication3
{
    public abstract class DoiTuongHinhHoc
    {
    }
}

```

```

{
    public float X;
    public float Y;
    public DoiTuongHinhHoc() { }
    public DoiTuongHinhHoc(float x, float y)
    {
        this.X = x;
        this.Y = y;
    }
    public abstract float TinhDienTich();
    public abstract void Ve();

}

public class HinhTron : DoiTuongHinhHoc
{
    float bk = 0;
    public HinhTron() { }
    public HinhTron(float x, float y, float r)
        : base(x, y)
    {
        bk = r;
    }
    public override void Ve()
    {
        Console.WriteLine("Duong tron co ban kinh " + bk);
    }
    public override float TinhDienTich()
    {
        return (float)Math.PI * bk * bk;
    }
}

public class HinhVuong : DoiTuongHinhHoc
{
    float canh = 0;
    public HinhVuong() { }
    public HinhVuong(float x, float y, float r)
        : base(x, y)
    {
        canh = r;
    }
}

```

```

public override void Ve()
{
    Console.WriteLine("Hinh vuong co canh " + canh);
}

public override float TinhDienTich()
{
    return canh * canh;
}

class ViDu
{
    static void Main()
    {
        DoiTuongHinhHoc[] a = { new HinhTron(1, 1, 2), new HinhVuong(10,
10, 4), new HinhTron(5, 5,5) };

        float tongDT = 0;
        for (int i = 0; i < a.Length; i++)
            tongDT += a[i].TinhDienTich();
        Console.WriteLine("Tong dien tich cac hinh la " + tongDT);
        for (int i = 0; i < a.Length; i++)
            a[i].Ve();
    }
}

```

Giới hạn của trừu tượng

Mặc dù thiết kế trừu tượng Ve() bắt buộc tất cả các lớp dẫn xuất thực thi phương thức trừu tượng của nó. Điều này sẽ gây khó khăn nếu lớp dẫn xuất từ lớp HinhTron không muốn thực thi phương thức Ve();

Lớp Sealed

Ngược lại của abstract là sealed. Dùng từ khóa sealed, một lớp có thể không cho phép các lớp khác dẫn xuất từ nó.

Gốc của tất cả các lớp là Object

Tất cả các lớp trong C# đều được dẫn xuất từ lớp Object. Lớp gốc là lớp trên nhất trong cây phân cấp kế thừa. Trong C#, lớp gốc là Object. Object cung cấp một số các phương thức mà các lớp con có thể ghi đè. Chúng gồm các phương thức sau:

Phương thức	Ý nghĩa
Equal()	Kiểm tra hai đối tượng có tương đương nhau không

GetHashCode()	Cho phép đối tượng cung cấp hàm Hash riêng để sử dụng trong kiểu tập hợp.
GetType()	Cung cấp truy xuất đến kiểu đối tượng.
ToString()	Cung cấp chuỗi biểu diễn đối tượng.
Finalize()	Xóa đối tượng trong bộ nhớ.
MemberwiseClone()	Tạo copy của đối tượng.

Ví dụ sau minh họa sử dụng phương thức ToString() trong lớp Object cũng như trong các kiểu dữ liệu cơ bản kế thừa từ Object:

```

using System;
public class ViDu
{
    public ViDu (int val)
    {
        giatri = val;
    }
    public override string ToString( )
    {
        return giatri.ToString( );
    }
    private int giatri;
}
public class KiemTra
{
    static void Main( )
    {
        int i = 5;
        Console.WriteLine("Giá trị của biến i là: " + i));
        ViDu s = new ViDu(7);
        Console.WriteLine("Giá trị của biến s là " + s );
    }
}

```

Trong trường hợp trên hàm ToString() của đối tượng s và I sẽ được gọi tự động.

4.5 Giao diện

C# hỗ trợ giao diện (interface). Khi kế thừa từ một giao diện, một lớp dẫn xuất sẽ phải thực thi tất cả các thành viên trong giao diện. Chúng ta sẽ minh họa về giao diện thông qua việc giới thiệu một giao diện đã được Microsoft định nghĩa, System.IDisposable. IDisposable chứa một phương thức Dispose() dùng để xoá mã.

```

public interface IHinhHoc
{
    float DienTich();
}

```

Trên ví dụ trên ta thấy việc khai báo một giao diện làm việc giống như việc khai báo một lớp trừu tượng, nhưng nó không cho phép thực thi bất kỳ một thành phần nào của giao diện. Một giao diện chỉ có thể chứa những khai báo của phương thức, thuộc tính, chỉ mục và sự kiện.

Bạn không thể khởi tạo một giao, nó không có phương thức tạo lập hay các trường và nó không cho phép chứa các phương thức ghi chép.

Nó cũng không cho phép khai báo những bộ từ trên các thành phần trong khi định nghĩa một giao diện. Các thành phần bên trong một giao diện luôn luôn là public và không thể khai báo virtual hay static.

Định nghĩa và thực thi một giao diện

Giả sử chúng ta viết mã để cho phép các máy điện toán chuyển khoản qua lại giữa các tài khoản. Có nhiều công ty tham gia vào hệ thống này tài khoản và đều đồng ý với nhau là phải thực thi một giao diện ITaiKhoan có các phương thức nạp hay rút tiền và thuộc tính trả về số tài khoản.

Để bắt đầu, ta định nghĩa giao diện ITaiKhoan:

```

public interface ITaiKhoan
{
    void GuiTien(decimal soLuong);
    bool RutTien(decimal soLuong);
    decimal SoTien
    {
        get;
    }
}

```

Chú ý: Tên của giao diện thường phải có ký tự I đứng đầu để nhận biết đó là một giao diện.

Khai báo lớp TaiKhoanTietKiem kế thừa từ ITaiKhoan

```

public class TaiKhoanTietKiem: ITaiKhoan
{
    private decimal soTien;

    public void GuiTien (decimal soLuong)
    {
        soTien += soLuong;
    }

    public bool RutTien(decimal soLuong)
    {
        if (soTien >= soLuong)
        {

```

```

        soTien -= soLuong;
        return true;
    }
    Console.WriteLine("Rut tien bi loi. ");
    return false;
}

public decimal SoTien
{
    get
    {
        return soTien;
    }
}
public override string ToString()
{
    return String.Format("Tien tiet kiem: so tien = {0, 6:C}",
soTien);
}
}

```

Trong ví dụ trên chúng ta duy trì một trường private soTien và điều chỉnh số lượng này khi tiền được nạp hay rút. Chú ý chúng ta xuất ra một thông báo lỗi khi thao tác rút tiền không thành công vì thiếu số tiền trong tài khoản.

Xét dòng lệnh sau:

```
public class TaiKhoanTietKiem: ITaiKhoan
```

Chúng ta khai báo lớp TaiKhoanTietKiem kế thừa giao diện ITaiKhoan. Ta cũng có thể khai báo một lớp kế thừa từ một lớp nào đó và từ nhiều giao diện theo cú pháp như sau như sau:

```
public class Lop_Dan_Xuat: Lop_Co_So, IGiaoDien1, IGiaoDien12
```

Thừa kế từ ITaiKhoan có nghĩa là TaiKhoanTietKiem lấy tất cả các thành phần của ITaiKhoan nhưng nó không thể sử dụng các phương thức đó nếu nó không định nghĩa lại các hành động của từng phương thức. Nếu bỏ quên một phương thức nào thì trình biên dịch sẽ báo lỗi.

Để minh họa cho các lớp khác nhau có thể thực thi cùng một giao diện ta xét ví dụ về một lớp khác là TaiKhoanVang.

```

public class TaiKhoanVang: ITaiKhoan
{
    // vv
}
```

Chúng ta không mô tả chi tiết lớp TaiKhoanVang bởi vì về cơ bản nó giống hệt lớp TaiKhoanTietKiem. Điểm nhấn mạnh ở đây là TaiKhoanVang có thể rút tiền thậm chí trong tài khoản của họ không còn tiền.

Chúng ta có phương thức main():

```

class ViDu
{
    static void Main()
    {
        ITaiKhoan tk1 = new TaiKhoanTietKiem();
        ITaiKhoan tk2 = new TaiKhoanVang();
        tk1.GuiTien(200);
        tk1.RutTien(100);
        Console.WriteLine(tk1.ToString());
        tk2.GuiTien(500);
        tk2.RutTien(600);
        tk2.RutTien(100);
        Console.WriteLine(tk2.ToString());
    }
}

```

Kết quả xuất ra là:

```

Tien tiet kiem: so tien = £100. 00
Rut tien bi loi.
Tien tiet kiem: so tien = £400. 00

```

Chúng ta có thể trỏ đến bất kỳ lớp nào thực thi cùng một giao diện. Nhưng chúng ta chỉ được gọi những phương thức là thành phần của giao diện thông qua sự tham khảo đến giao diện này. Nếu muốn gọi những phương thức mà không là thành phần trong giao diện thì ta phải tham khảo đến những kiểu thích hợp. Như ví dụ trên ta có thể thực thi phương thức ToString() mặc dù nó không là thành phần được khai báo trong giao diện ITaiKhoan bởi vì nó là thành phần của System.Object.

Một giao diện có thể tham khảo đến bất kỳ lớp nào thực thi giao diện đó.

Ví dụ ta có một mảng kiểu giao diện nào đó thì các phần tử của mảng có thể tham khảo đến bất kỳ lớp nào thực thi giao diện đó:

```

ITaiKhoan[] taiKhoan = new ITaiKhoan[2];
taiKhoan[0] = new TaiKhoanTietKiem();
taiKhoan[1] = new TaiKhoanVang();

```

Thùa kế giao diện

C# cho phép một giao diện có thể kế thừa các giao diện khác. Khi một giao diện kế thừa một giao diện khác thì nó có thể chứa tất cả các phương thức định nghĩa trong giao diện cha và những phương thức của nó định nghĩa. Ví dụ ta tạo ra một giao diện mới kế thừa giao diện ITaiKhoan:

```

public interface ITaiKhoanChuyenTien: ITaiKhoan
{

```

```
        bool ChuyenDen(ITaiKhoan dich, decimal soLuong);  
    }
```

Như vậy giao diện ITaiKhoanChuyenTien phải có tất cả các phương thức trong giao diện ITaiKhoan và phương thức ChuyenDen.

Chúng ta sẽ minh họa ITaiKhoanChuyenTien bằng một ví dụ bên dưới về một current account. Lớp TaiKhoanHienThoi được định nghĩa gần giống hệt với các lớp TaiKhoanTietKiem và TaiKhoanVang:

```
public class TaiKhoanHienThoi: ITaiKhoanChuyenTien  
{  
    private decimal soTien;  
    public void GuiTien(decimal soLuong)  
    {  
        soTien += soLuong;  
    }  
    public bool RutTien(decimal soLuong)  
    {  
        if (soTien >= soLuong)  
        {  
            soTien -= soLuong;  
            return true;  
        }  
        Console.WriteLine("Rut tien loi. ");  
        return false;  
    }  
    public decimal SoTien  
    {  
        get  
        {  
            return soTien;  
        }  
    }  
    public bool ChuyenDen(ITaiKhoan dich, decimal soLuong)  
    {  
        bool kq;  
        if ((kq = RutTien(soLuong)) == true)  
            dich.GuiTien(soLuong);  
        return kq;  
    }  
    public override string ToString()  
    {
```

