TUYÊN BỐ BẢN QUYỀN

Tài liệu này thuộc loại sách giáo trình nên các nguồn thông tin có thể được phép dùng nguyên bản hoặc trích dùng cho các mục đích về đào tạo và tham khảo.
Mọi mục đích khác mang tính lệch lạc hoặc sử dụng với mục đích kinh doanh thiếu lành mạnh sẽ bị nghiêm cấm.

LỜI GIỚI THIỆU

Đây là tài liệu được biên soạn theo chương trình đào tạo Cao đẳng nghề Công nghệ thông tin (ứng dụng phần mềm).

Để học tốt môn học này, người học nên có kiến thức về lập trình căn bản.

Lập trình Windows 2 là một mô đun nhằm giúp người học có kiến thức và kỹ năng lập trình cơ sở trên môi trường Windows. Với phạm vi của tài liệu này, chúng tôi cung cấp cho người học các kiến thức và kỹ năng chính sau:

- Cài đặt và sử dụng được với môi trường C# trên bộ Visual Studio.Net 2010 trở lên;
- Khai báo được lớp đối tượng, các thành phần của lớp đối tượng và sử dụng được lớp đối tượng trên ngôn ngữ C#;
- Cài đặt và xây dựng được chương trình theo phương pháp hướng đối tượng trên một ngôn ngữ lập trình C#;
- Xây dựng các ứng dụng Windows Forms đơn giản kết nối đến cơ sở dữ liệu;
- Nghiêm túc, tỉ mỉ trong quá trình tiếp cận với công cụ mới;
- Chủ động sáng tạo tìm kiếm các ứng dụng viết trên C#.

Trong quá trình biên soạn, chúng tôi có tham khảo nhiều nguồn tài liệu khác nhau và từ nguồn Internet. Mặc dù rất cố gắng biên soạn lại nhưng chắc chắn không tránh khỏi những thiếu sót, tác giả rất mong nhận được những ý kiến đóng góp để tài liệu ngày càng hoàn thiện hơn để cung cấp cho người học những kiến thức và kỹ năng trọng tâm..

Cần Thơ, ngày tháng năm 20

Tham gia biên soạn

1. Chủ biên Nguyễn Phát Minh

MỤC LỤC

TRANG

GIÁO TRÌNH MÔN HỌC/MÔ ĐUN

Tên môn học/mô đun: Lập trình C# căn bản
Mã môn học/mô đun: MĐ 11
Vị trí, tính chất, ý nghĩa và vai trò của môn học/mô đun:

- Vị trí: là mô đun được bố trí giảng dạy sau các môn cơ sở nghề.
- Tính chất: là mô đun bắt buộc thuộc chuyên môn nghề của chương trình đào tạo Cao đẳng (ứng dụng phần mềm).
- Ý nghĩa và vai trò của môn học/mô đun: Lập trình C# căn bản là môn học cơ bản để sinh viên tìm hiểu về lập trình để làm nền tảng học các môn chuyên sâu về lập trình sau này

Mục tiêu của môn học/mô đun:
- Về kiến thức:
  + Hiểu được các kiến thức về nền tảng Microsoft .NET.
  + Biết các kiến thức và kỹ năng về lập trình hướng đối tượng trên C#.
  + Có kiến thức và kỹ năng xử lý mảng, chuỗi;
- Về kỹ năng:
  + Tạo được các ứng dụng dạng console sử dụng ngôn ngữ C# trên môi trường .Net;
- Về năng lực tự chủ và trách nhiệm:
  + Nghiêm túc, tỉ mỉ trong việc tiếp nhận kiến thức. Chủ động, tích cực trong thực hành và tìm kiếm nguồn bài tập liên quan

Nội dung của môn học/mô đun:

Số TT	Tên các bài trong mô đun	Thời gian			
		Tổng số	Lý thuyết	Thực hành, Bài tập	Kiểm tra* (LT hoặc TH)
1.	Cơ bản về C#	8	4	4	
	C# là gì?				
	Chương trình đầu tiên				
	Các kiểu dữ liệu				
	Biến & hằng số				
2	Các lệnh có cấu trúc và vòng lặp	18	9	8	1
	Các lệnh có cấu trúc				
	Lệnh vòng lặp				
3	Mảng, chỉ mục và tập hợp	18	9	8	1
	Mảng				
	Chỉ mục				
	Tập hợp				
4	Xử lý chuỗi	16	8	7	1
	Lớp đối tượng string				