```
        return String.Format("Tai khoan hien thoi: so tien = {0, 6:C}",  
soTien);  
    }  
}  
  
static void Main()  
{  
    ITaiKhoan tk1 = new TaiKhoanTietKiem();  
    ITaiKhoanChuyenTien tk2 = new TaiKhoanHienThoi();  
    tk1.GuiTien(200);  
    tk2.GuiTien(500);  
    tk2.ChuyenDen(tk1, 100);  
    Console.WriteLine(tk1.ToString());  
    Console.WriteLine(tk2.ToString());  
}
```

Khi thực thi đoạn mã trên bạn sẽ thấy kết quả như sau:

```
TaiKhoanHienThoi  
Tien tiet kiem: so tien = £300. 00  
Tai khoan hien thoi: so tien = £400. 00
```

Chương 5: Toán tử và chuyển kiểu

Mục đích của chương:

- Sử dụng toán tử trong các biểu thức, độ ưu tiên thực hiện các phép toán.
- Định nghĩa các toán tử trong lớp.
- Chuyển đổi kiểu tường minh và không tường minh.

5. 1 Toán tử

Tìm hiểu toán tử

Chúng ta dùng toán tử để kết hợp các toán hạng với nhau trong một biểu thức. Mỗi toán tử có riêng ngữ nghĩa dựa trên kiểu mà chúng làm việc. Ví dụ, toán tử + nghĩa là cộng đối với kiểu dữ liệu số nhưng lại là nối chuỗi đối với kiểu dữ liệu string.

Mỗi toán tử có một độ ưu tiên khác nhau. Ví dụ toán tử * có độ ưu tiên cao hơn toán tử +.

Mỗi toán tử có một kết hợp với nó từ trái qua phải hay từ phải qua trái. Ví dụ, toán tử = là kết hợp phải (từ phải qua trái) vì $a = b = c$ cùng $a = (b=c)$.

Toán tử một ngôi là toán tử chỉ có một toán hạng. Ví dụ, toán tử tăng (++) là một toán tử một ngôi.

Toán tử nhị phân là toán tử có hai toán hạng. Ví dụ toán tử * là toán tử nhị phân.

Ràng buộc toán tử

C# cho phép bạn nạp chồng hầu hết các toán tử hiện tại theo kiểu riêng của bạn. Khi thực hiện điều này toán tử của bạn thực thi theo các luật sau:

- Bạn không thể thay đổi độ ưu tiên và sự kết hợp của một toán tử. Độ ưu tiên và sự kết hợp dựa trên kí hiệu toán tử (ví dụ +), không dựa trên kiểu (ví dụ int) mà kí hiệu toán tử đang được sử dụng. Vì vậy, biểu thức $a + b * c$ thì giống $a + (b*c)$ không liên quan đến kiểu a, b, c.
- Bạn không thể thay đổi số số hạng của toán tử. Ví dụ, * là toán tử nhị phân. Nếu bạn khai báo một toán tử * cho kiểu của bạn, nó phải là toán tử nhị phân.
- Bạn không thể tạo kí hiệu toán tử mới. Ví dụ, bạn không thể tạo một kí hiệu toán tử mới như là ** để tính mũ số của một số với một số khác. Bạn phải tạo một phương thức để thực hiện điều đó.
- Bạn không thể thay đổi ý nghĩa của các toán tử khi bạn dùng nó cho các kiểu dựng sẵn. Ví dụ, biểu thức $1 + 2$ có một ý nghĩa định nghĩa trước và bạn không được phép ghi đè nghĩa này. Nếu bạn làm điều này, mọi thứ trở nên phức tạp.
- Có một số kí hiệu toán tử mà bạn không thể quá tải. Ví dụ, bạn không thể quá tải toán tử “.”.

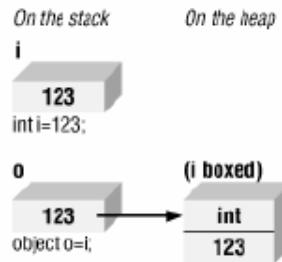
5. 2 Chuyển kiểu an toàn

Kiểu đóng hộp(Boxing) và mở hộp (unboxing)

Đóng hộp và mở hộp là quá trình xử lý cho phép các kiểu giá trị được xử lý như là kiểu tham chiếu. Đóng hộp chính là chuyển giá trị vào trong một đối tượng và mở hộp trả về kiểu giá trị của đối tượng đó. Đó là cách cho phép chúng ta gọi phương thức `ToString()` trên số nguyên trong các ví dụ trên.

Đóng hộp là dạng không tường minh

Đóng hộp chuyển một kiểu giá trị đến một kiểu Object không tường minh. Đóng hộp một giá trị sẽ thực hiện cấp phát một thẻ hiện của Object và chép giá trị vào trong thẻ hiện của đối tượng mới như trong hình vẽ sau:



Ví dụ minh họa:

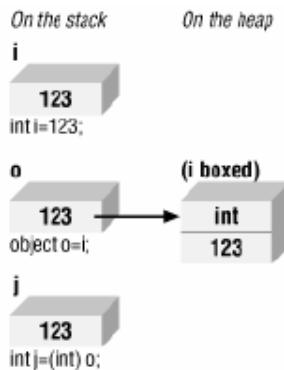
```
using System;
class DongHop
{
    public static void Main( )
    {
        int i = 123;
        object o = i;
        Console.WriteLine("Gia tri cua doi tuong la = {0}", o);
    }
}
```

Mở hộp tường minh

Để lấy giá trị từ đối tượng, mở hộp trả về một kiểu giá trị, chúng ta phải thực hiện mở hộp rõ ràng. Chúng ta nên thực hiện hai bước sau:

- Đảm bảo thẻ hiện của đối tượng là một giá trị có cùng kiểu với kiểu được đưa vào khi đóng hộp.
- Chép giá trị từ thẻ hiện của đối tượng đến biến kiểu giá trị.

Minh họa như trong hình vẽ sau:



Để thực hiện được quá trình mở hộp, đối tượng bị mở hộp phải là một tham chiếu đến đối tượng mà chúng được tạo bởi đóng. Minh họa mở và đóng hộp như sau:

```
using System;
public class UnboxingTest
{
    public static void Main( )
    {
        int i = 123;
        // đóng hộp
        object o = i;
        // mở hộp phải được khai báo tường minh
        int j = (int) o;
        Console.WriteLine("j: {0}", j);
    }
}
```

5.3 Quá tải toán tử

Trong C#, toán tử (operator) được định nghĩa là một phương thức tĩnh và phải trả về kết quả. Khi chúng ta tạo một toán tử cho lớp chúng ta có thể quá tải toán tử đó tương tự như cách chúng ta quá tải hàm. Ví dụ, quá tải toán tử cộng chúng ta có thể viết như sau:

```
public static PhanSo operator+ (PhanSo lhs, PhanSo rhs)
```

Tham số đầu tiên lhs là toán hạng bên trái và tham số thứ hai rhs là toán hạng bên phải.

Dùng từ khóa operator theo sau bởi toán tử để chỉ quá tải toán tử. Chúng ta có thể tạo các toán tử như các toán tử logic, toán tử tương đương (==).

Ví dụ sau minh họa dùng quá tải toán tử cho ứng dụng xây dựng kiểu dữ liệu phân số:

```
using System;
namespace ConsoleApplication6. PhanSo1
{
```

```

public class PhanSo
{
    public int tu;
    private int _mau;
    public int mau
    {
        get
        {
            return _mau;
        }
        set
        {
            if (value == 0)
            {
                Console.WriteLine("Mau so phai khac 0");
                _mau = 1;
            }
            else
                _mau = value;
        }
    }
    //Các phương thức tạo lập cho phân số
    public PhanSo()
    {
    }
    public PhanSo(int tu, int mau)
    {
        this.tu = tu;
        this.mau = mau;
    }
    public PhanSo(PhanSo a)
    {
        this.tu = a.tu;
        this.mau = a.mau;
    }
    //Các phương thức nhập phân số
    public void Nhap()
    {
        //Nhập giá trị cho tử
        Console.Ghi(" Nhap tu = ");

```

```

        this.tu = int.Parse(Console.ReadLine());
        //Nhập giá trị cho mẫu
        Console.Ghi(" Nhập mau = ");
        this.mau = int.Parse(Console.ReadLine());
    }

    //Chuyển phân số thành một chuỗi
    public override string ToString()
    {
        return tu + "/" + mau;
    }

    //Cộng hai phân số
    public static PhanSo operator +(PhanSo a, PhanSo b)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu * b.mau + a.mau * b.tu;
        kq.mau = a.mau * b.mau;
        return kq.RutGon();
    }

    //Cộng phân số với số x... . a/b = (a+x) / (b+x)
    public static PhanSo operator +(PhanSo a, int x)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu + x;
        kq.mau = a.mau + x;
        return kq.RutGon();
    }

    public static PhanSo operator -(PhanSo a, PhanSo b)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu * b.mau - a.mau * b.tu;
        kq.mau = a.mau * b.mau;
        return kq.RutGon();
    }

    public static PhanSo operator -(PhanSo a, int x)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu - x;
        kq.mau = a.mau - x;
        return kq.RutGon();
    }
}

```

```

public static PhanSo operator *(PhanSo a, PhanSo b)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu * b.tu;
    kq.mau = a.mau * b.mau;
    return kq.RutGon();
}

//nhân phân số a với một số nguyên x a/b * x = a/b*x
public static PhanSo operator *(PhanSo a, int x)
{
    PPhanSo tam = new PhanSo((PhanSo)x);
    return a * tam();
}

public static PhanSo operator /(PhanSo a, PhanSo b)
{
    return a * b.NghichDao();
}

//chia phân số a với một số nguyên x a/b chia x = a/b*x
public static PhanSo operator /(PhanSo a, int x)
{
    PhanSo tam = new PhanSo((PhanSo)x);
    return a * tam.NghichDao();
}

public static PhanSo operator ++(PhanSo a)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu + 1;
    kq.mau = a.mau + 1;
    return kq.RutGon();
}

public static PhanSo operator --(PhanSo a)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu--;
    kq.mau = a.mau--;
    return kq;
}

```

```

//Chuyển số nguyên thành phân số
public static implicit operator PhanSo(int a)
{
    PhanSo kq = new PhanSo();
    kq.tu = a;
    kq.mau = 1;
    return kq;
}

//Chuyển phân số thành số thực
public static explicit operator double(PhanSo a)
{
    return ((double)a.tu / (double)a.mau);
}

public PhanSo RutGon()
{
    int us = this.USCLN();
    PhanSo tam = new PhanSo();
    tam.tu = this.tu / us;
    tam.mau = this.mau / us;
    return tam;
}

private int USCLN()
{
    int a, b;
    //Kiểm tra trường hợp nếu tử và mẫu âm
    a = tu > 0 ? tu: -tu;
    b = mau > 0 ? mau: -mau;
    while (a != b)
    {
        if (a > b) a = a - b;
        if (b > a) b = b - a;
    }
    return a;
}

public PhanSo NghichDao()
{
    PhanSo kq = new PhanSo();
    kq.tu = this.mau;
    kq.mau = this.tu;
    return kq;
}

```

```

        }
    }

using System;
namespace ConsoleApplication6.PhanSo1
{
    public class KiemTra
    {
        public KiemTra()
        {
        }

        public static void KiemTraPhuongThucCong()
        {
            PhanSo ps1 = new PhanSo(-16, 5);
            Console.WriteLine("Phan so thu nhat 1" + ps1);
            PhanSo ps2 = new PhanSo(20, 2);
            Console.WriteLine("Phan so thu nhat 2 " + ps2);
            Console.WriteLine("Cong phan so mot va hai " + (ps1 + ps2));
        }
    }
}

```

5.4 Chuyển kiểu do người dùng định nghĩa

Cú pháp khai báo cho toán tử này tương tự như khai báo quá tải toán tử. Toán tử phải là public và static. Đây là một toán tử chuyển kiểu cho phép một đối tượng Gio chuyển kiểu không tường minh thành int:

```

struct Gio
{
    public static implicit operator int (Gio tu)
    {
        return this.giatri;
    }
    private int giatri;
}

```

Khi khai báo toán tử chuyển kiểu riêng, chúng ta phải chỉ ra toán tử chuyển kiểu tường minh hay không tường minh bằng cách dùng từ khóa implicit và explicit. Ví dụ, toán tử chuyển kiểu Gio thành int phần trên là không tường minh, nghĩa là trình biên dịch C# có thể sử dụng nó không tường minh:

```

class ViDu
{
    public static void PhuongThuc(int ts) {...}
}

```

```
public static void Main()
{
    Gio trua = new Gio(12);
    ViDu.PhuongThuc(trua); //chuyển kiểu không tường minh từ Gio
    sang int
}
```

Nếu toán tử chuyển kiểu được khai báo explicit, ví dụ trước sẽ gây ra lỗi vì chuyển kiểu tường minh được yêu cầu:

```
ViDu.PhuongThuc(int)trua;// chuyển kiểu tường minh từ Gio sang int
```

Khi nào bạn nên khai báo toán tử chuyển kiểu tường minh hay không tường minh?
Nếu chuyển kiểu là an toàn, không gây mất thông tin, và không thể tạo ra ngoại lệ bạn
có thể định nghĩa nó là chuyển kiểu không tường minh, ngược lại nên khai báo chuyển
kiểu tường minh.

Chương 6: Sự ủy nhiệm, sự kiện và quản lý lỗi

Mục đích của chương:

- Gọi phương thức gần giống với con trỏ hàm trong C++.
- Khai báo và sử dụng sự kiện.
- Cơ chế quản lý lỗi, bẫy lỗi trong chương trình.

6. 1 Sự ủy nhiệm (delegate)

Nhiều đoạn mã mà chúng ta đã viết trong các bài tập trước được giả định thực thi tuần tự. Nhưng thỉnh thoảng bạn thấy cần thiết ngắt luồng thực thi hiện tại và thực hiện các nhiệm vụ quan trọng hơn. Lúc nhiệm vụ hoàn thành, chương trình có thể tiếp tục quay lại nơi nó gọi thực thi nhiệm vụ.

Để điều khiển các ứng dụng dạng này, CLR phải cung cấp 2 điều: một phương tiện để chỉ ra có một cái gì đó khẩn cấp đã xảy ra và một điều chỉ ra phương thức nào nên chạy khi nó xảy ra. Đây là mục đích của ủy nhiệm và sự kiện (event).

Khai báo và sử dụng delegate

Một delegate là một con trỏ đến một phương thức. Một delegate trông giống và cư xử như phương thức lúc nó được gọi. Tuy nhiên, khi bạn gọi một delegate, CLR thực thi phương thức mà delegate tham chiếu tới. Bạn có thể thay đổi tham chiếu của delegate một cách tự động vì vậy mà để gọi một delegate chạy các phương thức khác nhau mỗi lần nó thực thi.

Khai báo delegate

```
delegate kiểu trả về Tên(danh_sách_tham_số);
```

Sử dụng delegate

```
using System;

delegate void HamDeGoi();

class ViDu
{
    public void PhuongThuc()
    {
        Console.WriteLine("Phuong thuc");
    }

    public static void staticMethod()
    {
        Console.WriteLine("Phuong thuc tinh");
    }
}
```

```

class MainClass
{
    static void Main()
    {
        ViDu t = new ViDu();
        HamDeGoi hamDelegate;
        hamDelegate = t.PhuongThuc;
        hamDelegate += ViDu.staticMethod;
        hamDelegate += t.PhuongThuc;
        hamDelegate += ViDu.staticMethod;
        hamDelegate();
    }
}

```

Sử dụng Delegate như là con trỏ hàm

```

using System;
class MainClass
{
    delegate int Hamdelegate(string s);
    static void Main(string[] args)
    {
        Hamdelegate del1 = new Hamdelegate(PhuongThuc);
        Hamdelegate del2 = new Hamdelegate(PhuongThuc2);
        string str = "Xin Chao";
        del1(str);
        del2(str);
    }
    static int PhuongThuc(string s)
    {
        Console.WriteLine("Phuong Thuc");
        return 0;
    }
    static int PhuongThuc2(string s)
    {
        Console.WriteLine("Phuong Thuc 2");
        return 0;
    }
}

```

6. 2 Sự kiện (Event)

Mặc dù delegate cho phép gọi bất kỳ số phương thức gián tiếp, nhưng chúng ta phải gọi các delegate tường minh. Trong nhiều trường hợp, sẽ rất hữu ích nếu delegate chạy tự động khi có một cái gì đó đặc biệt xảy ra. Trong .Net cho phép chúng ta định nghĩa và bẫy các hành động đặc biệt và sắp xếp để gọi một delegate để quản lý tình huống.

Nhiều lớp trong .NET đưa ra sự kiện. Hầu hết các điều khiển đặt trên form dùng sự kiện cho phép chạy mã chương trình tương ứng như: khi người dùng nhấp chuột vào nút lệnh hay nhập gì đó trên một trường. Bạn có thể định nghĩa riêng sự kiện.

Khai báo sự kiện

Bạn khai báo một sự kiện trong một lớp để nhắm đến một hành động là nguồn của sự kiện. Nguồn sự kiện thường là một lớp quan sát môi trường và phát triển sự kiện khi có một dấu hiệu đặc biệt xảy ra. Một sự kiện chứa danh sách các phương thức để gọi khi sự kiện được tạo.

Vì sự kiện thường được dùng với delegate nên kiểu của sự kiện phải là delegate và bắt đầu với từ khóa event.

```
public delegate void BatCongTac(bool state);  
public class CongTac  
{  
    public event BatCongTac OnBatCongTac;
```

Trong ví dụ trên, sự kiện OnBatCongTac cung cấp một giao tiếp cho phép theo dõi trạng thái của công tác để thực hiện thao tác tắt mở bóng đèn tương ứng. Một sự kiện quản lý riêng các delegate của nó và không cần quản lý thủ công biến delegate.

Gán một sự kiện

Giống delegate, sự kiện có thể được gán với toán tử += .

```
c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
```

Bỏ gán sự kiện

Bạn có thể dùng toán tử -= để xóa phương thức từ tập hợp delegate bên trong.

Tạo một sự kiện

Một sự kiện có thể được tạo giống delegate bằng cách gọi nó như là một phương thức. Khi bạn gọi một sự kiện, tất cả các delegate gắn với nó được gọi tuần tự. Ví dụ:

```
public class CongTac  
{  
    public event BatCongTac OnBatCongTac;  
    public bool state;  
    public void KhiBatCongTac()  
    {  
        if(OnBatCongTac!=null)  
        {  
            OnBatCongTac(state);  
        }  
    }  
}
```

```

        state = state ? false: true;
    }
}
}

```

Kiểm tra null là cần thiết vì trường sự kiện ngầm định là null và nó khác null khi ta thực hiện gán một phương thức dùng toán tử `+=`. Nếu bạn tạo một sự kiện null, bạn sẽ có một ngoại lệ `NullReferenceException`. Chúng ta cũng phải truyền các tham số tương ứng khi bạn tạo sự kiện ứng với các delegate đã định nghĩa.

Ví dụ sử dụng sự kiện và ủy nhiệm:

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
namespace SuDungDelegate
{
    public delegate void BatCongTac(bool state);
    public class CongTac
    {
        public event BatCongTac OnBatCongTac;
        public bool state;
        public void KhiBatCongTac()
        {
            OnBatCongTac(state);
            state = state ? false: true;
        }
    }
    public class BongDen
    {
        public void TrangThaiDen(bool state)
        {
            if (state)
                Console.WriteLine("Den Sang");
            else
                Console.WriteLine("Den tat");
        }
    }
    public class TV
    {
        public void TrangThaiTV(bool state)
        {

```

```

        if (state)
            Console.WriteLine("Mo TV");
        else
            Console.WriteLine("Tat tivi");
    }
}

class Program
{
    static CongTac c = new CongTac();
    static BongDen d = new BongDen();
    static TV t = new TV();
    static void Main(string []args)
    {
        c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
        c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
        c.KhiBatCongTac();
        c.KhiBatCongTac();
    }
}
}

```

6.3 Quản lý lỗi và biệt lệ

Không gì quan trọng bằng một đoạn mã tốt, chương trình của bạn phải luôn có khả năng xử lí những lỗi có thể xảy ra. Ví dụ, trong một quy trình xử lí phức tạp, một đoạn mã nào đó phát hiện nó không được phép đọc một tập tin, hoặc trong khi nó đang gửi yêu cầu đến mạng thì mạng rớt. Trong những tình huống ngoại lệ (exception) như vậy, không có đủ phương thức dù chỉ đơn giản là trả về một mã lỗi tương đương. C# có những kĩ thuật tốt để xử lí những loại tình huống này, bằng cơ chế xử lí biệt lệ (exception handling).

Những lớp biệt lệ của lớp cơ sở

Trong C#, một biệt lệ là một đối tượng được tạo ra (hoặc được ném) khi một trạng thái lỗi biệt lệ cụ thể xuất hiện. Những đối tượng này chứa đựng những thông tin giúp ích cho việc truy ngược lại vấn đề. Mặc dù chúng ta có thể tự tạo ra những lớp biệt lệ riêng, .NET cũng cung cấp cho chúng ta nhiều lớp biệt lệ được định nghĩa trước.

Những lớp ngoại lệ cơ bản

Một phần của lớp ngoại lệ nằm trong IOException và những lớp dẫn xuất từ IOException. Một phần nằm trong System.IO nó liên quan đến việc đọc và viết dữ liệu lên tập tin. Nói chung, không có namespace cụ thể cho biệt lệ; những lớp biệt lệ nên được đặt trong lớp nơi mà chúng có thể được sinh ra ngoại lệ, vì lí do đó những ngoại lệ có liên quan đến IO thì nằm trong namespace System.IO, chúng ta có thể tìm thấy những lớp biệt lệ nằm trong một vài namespace lớp cơ sở.