	Các biểu thức quy tắc				
	Tổng cộng	**60**	**30**	**27**	**3**

BÀI 1: CƠ BẢN VỀ C#
Mã bài: MĐ 11 - 02

Giới thiệu:
Trong bài học này sinh viên sẽ được làm quen với ngôn ngữ lập trình phổ biến nhất hiện nay là C#

Mục tiêu:
+ Biết kiến thức và các chức năng tiên tiến trên C#;
+ Hiểu về các kiểu dữ liệu dựng sẵn của C#;
+ Hiểu được các cơ chế thực thi các biến, hằng và các biểu thức trên C#;
+ Hiểu về khoảng trắng;
+ Biết kiến thức về không gian tên (Namespace);
+ Biết kiến thức về các toán tử;
+ Biết kiến thức về chỉ dẫn biên dịch;
+ Tạo và thực thi được ứng dụng đơn giản trên C#;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Nội dung chính:

1. Tại sao phải sử dụng C#

Nhiều người tin rằng không cần thiết có một ngôn ngữ lập trình mới. Java, C++, Perl, Microsoft Visual Basic, và những ngôn ngữ khác được nghĩ rằng đã cung cấp tất cả những chức năng cần thiết.

Ngôn ngữ C# là một ngôn ngữ được dẫn xuất từ C và C++, nhưng nó được tạo từ nền tảng phát triển hơn. Microsoft bắt đầu với công việc trong C và C++ và thêm vào những đặc tính mới để làm cho ngôn ngữ này dễ sử dụng hơn. Nhiều trong số những đặc tính này khá giống với những đặc tính có trong ngôn ngữ Java. Không dừng lại ở đó, Microsoft đưa ra một số mục đích khi xây dựng ngôn ngữ này. Những mục đích này được được tóm tắt như sau:

- C# là ngôn ngữ đơn giản
- C# là ngôn ngữ hiện đại
- C# là ngôn ngữ hướng đối tượng
- C# là ngôn ngữ mạnh mẽ và mềm dẻo
- C# là ngôn ngữ có ít từ khóa
- C# là ngôn ngữ hướng module
- C# sẽ trở nên phổ biến

2. Kiểu dữ liệu

Như chúng ta đã biết C# là một ngôn ngữ hướng đối tượng rất mạnh, và công việc của người lập trình là kế thừa để tạo và khai thác các đối tượng. Do vậy để nắm vững và phát triển tốt người lập trình cần phải đi từ những bước đi dầu tiên tức là đi vào tìm hiểu những phần cơ bản và cốt lõi nhất của ngôn ngữ.

Kiểu dữ liệu

C# là ngôn ngữ lập trình mạnh về kiểu dữ liệu, một ngôn ngữ mạnh về kiểu dữ liệu là phải khai báo kiểu của mỗi đối tượng khi tạo (kiểu số nguyên, số thực, kiểu chuỗi, kiểu điều khiển...) và trình biên dịch sẽ giúp cho người lập trình không bị lỗi khi chỉ cho phép một loại kiểu dữ liệu có thể được gán cho các kiểu dữ liệu khác. Kiểu dữ liệu của một đối tượng là một tín hiệu để trình biên dịch nhận biết kích thước của một đối tượng

(kiểu int có kích thước là 4 byte) và khả năng của nó (như một đối tượng button có thể vẽ, phản ứng khi nhấn,...).

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

C# phân tập hợp kiểu dữ liệu này thành hai loại: Kiểu dữ liệu giá trị (value) và kiểu dữ liệu tham chiếu (reference). Việc phân chi này do sự khác nhau khi lưu kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu trong bộ nhớ. Đối với một kiểu dữ liệu giá trị thì sẽ được lưu giữ kích thước thật trong bộ nhớ đã cấp phát là stack. Trong khi đó thì địa chỉ của kiểu dữ liệu tham chiếu thì được lưu trong stack nhưng đối tượng thật sự thì lưu trong bộ nhớ heap.

Nếu chúng ta có một đối tượng có kích thước rất lớn thì việc lưu giữ chúng trên bộ nhớ heap rất có ích, trong chương 4 sẽ trình bày những lợi ích và bất lợi khi làm việc với kiểu dữ liệu tham chiếu, còn trong chương này chỉ tập trung kiểu dữ kiểu cơ bản hay kiểu xây dựng sẵn.

Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu cấu trúc đều là kiểu dữ liệu tham chiếu.

Ngoài ra C# cũng hỗ trợ một kiểu con trỏ C++, nhưng hiếm khi được sử dụng, và chỉ khi nào làm việc với những đoạn mã lệnh không được quản lý (unmanaged code). Mã lệnh không được quản lý là các lệnh được viết bên ngoài nền .MS.NET, như là các đối tượng COM.

Kiểu dữ liệu xây dựng sẵn

Ngôn ngữ C# đưa ra các kiểu dữ liệu xây dựng sẵn rất hữu dụng, phù hợp với một ngôn ngữ lập trình hiện đại, mỗi kiểu dữ liệu được ánh xạ đến một kiểu dữ liệu được hỗ trợ bởi hệ thống xác nhận ngôn ngữ chung (Common Language Specification: CLS) trong MS.NET. Việc ánh xạ các kiểu dữ liệu nguyên thuỷ của C# đến các kiểu dữ liệu của .NET sẽ đảm bảo các đối tượng được tạo ra trong C# có thể được sử dụng đồng thời với các đối tượng được tạo bởi bất cứ ngôn ngữ khác được biên dịch bởi .NET, như VB.NET.

Mỗi kiểu dữ liệu có một sự xác nhận và kích thước không thay đổi, không giống như C++, int trong C# luôn có kích thước là 4 byte bởi vì nó được ánh xạ từ kiểu Int32 trong . NET.

Bảng sau sẽ mô tả một số các kiểu dữ liệu được xây dựng sẵn

Mô tả các kiểu dữ liệu xây dựng sẵn

Kiểu C#	Số byte	Kiểu .NET	Mô tả
byte	1	Byte	Số nguyên dương không dấu từ 0-255
char	2	Char	Ký tự Unicode
bool	1	Boolean	Giá trị logic true/ false
sbyte	1	Sbyte	Số nguyên có dấu (từ -128 đến 127)
short	2	Int16	Số nguyên có dấu giá trị từ -32768 đến32767.
ushort	2	Uịnt16	Số nguyên không dấu 0 – 65.535

int	4	Int32	Số nguyên có dấu –2.147.483.647 và2.147.483.647
uint	4	Uint32	Số nguyên không dấu 0 – 4.294.967.295
float	4	Single	Kiểu dấu chấm động, giá trị xấp xỉ từ 3,4E-38 đến 3,4E+38, với 7 chữ số có nghĩa..
double	8	Double	Kiểu dấu chấm động có độ chính xác gấp đôi, giá trị xấp xỉ từ 1,7E-308 đến 1,7E+308,với 15,16 chữ số có nghĩa.
decimal	8	Decimal	Có độ chính xác đến 28 con số và giá trị thập phân, được dùng trong tính toán tài chính, kiểu này đòi hỏi phải có hậu tố “m” hay “M”theo sau giá trị.
long	8	Int64	Kiểu số nguyên có dấu có giá trị trong khoảng :-9.223.370.036.854.775.808 đến9.223.372.036.854.775.807
ulong	8	Uint64	Số nguyên không dấu từ 0 đến0xffffffffffffffff

Kiểu giá trị logic chỉ có thể nhận được giá trị là true hay false mà thôi. Một giá trị nguyên không thể gán vào một biến kiểu logic trong C# và không có bất cứ chuyển đổi ngầm định nào. Điều này khác với C/C++, cho phép biến logic được gán giá trị nguyên, khi đó giá trị nguyên 0 là false và các giá trị còn lại là true.

Chọn kiểu dữ liệu

Thông thường để chọn một kiểu dữ liệu nguyên để sử dụng như short, int hay long thường dựa vào độ lớn của giá trị muốn sử dụng. Ví dụ, một biến ushort có thể lưu giữ giá trị từ 0 đến 65.535, trong khi biến ulong có thể lưu giữ giá trị từ 0 đến 4.294.967.295, do đó tùy vào miền giá trị của phạm vi sử dụng biến mà chọn các kiểu dữ liệu thích hợp nhất. Kiểu dữ liệu int thường được sử dụng nhiều nhất trong lập trình vì với kích thước

4 byte của nó cũng đủ để lưu các giá trị nguyên cần thiết.