Những lớp ngoại lệ có đặc điểm chung là đều dẫn xuất từ System.Exception. Có 2 lớp quan trọng trong hệ thống các lớp được dẫn xuất từ System.Exception là:

- System.SystemException: sử dụng cho những biệt lệ thường xuyên được phát sinh trong thời gian chạy của .NET, hoặc là những lỗi chung thường được sinh ra bởi hầu hết những ứng dụng. Ví dụ như là StackOverflowException được phát sinh trong thời gian chạy .NET nếu nó thăm dò thấy Stack đầy. Chúng ta có thể chọn kiểu phát sinh của ArgumentException, hoặc là phát sinh từ một lớp con của nó trong chương trình. Những lớp con của System.SystemException có thể trình bày những lỗi nghiêm trọng và không nghiêm trọng.
- System.ApplicationException : đây là một lớp quan trọng bởi vì nó được dùng cho các lớp biệt lệ được định nghĩa bởi những hàng thứ ba. Do đó, nếu bạn định nghĩa bất kì biệt lệ nào bao phủ trạng thái lỗi của ứng dụng, bạn nên dẫn xuất một cách trực tiếp hay gián tiếp từ System.ApplicationException.

Đón bắt biệt lệ

Giả sử chúng ta có những đối tượng biệt lệ có giá trị, vậy làm thế nào chúng ta có thể sử dụng chúng trong đoạn mã để bẫy những trạng thái lỗi? Để có thể giải quyết điều này trong C# bạn thường là phải chia chương trình của bạn thành những khối thuộc 3 kiểu khác nhau:

- Khối try chứa đựng đoạn mã mà có dạng là một phần thao tác bình thường trong chương trình của bạn, nhưng đoạn mã này có thể gặp phải một vài trạng thái lỗi nghiêm trọng.
- Khối catch chứa đựng đoạn mã mà giải quyết những trạng thái lỗi nghiêm trọng trong đoạn try.
- Khối finally chứa đựng đoạn mã mà dọn dẹp tài nguyên hoặc làm bất kì hành động nào mà bạn thường muốn làm xong vào cuối khối try hay catch... điều quan trọng phải hiểu rằng khối finally được thực thi dù có hay không có bất kì biệt lệ nào được ném ra. Bởi vì mục tiêu chính của khối finally là chứa đựng đoạn mã rõ ràng mà luôn được thực thi, trình biên dịch sẽ báo lỗi nếu bạn đặt một lệnh return bên trong khối finally.

Cú pháp C# được sử dụng để thể hiện tất cả điều này cụ thể như sau:

```
try
{
    // mã cho việc thực thi bình thường
}
catch
{
    // xử lý lỗi
}
finally
{
    // dọn dẹp
}
```

Có một vài điều có thể thay đổi trong cú pháp trên:

- Ta có thể bỏ qua khối finally.

- Ta có thể cung cấp nhiều khối catch mà ta muốn xử lý những kiểu lỗi khác nhau.
- Ta có thể bỏ qua khối catch, trong trường hợp cú pháp phục vụ không xác định biệt lẻ, nhưng phải đảm bảo rằng mã trong khối finally sẽ được thực thi khi việc thực thi rời khỏi khối try.

Nhưng điều này đặt ra một câu hỏi: nếu đoạn mã đang chạy trong khối try, làm thế nào nó biết chuyển đến khối catch nếu một lỗi xuất hiện? nếu một lỗi được phát sinh, mã chương trình sẽ ném ra một biệt lẻ. Nói cách khác, nó chỉ ra một lớp đối tượng biệt lẻ và ném nó:

```
throw new OverflowException();
```

Ở đây chúng ta có một thể hiện của đối tượng biệt lẻ của lớp OverflowException. Ngay khi máy tính gặp một câu lệnh throw bên trong khối try, nó ngay lập tức tìm khối catch kết hợp với khối try, nó xác định khối catch đúng bởi việc kiểm tra lớp biệt lẻ nào mà khối catch kết hợp với. Ví dụ, khi đối tượng OverflowException được ném ra việc thực thi sẽ nhảy vào khối catch sau:

```
catch (OverflowException e)
{
}
```

Nói cách khác, máy tính sẽ tìm khối catch mà chỉ định một thể hiện lớp biệt lẻ phù hợp trong cùng một lớp (hoặc của lớp cơ sở).

Giả sử, lúc xem xét đối số, có 2 lỗi nghiêm trọng có thể xảy ra trong nó: lỗi tràn và mảng ngoài biên. Giả sử rằng đoạn mã của chúng ta chưa đựng hai biến luận lý. Overflow và OutOfBounds, mà chỉ định liệu trạng thái lỗi này có tồn tại không. Chúng ta đã thấy lớp biệt lẻ đã định nghĩa trước đây tồn tại để chỉ định tràn (OverflowException); tương tự một lớp IndexOutOfBoundsException tồn tại để xử lý mảng ngoài biên.

Bây giờ hãy nhìn vào khối try sau:

```
try
{
    // mã cho thực thi bình thường
    if (Overflow == true)
        throw new OverflowException();
    // xử lý nhiều hơn
    if (OutOfBounds == true)
        throw new IndexOutOfBoundsException();
    // hoặc tiếp tục xử lý bình thường
}
catch (OverflowException e)
{
    // xử lý lỗi cho trạng thái lỗi tràn
}
catch (IndexOutOfBoundsException e)
{
    // xử lý lỗi cho trạng thái lỗi chỉ mục nằm ngoài vùng
}
finally
{
    // dọn dẹp
}
```

Thực thi nhiều khối catch

Xem ví dụ sau, nó lặp lại việc hỏi người sử dụng nhập vào một số và xuất giá trị của nó. Tuy nhiên vì mục đích của ví dụ, chúng ta sẽ yêu cầu số cần nhập phải trong khoảng từ 0 đến 5 hoặc là chương trình sẽ không thể xử lý số một cách chính xác. Do đó chúng ta sẽ ném ra một biệt lệ nếu người sử dụng nhập một thứ gì đó ngoài vùng này.

Chương trình sẽ tiếp tục hỏi số cho đến khi người sử dụng nhấn enter mà không gõ bất kì phím gì khác.

```
using System;

public class ViDu
{
    public static void Main()
    {
        string str;
        while ( true )
        {
            try
            {
                Console.WriteLine("Nhập số từ 1 đến 5" +
                    "(hay nhấn enter để thoát)> ");
                str = Console.ReadLine();
                if (str == "")
                    break;
                int index = Convert.ToInt32(str);
                if (index < 0 || index > 5)
                    throw new IndexOutOfRangeException(
                        "Bạn nhập " + str);
                Console.WriteLine("Bạn vừa nhập là " + index);
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("Exception: " +
                    "Bạn nhập số giữa 0 và 5. " + e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine(
                    "Ngoài lỗi được ném ra là: " + e.Message);
            }
            catch
            {
                Console.WriteLine("Một lỗi bất kỳ khác xuất hiện");
            }
            finally
            {
                Console.WriteLine("Cám ơn");
            }
        }
    }
}
```

Chapter 7: Quản lý bộ nhớ và con trỏ

Mục đích của chương:

- Tìm hiểu về heap và stack và cách thức lưu trữ các biến tham trị và tham chiếu
- Khai báo các khối mã “không an toàn” để truy xuất bộ nhớ trực tiếp.
- Cơ chế tổ chức và hoạt động của cơ chế thu gom rác trong .Net.

7. 1 Quản lý bộ nhớ

Một trong những ưu điểm của C# là chúng ta không cần quan tâm về việc quản lý bộ nhớ bên dưới vì điều này đã được bộ gom rác (garbage collector) của C# làm rồi. Mặc dù vậy nếu ta muốn viết các đoạn mã tốt, có hiệu suất cao, ta cần tìm hiểu về cách quản lý bộ nhớ bên dưới.

Giá trị các kiểu dữ liệu

Windows dùng hệ thống địa chỉ ảo (virtual addressing) để ánh xạ từ địa chỉ bộ nhớ đến vị trí thực sự trong bộ nhớ vật lý hoặc trên đĩa được quản lý phía sau Windows. Kết quả là mỗi ứng dụng trên nền xử lí 32-bit thấy được 4GB bộ nhớ, không cần biết bộ nhớ vật lý thực sự có kích thước bao nhiêu (nền xử lí 64 bit thì bộ nhớ này lớn hơn 4GB bộ nhớ này được gọi là không gian địa chỉ ảo (virtual address space) hay bộ nhớ ảo (virtual memory). Để đơn giản ta gọi nó là bộ nhớ mỗi vùng nhớ từ 4GB này được đánh số từ 0. Nếu ta muốn chỉ định một giá trị lưu trữ trên 1 phần cụ thể trong bộ nhớ, ta cần cung cấp số đại diện cho vùng nhớ này. Trong ngôn ngữ cấp cao như là C#, VB, C++, Java... một trong những cách mà trình biên dịch làm là chuyển đổi tên đọc được (ví dụ tên biến) thành địa chỉ vùng nhớ mà bộ xử lí hiểu. 4GB bộ nhớ này thực sự chứa tất cả các phần của chương trình bao gồm mã thực thi và nội dung của biến được dùng khi chương trình chạy. Bất kì thư viện liên kết động (DLL-Dynamic Link Library) được gọi sẽ nằm trong cùng không gian địa chỉ này, mỗi mục của mã hoặc dữ liệu sẽ có vùng định nghĩa riêng.

Stack là một vùng nhớ nơi giá trị kiểu dữ liệu được lưu. Khi ta gọi phương thức, stack cũng được dùng để sao chép các tham số được truyền.

Các kiểu dữ liệu tham chiếu

Chúng ta có thể dùng một số phương thức để cấp phát vùng nhớ để lưu trữ dữ liệu, và giữ cho dữ liệu còn nguyên giá trị ngay cả khi phương thức kết thúc. Điều này có thể làm với kiểu tham chiếu.

Xét đoạn mã sau:

```
void ThucHien()
{
    KhachHang kh;
    kh = new KhachHang();
    KhachHang m = new KhachHang60();
```

```
}
```

trong đoạn mã này ta giả sử gồm 2 lớp KhachHang và KhachHang60. Ta khai báo một tham chiếu gọi là kh, được cấp phát trong Stack nhưng nhớ rằng đây là một tham chiếu, không phải là một thể hiện KhachHang. Không gian mà tham chiếu kh chiếm là 4 byte. Ta cần 4 byte để có thể lưu một số nguyên giá trị từ 0 đến 4GB

sau đó ta có dòng:

```
kh = new KhachHang();
```

Dòng này đầu tiên cấp phát vùng nhớ trong heap để lưu trữ một thể hiện của KhachHang. Sau đó nó đặt biến kh để lưu địa chỉ của vùng nhớ được cấp phát, đồng thời nó cũng gọi phương thức tạo lập KhachHang() để khởi tạo một thể hiện lớp.

Thể hiện của KhachHang không đặt trong stack mà sẽ đặt trong heap. Ta không biết chính xác một thể hiện KhachHang chiếm bao nhiêu byte, ta xem nó là 32 byte. 32 byte này chứa các trường thể hiện của KhachHang cùng vài thông tin mà .NET dùng để xác định danh tính và quản lý các thể hiện lớp của nó. .NET tìm trong heap khối 32 byte còn trống, giả sử nằm ở địa chỉ 200000, và tham chiếu kh nằm ở vị trí 799996-799999 trên stack. Không như stack, bộ nhớ trong heap được cấp phát theo chiều từ dưới lên, vì thế không gian trống được tìm thấy phía trên không gian đã dùng.

Dòng kế tiếp thực hiện tương tự, ngoại trừ không gian trên stack cho tham chiếu m cần được cấp phát vào cùng lúc cấp phát cho m trên heap:

```
KhachHang m = new KhachHang60();
```

4 byte được cấp phát trên stack cho tham chiếu m lưu ở địa chỉ 799992-799995. Trong khi thể hiện m sẽ được cấp phát từ vị trí 200032 đi lên trên heap.

.NET runtime sẽ cần duy trì thông tin về trạng thái của heap, thông tin này cũng cần được cập nhật khi dữ liệu mới được thêm vào heap. Để minh họa điều này, ta hãy xem điều gì xảy ra khi ta thoát phương thức và tham chiếu kh và m nằm ngoài phạm vi. Theo cách làm việc bình thường thì con trỏ stack sẽ được tăng để những biến này không còn tồn tại nữa. Tuy nhiên các biến này chỉ lưu địa chỉ, không phải thể hiện lớp, dữ liệu của nó vẫn nằm trong heap. Ta có thể thiết lập các biến tham chiếu khác nhau để trỏ đến cùng một đối tượng- nghĩa là những đối tượng đó sẽ có giá trị sau khi tham chiếu kh và m nằm ngoài phạm vi và sự khác biệt quan trọng giữa stack và heap: đối tượng được cấp phát liên tiếp trên heap, thời gian sống không lồng nhau.

Khi ta giải thích cách hoạt động trên heap, ta nhấn mạnh rằng chỉ stack mới có khả năng lồng thời gian sống của các biến. Vậy khi thời gian sống của các tham chiếu nằm ngoài phạm vi thì heap làm việc như thế nào trên các biến này? Câu trả lời là bộ thu gom rác sẽ làm điều này. Khi bộ gom rác chạy, nó sẽ gỡ bỏ tất cả những đối tượng từ heap mà không còn tham chiếu nữa. Ngay sau khi nó làm điều này, heap sẽ có các đối tượng rải rác trên nó, nằm lẫn với các khoảng trống.

Bộ gom rác không để heap trong tình trạng này, ngay khi nó giải phóng tất cả các đối tượng có thể, nó sẽ di chuyển tất cả chúng trở về cuối của heap để thành một khối liên tục lại. Tất nhiên khi những đối tượng này được di chuyển tất cả các tham chiếu của nó đều được cập nhật lại.

7. 2 Giải phóng tài nguyên

Chu kỳ sống và thời gian của một đối tượng

Điều gì xảy ra khi bạn tạo và hủy một đối tượng.

Bạn tạo đối tượng như sau:

```
SinhVien a = new SinhVien(); // a là kiểu tham chiếu
```

Tiến trình tạo một đối tượng gồm hai giai đoạn. Đầu tiên, hoạt động new cấp phát bộ nhớ thô từ heap. Bạn không thể điều khiển giai đoạn này khi đối tượng được tạo. Thứ hai, hoạt động new chuyển bộ nhớ thô vào trong một đối tượng; nó phải khởi tạo đối tượng. Bạn có thể điều khiển giai đoạn này dùng phương thức tạo lập.

Khi bạn đã tạo đối tượng, bạn có thể truy xuất thành viên của nó dùng toán tử “.”. Ví dụ:

```
a.Ten = "Nguyen Van A";
```

Bạn có thể khai báo biến tham chiếu đến cùng đối tượng:

```
SinhVien ref = a;
```

Bạn có thể tạo bao nhiêu tham chiếu đến đối tượng cũng được. CLR sẽ theo dõi tất cả những tham chiếu này. Nếu biến a biến mất (ngoài phạm vi), những biến khác (như là ref) vẫn tồn tại. Vì vậy thời gian sống của một đối tượng không bị ràng buộc vào một biến tham chiếu cụ thể. Một đối tượng chỉ có thể bị hủy khi tất cả các tham chiếu đến nó biến mất.

Tương tự như tạo, hủy đối tượng cũng gồm 2 giai đoạn. Đầu tiên, bạn phải thực hiện một số dọn dẹp được viết trong phương thức hủy. Thứ hai, bộ nhớ thô được trả lại cho heap, bạn không thể kiểm soát giai đoạn này. Quá trình hủy một đối tượng và trả bộ nhớ lại heap được biết là cơ chế thu dọn (garbage collection).

Viết phương thức hủy

Bạn có thể sử dụng một phương thức hủy để thực hiện việc dọn dẹp khi đối tượng cần được gom rác. Bạn viết dấu “~” theo sau bởi tên lớp. Trong ví dụ sau đếm số lần hiện của lớp bằng cách tăng biến tĩnh count trong phương thức tạo lập và giảm biến này trong phương thức hủy:

```
class ViDu
{
    public ViDu()
    {
        this.soTheHien++;
    }

    ~ViDu()
    {
        this.soTheHien--;
    }

    public static int SoTheHien()
```

```

    {
        return this.soTheHien;
    }
    ...
    private static int soTheHien = 0;
}

```

Một số hạn chế khi sử dụng phương thức tạo lập:

- Bạn không thể khai báo một phương thức hủy trong cấu trúc vì cấu trúc là một kiểu giá trị lưu trên stack không phải trên heap nên không cần sử dụng cơ chế thu dọn:

```

struct ViDu
{
    ~ViDu() {...} // lỗi biên dịch
}

```

- Bạn không thể khai báo chỉ định truy xuất (như là public) cho phương thức hủy bởi vì bạn không thể gọi phương thức hủy, nó được gọi bởi cơ chế thu dọn.
public ~ViDu() {...} // lỗi biên dịch
- Bạn không được khai báo phương thức hủy với tham số.

```
~ViDu(int parameter) {...} //lỗi biên dịch
```

- Trình biên dịch tự động dịch phương thức hủy thành phương thức Object.Finalize.

```

class ViDu
{
    ~ViDu() {...}
}

```

Dịch thành:

```

class ViDu
{
    protected override void Finalize()
    {
        try {...}
        finally { base.Finalize(); }
    }
}

```

Tại sao sử dụng cơ chế thu dọn

Trong C#, bạn không thể chủ động hủy đối tượng, lý do tại sao các nhà thiết kế C# cấm bạn thực hiện điều này:

- Bạn quên xóa đối tượng, điều này có nghĩa là phương thức hủy không được chạy và bộ nhớ sẽ không được thu hồi lại cho heap và bạn có thể nhanh chóng cạn kiệt bộ nhớ.

- Khi bạn đang cố xóa một đối tượng đang hoạt động. Nhớ rằng đối tượng là kiểu tham chiếu. Nếu một lớp giữ một tham chiếu đến một đối tượng đã bị hủy. Nó có thể tham chiếu đối tượng không sử dụng hay tham chiếu đến một đối tượng hoàn toàn khác trong cùng vùng nhớ.
- Bạn muốn xóa cùng đối tượng nhiều lần. Điều này có thể rất tai hại dựa trên mã trong phương thức hủy.

Cơ chế thu dọn có nhiệm vụ hủy đối tượng cho bạn và nó đảm bảo các vấn đề sau:

- Mỗi đối tượng sẽ bị hủy và khi chương trình kết thúc tất cả các đối tượng đang tồn tại sẽ bị hủy.
- Mỗi đối tượng chính xác bị hủy một lần.
- Mỗi đối tượng bị hủy khi không còn tham chiếu đến nó.

Những đảm bảo này rất hữu ích và tiện lợi cho các lập trình viên, bạn chỉ tập trung vào phần logic của chương trình.

Một đặc trưng quan trọng nữa của cơ chế thu dọn là phương thức hủy sẽ không chạy cho tới khi đối tượng là rác cần được thu dọn. Nếu bạn viết một phương thức tạo lập, bạn sẽ biết nó sẽ chạy nhưng không biết lúc nào chạy?

Cách thức làm việc của cơ chế thu dọn

Cơ chế thu dọn chạy trên một tiến trình riêng của nó và chỉ chạy ở một thời gian nào đó (thông thường khi ứng dụng chạy đến cuối phương thức). Khi nó chạy, các tiến trình khác đang chạy trong ứng dụng của bạn sẽ tạm thời treo, bởi vì cơ chế thu dọn cần di chuyển các đối tượng và cập nhật các tham chiếu đến các đối tượng. Cơ chế thu dọn thực hiện các bước sau:

1. Xây dựng một bản đồ của tất cả các đối tượng có thể đến được. Cơ chế thu dọn xây dựng bản đồ này rất cẩn thận vì tránh tham chiếu vòng gây ra lặp vô hạn. Bất kì đối tượng nào không có trong bản đồ này được xem là không đến được.
2. Nó kiểm tra các đối tượng không thể đến có phương thức hủy cần để chạy hay không. Nếu có nó đưa vào trong một hàng đợi đặc biệt gọi là F-reachable.
3. Nó thu hồi các đối tượng không thể đến còn lại, bằng cách di chuyển các đối tượng xuống dưới heap. Vì vậy sự phân mảnh và giải phóng bộ nhớ ở đầu heap. Khi cơ chế thu dọn di chuyển một đối tượng, nó cũng cập nhật tất cả tham chiếu đến đối tượng này.
4. Ở thời điểm này, nó cho phép các tiến trình khác chạy lại.
5. Nó thu hồi các đối tượng trong F-reachable trong một tiến trình độc lập.

7.3 Mã không an toàn

Có những trường hợp ta cần truy xuất bộ nhớ trực tiếp như khi ta muốn truy xuất vào các hàm bên ngoài (không thuộc .NET) hay tham số yêu cầu truyền vào là con trỏ, hoặc là vì ta muốn truy nhập vào nội dung bộ nhớ để sửa lỗi. Trong phần này ta sẽ xem xét cách C# đáp ứng những điều này như thế nào.

Con trỏ

Con trỏ đơn giản là một biến lưu địa chỉ như là một tham chiếu. Sự khác biệt là cú pháp C# trong tham chiếu không cho phép ta truy xuất vào địa chỉ bộ nhớ.

Ưu điểm của con trỏ:

- Cải thiện sự thực thi: cho ta biết những gì ta đang làm, đảm bảo rằng dữ liệu được truy xuất hay thao tác theo cách hiệu quả nhất - đó là lí do mà C và C++ cho phép dung con trỏ trong ngôn ngữ của mình.
- Khả năng tương thích ngược: đôi khi ta phải sử dụng lại các hàm API cho mục đích của ta. Mà các hàm API được viết bằng C, ngôn ngữ dùng con trỏ rất nhiều, nghĩa là nhiều hàm lấy con trỏ như tham số. Hoặc là các DLL do một hãng nào đó cung cấp chứa các hàm lấy con trỏ làm tham số. Trong nhiều trường hợp ta có thể viết các khai báo DLLImport theo cách tránh sử dụng con trỏ, ví dụ như dùng lớp System.IntPtr.
- Ta có thể cần tạo ra các địa chỉ vùng nhớ có giá trị cho người dùng - ví dụ nếu ta muốn phát triển một ứng dụng mà cho phép người dùng tương tác trực tiếp đến bộ nhớ, như là một debugger.

Nhược điểm:

- Cú pháp để lấy các hàm phức tạp hơn.
- Con trỏ khó sử dụng.
- Nếu không cẩn thận ta có thể viết lên các biến khác, làm tràn stack, mất thông tin, đụng độ...
- C# có thể từ chối thực thi những đoạn mã không an toàn này (đoạn mã có sử dụng con trỏ)

Ta có thể đánh dấu đoạn mã có sử dụng con trỏ bằng cách dùng từ khoá **unsafe**

Ví dụ: dùng cho hàm

```
unsafe int PhuongThuc()
{
    // mã có thể sử dụng con trỏ
}

Dùng cho lớp hay struct
unsafe class ViDu
{
    // bất kì phương thức nào trong lớp cũng có thể dùng con trỏ
}

Dùng cho một trường
class ViDu
{
    unsafe int *px; //khai báo một trường con trỏ trong lớp
}
```

Hoặc một khối mã

```

void PhuongThuc()
{
    // mã không sử dụng con trỏ
    unsafe
    {
        // Mã sử dụng con trỏ
    }
    // Mã không sử dụng con trỏ
}

```

Tuy nhiên ta không thể đánh dấu một biến cục bộ là unsafe

```

int PhuongThuc()
{
    unsafe int *pX;    // Sai
}

```

Để biên dịch các mã chứa khỏi unsafe ta dùng lệnh sau:

```
csc /unsafe Nguon.cs
```

hay

```
csc -unsafe Nguon.cs
```

Cú pháp con trỏ

```

int * pWidth, pHeight;
double *pResult;

```

Lưu ý khác với C++, kí tự * kết hợp với kiểu hơn là kết hợp với biến - nghĩa là khi ta khai báo như ở trên thì pWidth và pHeight đều là con trỏ do có * sau kiểu int, khác với C++ ta phải khai báo * cho cả hai biến trên thì cả hai mới là con trỏ.

Cách dùng * và & giống như trong C++:

- &: lấy địa chỉ
- * : lấy nội dung của địa chỉ

Ép kiểu con trỏ thành kiểu int

Vì con trỏ là một số int lưu địa chỉ nên ta có thể chuyển tường minh con trỏ thành kiểu int hay ngược lại. Ví dụ:

```

int x = 10;
int *pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int *pD = (int*)y;

```

Một lý do để ta phải ép kiểu là Console.WriteLine không có nạp chòg hàm nào nhận thông số là con trỏ do đó ta phải ép nó sang kiểu số nguyên int

```
Console.WriteLine("Dia chi la " + pX); // sai  
                                // Lỗi biên dịch  
Console.WriteLine("Dia chi la " + (uint) pX); // Đúng
```

Ép kiểu giữa những kiểu con trỏ

Ta cũng có thể chuyển đổi tường minh giữa các con trỏ trở đến một kiểu khác ví dụ:

```
byte aByte = 8;  
byte *pByte= &aByte;  
double *pDouble = (double*)pByte;
```

void Pointers

Nếu ta muốn giữ một con trỏ, nhưng không muốn đặc tả kiểu cho con trỏ ta có thể khai báo con trỏ là void:

```
void *pointerToVoid;  
pointerToVoid = (void*)pointerToInt;
```

mục đích là khi ta cần gọi các hàm API mà đòi hỏi thông số void*.

Toán tử sizeof

Lấy thông số là tên của kiểu và trả về số byte của kiểu đó ví dụ:

```
int x = sizeof(double);
```

x có giá trị là 8

Bảng kích thước kiểu:

sizeof(sbyte) = 1;	sizeof(byte) = 1;
sizeof(short) = 2;	sizeof(ushort) = 2;
sizeof(int) = 4;	sizeof(uint) = 4;
sizeof(long) = 8;	sizeof(ulong) = 8;
sizeof(char) = 2;	sizeof(float) = 4;
sizeof(double) = 8;	sizeof(bool) = 1;

Ta cũng có thể dùng sizeof cho struct nhưng không dùng được cho lớp.

Chương 8: Chuỗi, biểu thức quy tắc và tập hợp

Mục đích của chương:

- Sử dụng bộ thư viện thao tác trên chuỗi.
- Sử dụng biểu thức quy tắc trong việc kiểm tra hợp lệ dữ liệu.
- Làm việc với các cấu trúc dữ liệu động như ArrayList, HashTable...

8. 1 System.String

Trước khi kiểm tra các lớp chuỗi khác, ta sẽ xem lại nhanh những phương thức trong lớp chuỗi. System.String là lớp được thiết kế để lưu trữ chuỗi, bao gồm một số lớn các thao tác trên chuỗi. Không chỉ thế mà còn bởi vì tầm quan trọng của kiểu dữ liệu này, C# có từ khoá riêng cho nó và kết hợp với cú pháp để tạo nên cách dễ dàng trong thao tác chuỗi.

Ta có thể nối chuỗi:

```
string message1 = "Hello";
message1 += ", There";
string message2 = message1 + "!";
Trích 1 phần chuỗi dùng chỉ mục:
char char4 = message[4]; // trả về 'a'. lưu ý rằng kí tự bắt đầu tính
từ chỉ mục 0
các phương thức khác (số lược):
```

Phương thức	Mục đích
Compare	so sánh nội dung của 2 chuỗi
CompareOrdinal	giống compare nhưng không kể đến ngôn ngữ bản địa hoặc văn hoá
Format	định dạng một chuỗi chứa một giá trị khác và chỉ định cách mỗi giá trị nên được định dạng.
IndexOf	vị trí xuất hiện đầu tiên của một chuỗi con hoặc kí tự trong chuỗi
IndexOfAny	vị trí xuất hiện đầu tiên của bất kì một hoặc một tập kí tự trong chuỗi
LastIndexOf	giống indexof, nhưng tìm lần xuất hiện cuối cùng
LastIndexOfAny	giống indexofAny, nhưng tìm lần xuất hiện cuối cùng
PadLeft	canh phải chuỗi, điền chuỗi bằng cách thêm một kí tự được chỉ định lặp lại vào đầu chuỗi

PadRight	canh trái chuỗi, điền chuỗi bằng cách thêm một kí tự được chỉ định lặp lại vào cuối chuỗi
Replace	thay thế kí tự hay chuỗi con trong chuỗi với một kí tự hoặc chuỗi con khác
Split	chia chuỗi thành 1 mảng chuỗi con, ngắt bởi sự xuất hiện của một kí tự nào đó
Substring	trả về chuỗi con bắt đầu ở một vị trí chỉ định trong chuỗi.
ToLower	chuyển chuỗi thành chữ thường
ToUpper	chuyển chuỗi thành chữ in
Trim	bỏ khoảng trắng ở đầu và cuối chuỗi

Xây dựng chuỗi

Chuỗi là một lớp mạnh với nhiều phương thức hữu ích. Tuy nhiên, nó thực sự là kiểu dữ liệu cố định, nghĩa là mỗi lần ta khởi động một đối tượng chuỗi, thì đối tượng chuỗi đó không bao giờ được thay đổi. Những phương thức hoặc toán tử mà cập nhật nội dung của chuỗi thực sự là tạo ra một chuỗi mới, sao chép chuỗi cũ vào nếu cần thiết.

Ví dụ:

```
string str = "Lap trinh huong doi tuong.";
str += "Ngon ngu C#";
```

Đầu tiên lớp System.String được tạo và khởi tạo giá trị "Lap trinh huong doi tuong.". Trong thời gian thực thi .NET sẽ định vị đủ bộ nhớ trong chuỗi để chứa đoạn kí tự này và tạo ra một biến str để chuyển đến một thể hiện chuỗi.

Ở dòng tiếp theo, khi ta thêm kí tự vào, ta sẽ tạo ra một chuỗi mới với kích thước đủ để lưu trữ cả hai đoạn; đoạn gốc " Lap trinh huong doi tuong.", sẽ được sao chép vào chuỗi mới với đoạn bổ sung " Ngon ngu C#". Sau đó địa chỉ trong biến str được cập nhật, vì vậy biến sẽ trả đúng đến đối tượng chuỗi mới. Chuỗi cũ không còn được tham chiếu, không có biến nào truy cập vào nó, và vì vậy nó sẽ được bộ thu rác gỡ bỏ.

8. 2 Biểu thức quy tắc

Ngôn ngữ biểu thức chính quy là ngôn ngữ được thiết kế đặc biệt cho việc xử lý chuỗi, chứa đựng 2 đặc tính:

- Một tập mã escape cho việc xác định kiểu của các kí tự. Ta quen với việc dùng kí tự * để trình bày chuỗi con bắt kì trong biểu thức DOS. Biểu thức chính quy dùng nhiều chuỗi như thế để trình bày các mục như là 'bắt kì một kí tự', 'một từ ngắt', 'một kí tự tùy chọn', ...

- Một hệ thống cho việc nhóm những phần chuỗi con, và trả về kết quả trong suốt thao tác tìm.

Dùng biểu thức chính quy, có thể biểu diễn những thao tác ở cấp cao và phức tạp trên chuỗi. Mặc dù có thể sử dụng các phương thức System.String và System.Text.StringBuilder để làm các việc trên nhưng nếu dùng biểu thức chính quy thì mã có thể được giảm xuống còn vài dòng. Ta khởi tạo một đối tượng System.Text.RegularExpressions.RegEx, truyền vào nó chuỗi được xử lí, và một biểu thức chính quy (một chuỗi chứa đựng các lệnh trong ngôn ngữ biểu thức chính quy). Một chuỗi biểu thức chính quy nhìn giống một chuỗi bình thường nhưng có thêm một số chuỗi hoặc kí tự khác làm cho nó có ý nghĩa đặc biệt hơn. Ví dụ chuỗi \b chỉ định việc bắt đầu hay kết thúc một từ, vì thế nếu ta muốn chỉ định tìm kí tự “th” bắt đầu một từ, ta có thể tìm theo biểu thức chính quy “\bth”. Nếu muốn tìm tất cả sự xuất hiện của “th” ở cuối từ ta viết “th\b”. Tuy nhiên, biểu thức chính quy có thể phức tạp hơn thế, ví dụ điều kiện để lưu trữ phần kí tự mà tìm thấy bởi thao tác tìm kiếm.

Bảng sau thể hiện một số kí tự hoặc chuỗi escape mà ta có thể dùng

	Ý nghĩa	Ví dụ	Các mẫu sẽ so khớp
^	Bắt đầu của chuỗi nhập	^B	B, nhưng chỉ nếu kí tự đầu tiên trong chuỗi
\$	Kết thúc của chuỗi nhập	X\$	X, nhưng chỉ nếu kí tự cuối cùng trong chuỗi
.	Bất kì kí tự nào ngoại trừ kí tự xuống dòng(\n)	i. ation	isation, ization
*	Kí tự trước có thể được lặp lại 0 hoặc nhiều lần	ra*t	rt, rat, raat, raaat, ...
+	Kí tự trước có thể được lặp lại một hoặc nhiều lần	ra+t	rat, raat, raaat ..., (nhưng không rt)
?	Kí tự trước có thể được lặp lại 0 hoặc một lần	ra?t	Chỉ rt và rat
\s	Bất kì kí tự khoảng trắng	\sa	[space]a, \ta, \na (\t và \n có ý nghĩa giống như trong C#)
\S	Bất kì kí tự nào không phải là khoảng trắng	\SF	aF, rF, cF, nhưng không \tf
\b	Từ biên	ion\b	Bất kì từ kết thúc với ion
\B	bất kì vị trí nào không phải là từ biên	\BX\B	bất kì kí tự X ở giữa của một từ

Ví dụ chương trình sau dùng biểu thức quy tắc kiểm tra một chuỗi có phải là địa chỉ email hay không?

```
using System;
using System.Text.RegularExpressions;
```

```

public class MainClass{

    public static void Main(){
        Regex r = new Regex(@"\w+@(\w+\.)+\w+");
        Console.WriteLine(r.IsMatch("j@by.com"));
        Console.WriteLine(r.IsMatch("jn@b.com"));
        Console.WriteLine(r.IsMatch("o47@l.com"));
        Console.WriteLine(r.IsMatch("oop.com"));
    }
}

```

8.3 Nhóm các đối tượng

ArrayList

ArrayList giống như mảng, ngoại trừ nó có khả năng mở rộng, được đại diện bởi lớp System.Collection.ArrayList.

ArrayList cung cấp vùng nhớ để lưu trữ một số các tham chiếu đối tượng. Ta có thể thao tác trên những tham chiếu đối tượng này. Nếu ta thử thêm một đối tượng đến ArrayList hơn dung lượng cho phép của nó, thì nó sẽ tự động tăng dung lượng bằng cách cấp phát thêm vùng nhớ mới lớn đủ để giữ gấp 2 lần số phần tử của dung lượng hiện thời.

Ta có thể khởi tạo một danh sách bằng cách chỉ định dung lượng ta muốn.

Ví dụ, ta tạo ra một danh sách SinhVien:

```
ArrayList a = new ArrayList(20);
```

Nếu ta không chỉ định kích cỡ ban đầu, mặc định sẽ là 16:

```
ArrayList a = new ArrayList(); // kích cỡ là 16
```

Ta có thể thêm phần tử bằng cách dùng phương thức Add():

```
a.Add(new SinhVien());
a.Add(new SinhVien());
```

ArrayList xem tất cả các phần tử của nó như là các tham chiếu đối tượng. Nghĩa là ta có thể lưu trữ bất kỳ đối tượng nào mà ta muốn vào trong một ArrayList. Nhưng khi truy nhập đến đối tượng, ta sẽ cần ép kiểu chúng trở lại kiểu dữ liệu tương đương:

```
SinhVien x = (SinhVien)a[1];
```

Ví dụ này cũng chỉ ra ArrayList định nghĩa một chỉ mục, để ta có thể truy nhập những phần tử của nó với cấu trúc như mảng. Ta cũng có thể chèn các phần tử vào ArrayList:

```
a.Insert(1, new SinhVien()); // chèn vào vị trí 1
```

Đây là một phương thức nạp chồng, vì vậy rất có ích khi ta muốn chèn tất cả các phần tử trong một tập hợp vào ArrayList. Ta có thể bỏ một phần tử:

```
a.RemoveAt(1); // bỏ đối tượng ở vị trí 1
```

Lưu ý rằng việc thêm và bỏ một phần tử sẽ làm cho tất cả các phần tử sau phải bị thay đổi tương ứng trong bộ nhớ, thậm chí nếu cần thì có thể tái định vị toàn bộ ArrayList.

Ta có thể cập nhật hoặc đọc dung lượng qua thuộc tính:

```
a.Capacity = 30;
```

Tuy nhiên việc thay đổi dung lượng đó sẽ làm cho toàn bộ ArrayList được tái định vị đến một khối bộ nhớ mới với dung lượng được yêu cầu.

Để biết số phần tử thực sự trong ArrayList ta dùng thuộc tính Count:

```
int num = a.Count;
```

Một ArrayList có thể thực sự hữu ích nếu ta cần xây dựng một mảng đối tượng mà ta không biết kích cỡ của mảng sẽ là bao nhiêu. Trong trường hợp đó, ta có thể xây dựng 'mảng' trong ArrayList, sau đó sao chép ArrayList trở lại mảng khi ta hoàn thành xong. Ví dụ nếu mảng được xem là một tham số của phương thức, ta chỉ phải sao chép tham chiếu chứ không phải đối tượng:

```
SinhVien [] ds = new SinhVien[a.Count];  
for (int i=0 ; i< a.Count ; i++)  
    ds[i] = (SinhVien)a[i];
```

Tập hợp (Collection)

Ý tưởng của Collection là nó trình bày một tập các đối tượng mà ta có thể truy xuất bằng việc lặp qua từng phần tử. Cụ thể là một tập đối tượng mà ta có thể truy nhập sử dụng vòng lặp foreach. Ví dụ:

```
foreach (SinhVien v in a)  
{  
    //Thực hiện thao tác nào đó cho SinhVien v  
}
```

Ta xem biến a là một tập hợp, khả năng để dùng vòng lặp foreach là mục đích chính của collection.

Collection là gì ?

Một đối tượng là một collection nếu nó có thể cung cấp một tham chiếu đến một đối tượng có liên quan, được biết đến như là enumerator, mà có thể duyệt qua từng mục trong collection. Đặc biệt hơn, một collection phải thực thi một interface System.Collections.IEnumerable. IEnumerable định nghĩa chỉ một phương thức như sau:

```
interface IEnumerable  
{  
    IEnumerator GetEnumerator();  
}
```

Mục đích của GetEnumerator() là để trả về đối tượng enumerator. Khi ta tập hợp những đoạn mã trên đối tượng enumerator được mong đợi để thực thi một interface, System.Collections. IEnumerator.

Ngoài ra còn có một interface khác, ICollection, được dẫn xuất từ IEnumerable. Những collection phức tạp hơn sẽ thực thi interface này. Bên cạnh GetEnumerator(),

nó thực thi một thuộc tính trả về trực tiếp số phần tử trong collection. Nó cũng hỗ trợ việc sao chép một collection đến một mảng và cung cấp thông tin đặc tả nếu đó là một luồng an toàn.

IEnumerable có cấu trúc sau:

```
interface IEnumerable
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

IEnumerable làm việc như sau: đối tượng thực thi nên được kết hợp với một collection cụ thể. Khi đối tượng này được khởi động lần đầu tiên, nó chưa trả đến bất kỳ một phần tử nào trong collection, và ta phải gọi MoveNext() để nó chuyển đến phần tử đầu tiên trong collection. Ta có thể nhận phần tử này với thuộc tính Current. Current trả về một tham chiếu đối tượng, vì thế ta sẽ ép kiểu nó về kiểu đối tượng mà ta muốn tìm trong Collection. Ta có thể làm bắt cứ điều gì ta muốn với đối tượng đó sau đó di chuyển đến mục tiếp theo trong collection bằng cách gọi MoveNext() lần nữa. Ta lặp lại cho đến khi hết các mục trong collection- khi current trả về null. Nếu muốn ta có thể quay trở về vị trí đầu trong collection bằng cách gọi Reset(). Lưu ý rằng Reset() thực sự trả về trước khi bắt đầu collection, vì thế nếu muốn di chuyển đến phần tử đầu tiên ta phải gọi MoveNext().

Một collection là một kiểu cơ bản của nhóm đối tượng. Bởi vì nó không cho phép ta thêm hoặc bỏ mục trong nhóm. Tất cả ta có thể làm là nhận các mục theo một thứ tự được quyết định bởi collection và kiểm tra chúng. Thậm chí, ta không thể thay thế hoặc cập nhật mục vì thuộc tính current là chỉ đọc. Hầu như cách dùng thường nhất của collection là cho ta sự thuận tiện trong cú pháp của lặp foreach.

Mảng cũng là một collection, nhưng lệnh foreach sử dụng trong collection làm việc tốt hơn trên mảng.

Ta có thể xem vòng lặp foreach trong C# là cú pháp ngắn trong việc viết:

```
{
    IEnumerable enumerator = a.GetEnumerator();
    SinhVien v;
    enumerator.MoveNext();
    while ( (v = (SinhVien)enumerator.Current) != null)
    {
        //Thao tac sinh vien v
        enumerator.MoveNext();
    }
}
```

Một khía cạnh quan trọng của collection là bộ đếm được trả về như là một đối tượng riêng biệt. Lý do là để cho phép khả năng có nhiều hơn một bộ đếm có thể áp dụng đồng thời trong cùng collection.

Từ điển (Dictionary)

Từ điển trình bày một cấu trúc dữ liệu rất phức tạp mà cho phép ta truy nhập vào các phần tử dựa trên một khoá nào đó, mà có thể là kiểu dữ liệu bất kỳ. Ta hay gọi là bảng ánh xạ hay bảng băm. Từ điển được dùng khi ta muốn lưu trữ dữ liệu như mảng nhưng muốn dùng một kiểu dữ liệu nào đó thay cho kiểu dữ liệu số làm chỉ mục. Nó cũng

cho phép ta thêm hoặc bỏ các mục, hơi giống danh sách mảng tuy nhiên nó không phải dịch chuyển các mục phía sau trong bộ nhớ.

Ta có thể dùng kiểu dữ liệu bất kì làm chỉ mục, lúc này ta gọi nó là khoá chứ không phải là chỉ mục nữa. Khi ta cung cấp một khoá truy nhập vào một phần tử, nó sẽ xử lí trên giá trị của khoá và trả về một số nguyên tùy thuộc vào khoá, và được dùng để truy nhập vào 'mảng' để lấy dữ liệu.

Chương 9: Reflection

Mục đích của chương:

- Sử dụng siêu dữ liệu của .Net.
- Vai trò của thuộc tính trong quá trình xây dựng ứng dụng.
- Sử dụng thuộc tính trong quá trình phát triển.
- Sử dụng reflection để lấy tất cả thông tin về lớp.

9. 1 Thuộc tính (attribute) tùy chọn

Viết thuộc tính tùy chọn

Để hiểu cách viết thuộc tính tùy chọn, ta cần xem trình biên dịch làm gì khi nó gặp một mục trong mã được đánh dấu với một attribute. Giả sử ta có thuộc tính khai báo như sau:

```
[TenTruong("SoCMND")]
public string SoCMND
{
    get {
        // vv...
    }
}
```

Như ta thấy thuộc tính SoCMND có một thuộc tính TenTruong, trình biên dịch sẽ nối chuỗi attribute với tên này thành TenTruongAttribute, sau đó tìm trong tất cả các namespace lớp có tên này. Tuy nhiên nếu ta đánh dấu một mục với một thuộc tính mà tên của nó có phần cuối là attribute thì trình biên dịch sẽ không thêm chuỗi attribute lần nữa ví dụ:

```
[TenTruongattribute("SoCMND")]
public string SoCMND
{
    get {
        // vv...
    }
}
```

Nếu trình biên dịch không tìm thấy một lớp thuộc tính tương ứng, hoặc thấy nhưng cách mà ta dùng thuộc tính không phù hợp với thông tin trong lớp thuộc tính, thì trình biên dịch sẽ sinh ra lỗi.

Các lớp thuộc tính tùy chọn

Giả sử ta đã định nghĩa một thuộc tính TenTruong như sau:

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false,
    Inherited=false)]
public class TenTruongAttribute: Attribute
{
    private string ten;
    public TenTruongAttribute(string ten)
    {
        this.ten = ten;
    }
}
```

Điều đầu tiên ta chú ý là lớp attribute được đánh dấu với một thuộc tính “AttributeUsage”. AttributeUsage chỉ định các mục nào trong mã của chúng ta áp dụng thuộc tính tùy chọn. Thông tin này được cho bởi thông số đầu tiên. Thông số này là một kiểu liệt kê attributeTargets. Trong ví dụ trên ta chỉ định thuộc tính TenTruong chỉ được áp dụng đến các thuộc tính.

Định nghĩa của kiểu liệt kê attributeTargets là:

```
public enum attributeTargets
{
    All = 0x00003FFF,
    Assembly = 0x00000001,
    Class = 0x00000004,
    Constructor = 0x00000020,
    Delegate = 0x00001000,
    Enum = 0x00000010,
    Event = 0x00000200,
    Field = 0x00000100,
    Interface = 0x00000400,
    Method = 0x00000040,
    Module = 0x00000002,
    Parameter = 0x00000800,
    Property = 0x00000080,
    ReturnValue = 0x00002000,
    Struct = 0x00000008
}
```

Khi áp dụng thuộc tính đến các phần tử chương trình, ta đặt thuộc tính trong ngoặc vuông ngay trước phần tử. Một thuộc tính có thể được áp dụng đến một Assembly nhưng cần được đánh dấu với từ khóa Assembly:

```
[assembly: SomeAssemblyattribute(Parameters)]
```

Để kết hợp nhiều kiểu khác nhau trên một phần tử nào đó, ta viết như sau:

```
[attributeUsage(attributeTargets.Property | attributeTargets.Field,
    AllowMultiple=false,
    Inherited=false)]
public class TenTruongattribute: attribute
```

Ta cũng có thể dùng attributeTargets.All để áp dụng thuộc tính cho tất cả các trường hợp. Thuộc tính attributeUsage còn chứa hai thông số khác là AllowMultiple and Inherited, chỉ định với cú pháp khác của <tên_thuộc_tính>=<giá_trị_thuộc_tính>. Thông số này là tùy chọn, thông số AllowMultiple chỉ định một attribute có thể áp dụng nhiều hơn một lần đến cùng một mục. Nếu thiết đặt là false thì trình biên dịch sẽ thông báo lỗi nếu nó thấy:

```
[TenTruong ("SOCMND")]
[TenTruong ("SOBaoHiem")]
public string SOCMND
{
    // vv...
```

Nếu thông số Inherited là true, thì một thuộc tính có thể áp dụng đến một lớp hay một giao diện cũng sẽ được áp dụng đến tất cả các lớp hay giao diện được kế thừa. Nếu

thuộc tính được áp dụng cho phương thức hay thuộc tính thì nó tự động áp dụng đến bất kỳ phương thức hay thuộc tính nào được khai báo override.

Đặc tả các thông số thuộc tính

Ta sẽ kiểm tra làm thế nào ta có thể chỉ định thông số cho thuộc tính tùy chọn, Khi trình biên dịch gặp lệnh:

```
[TenTruong("SOCMND")]
public string SOCMND
{
    ...
}
```

Nó kiểm tra thông số truyền vào attribute , trong trường hợp này là chuỗi và tìm phương thức tạo lập của thuộc tính mà nhận các thông số này, nếu thấy thì không có vấn đề gì ngược lại trình biên dịch sẽ sinh ra lỗi.

Các thông số tùy chọn

Ta thấy thuộc tính attributeUsage có một cú pháp cho phép thêm các giá trị vào trong thuộc tính. Cú pháp này có liên quan đến việc chỉ định tên của các thông số được chọn. Giả sử ta cập nhật lại thuộc tính SoCMND như sau:

```
[TenTruong("SoCMND ", ChuThich="Day la truong khoa")]
public string SoCMND
{
    ...
}
```

Trong trường hợp này, trình biên dịch sẽ nhận ra<ten_tham_số>= cú pháp của thông số thứ hai. Nó sẽ tìm một thuộc tính public (hoặc field) của tên đó mà nó có thể dùng để đặt giá trị của thông số này. Nếu ta muốn đoạn mã trên làm việc ta thêm mã sau vào TenTruongattribute:

```
[attributeUsage(attributeTargets.Property,
AllowMultiple=false,
Inherited=false)]
public class TenTruongattribute: attribute
{
    private string chuthich;
    public string ChuThich
    {
        ...
    }
}
```

9.2 Reflection

Reflection là một kỹ thuật cho phép ta tìm ra thông tin về các kiểu dữ liệu trong chương trình. Hầu hết những lớp này nằm trong namespace System.Reflection. Lớp System.Type cho phép ta truy nhập thông tin liên quan đến việc định nghĩa bất kỳ kiểu dữ liệu nào.

Lớp System.Type

Ta dùng lớp Type để lấy tên của một kiểu:

```
Type t = typeof(double);
```

Mặc dù ta cho rằng Type là một lớp nhưng thực sự nó là một lớp cơ sở trừu tượng, bắt cứ khi nào ta khởi tạo một đối tượng Type ta thực sự khởi tạo một lớp dẫn xuất của

Type. Type có một lớp dẫn xuất đáp ứng mỗi kiểu dữ liệu. Có 3 cách lấy một tham chiếu Type cho kiểu dữ liệu bất kì:

- Dùng tác từ typeof, tác từ này lấy tên của kiểu như là thông số.
- Dùng phương thức GetType(), mà tất cả các lớp kế thừa từ System.Object:

```
double d = 10;  
Type t = d.GetType();
```

GetType() hữu ích khi ta có một tham chiếu đối tượng và không chắc đối tượng thực sự là thể hiện của lớp nào.

- Ta cũng có thể gọi phương thức static của lớp type, GetType():

```
Type t = Type.GetType("System.Double");
```

Các thuộc tính của Type

Một số thuộc tính lấy chuỗi chứa các tên khác nhau kết hợp với lớp:

Thuộc tính	Trả về
Name	tên của kiểu dữ liệu
FullName	tên đầy đủ bao gồm cả namespace
Namespace	tên namespace của kiểu dữ liệu.

Có thể lấy tham chiếu đến kiểu đối tượng của các lớp liên quan:

Thuộc tính	Kiểu tham chiếu trả về tương ứng với
BaseType	kiểu cơ sở trực tiếp của kiểu này
UnderlyingSystemType	kiểu mà kiểu này ánh xạ trong thời gian chạy.NET

Một số thuộc tính luận lý kiểm tra kiểu, ví dụ là một lớp hay một kiểu liệt kê... những thuộc tính này bao gồm: **IsAbstract**, **IsArray**, **IsClassebly**, **IsEnum**, **IsInterface**, **IsPointer**, **IsPrimitive**, **IsPublic**, **IsSealed**, and **IsValueType**.

Ví dụ dùng kiểu dữ liệu cơ bản:

```
Type intType = typeof(int);  
  
Console.WriteLine(intType.IsAbstract); // false  
Console.WriteLine(intType.IsClassebly); // false  
Console.WriteLine(intType.IsEnum); // false  
Console.WriteLine(intType.IsPrimitive); // true  
Console.WriteLine(intType.IsValueType); // true  
hoặc dùng lớp Vector:  
Type intType = typeof(Vector); Console.WriteLine(intType.IsAbstract);  
// false  
Console.WriteLine(intType.IsClassebly); // true  
Console.WriteLine(intType.IsEnum); // false  
Console.WriteLine(intType.IsPrimitive); // false  
Console.WriteLine(intType.IsValueType); // false
```

Các phương thức

Hầu hết các phương thức của System.Type được sử dụng để chứa chi tiết các thành viên của kiểu dữ liệu tương ứng - hàm tạo lập, thuộc tính, phương thức, sự kiện... có nhiều phương thức nhưng tất cả chúng đều theo nền chung. Ví dụ, có hai phương thức mà nhận chi tiết phương thức của kiểu dữ liệu: GetMethod() và GetMethods().

GetMethod() trả về một tham chiếu đến đối tượng System.Reflection MethodInfo chứa chi tiết của một phương thức. GetMethods() trả về một mảng tham chiếu.

Ví dụ phương thức GetMethods() không lấy thông số nào và trả về chi tiết của tất cả phương thức thành viên của kiểu dữ liệu:

```
Type t = typeof(double);
MethodInfo [] methods = t.GetMethods();
foreach (MethodInfo nextMethod in methods)
{
    ...
}
```

Kiểu đối tượng trả về	Các phương thức (phương thức số nhiều (có 's' ở cuối tên) trả về một mảng)
ConstructorInfo	Gvconstructor(), Gvconstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()
PropertyInfo	GetProperty(), GetProperties()

Phương thức GetMember() và GetMembers() trả về chi tiết của bất kì hay tất cả thành viên của kiểu dữ liệu không cần biết đó là hàm tạo lập hay thuộc tính phương thức.

Chương trình minh họa lấy tên của tất cả các phương thức của một lớp dùng reflection:

```
using System;
using System.Reflection;
public interface IGiaoDien1
{
    void PhuongThucA();
}
public interface IGiaoDien2
{
    void PhuongThucB();
}
```

```
public class ViDu : IGiaoDien1, IGiaoDien2
{
    public enum KieuLietKe { }

    public int nguyen;
    public string chuoitinh;

    public void PhuongThuc(int p1, string p2)
    {
    }

    public int ThuocTinh
    {
        get { return nguyen; }
        set { nguyen = value; }
    }

    void IGiaoDien1.PhuongThucA() { }
    void IGiaoDien2.PhuongThucB() { }
}

public class MainClass
{
    public static void Main(string[] args)
    {
        ViDu f = new ViDu();
        Type t = f.GetType();
        MethodInfo[] mi = t.GetMethods();
        foreach (MethodInfo m in mi)
            Console.WriteLine("Phuong Thuc: {0}", m.Name);
    }
}
```