Kiểu số nguyên có dấu thường được lựa chọn sử dụng nhiều nhất trong kiểu số trừ khi có lý do chính đáng để sử dụng kiểu dữ liệu không dấu.

Stack và Heap
Stack là một cấu trúc dữ liệu lưu trữ thông tin dạng xếp chồng tức là vào sau ra trước (Last In First Out : LIFO), điều này giống như chúng ta có một chồng các đĩa, ta cứ xếp các đĩa vào chồng và khi lấy ra thì đĩa nào nằm trên cùng sẽ được lập ra trước, tức là đĩa vào sau sẽ được lấy ra trước. Trong C#, kiểu giá trị như kiểu số nguyên được cấp phát trên stack, đây là vùng nhớ được thiết lập để lưu các giá trị, và vùng nhớ này được tham chiếu bởi tên của biến. Kiểu tham chiếu như các đối tượng thì được cấp phát trên heap. Khi một đối tượng được cấp phát trên heap thì địa chỉ của nó được trả về, và địa chỉ này được gán đến một tham chiếu. Thỉnh thoảng cơ chế thu gom sẽ hủy đối tượng trong stack sau khi một vùng trong stack được đánh dấu là kết thúc. Thông thường một vùng trong stack được định nghĩa bởi một hàm. Do đó, nếu chúng ta khai báo một biến cục bộ trong một hàm là một đối tượng thì đối tượng này sẽ đánh dấu để hủy khi kết thúc hàm. Những đối tượng trên heap sẽ được thu gom sau khi một tham chiếu cuối cùng đến đối tượng đó được gọi.

Cách tốt nhất khi sử dụng biến không dấu là giá trị của biến luôn luôn dương, biến này thường thể hiện một thuộc tính nào đó có miền giá trị dương. Ví dụ khi cần khai báo một biến lưu giữ tuổi của một người thì ta dùng kiểu byte (số nguyên từ 0-255) vì tuổi của người không thể nào âm được.

Kiểu float, double, và decimal đưa ra nhiều mức độ khác nhau về kích thước cũng như độ chính xác.Với thao tác trên các phân số nhỏ thì kiểu float là thích hợp nhất. Tuy nhiên lưu ý rằng trình biên dịch luôn luôn hiểu bất cứ một số thực nào cũng là một số kiểu double trừ khi chúng ta khai báo rõ ràng. Để gán một số kiểu float thì số phải có ký tự f theo sau.

```
float soFloat = 24f;
```

Kiểu dữ liệu ký tự thể hiện các ký tự Unicode, bao gồm các ký tự đơn giản, ký tự theo mã Unicode và các ký tự thoát khác được bao trong những dấu nháy đơn. Ví dụ, A là một ký tự đơn giản trong khi \u0041 là một ký tự Unicode. Ký tự thoát là những ký tự đặc biệt bao gồm hai ký tự liên tiếp trong đó ký tự dầu tiên là dấu chéo '\'. Ví dụ, \t là dấu tab. Bảng 3.2 trình bày các ký tự đặc biệt.

Các kiểu ký tự đặc biệt

Ký tự	Ý nghĩa
\'	Dấu nháy đơn
\"	Dấu nháy kép

\\	Dấu chéo
\0	Ký tự null
\a	Alert
\b	Backspace
\f	Sang trang form feed
\n	Dòng mới
\r	Đầu dòng
\t	Tab ngang
\v	Tab dọc

Chuyển đổi các kiểu dữ liệu

Những đối tượng của một kiểu dữ liệu này có thể được chuyển sang những đối tượng của một kiểu dữ liệu khác thông qua cơ chế chuyển đổi tường minh hay ngầm định. Chuyển đổi nhầm định được thực hiện một cách tự động, trình biên dịch sẽ thực hiện công việc này. Còn chuyển đổi tường minh diễn ra khi chúng ta gán ép một giá trị cho kiểu dữ liệu khác.

Việc chuyển đổi giá trị ngầm định được thực hiện một cách tự động và đảm bảo là không mất thông tin. Ví dụ, chúng ta có thể gán ngầm định một số kiểu short (2 byte) vào một số kiểu int (4 byte) một cách ngầm định. Sau khi gán hoàn toàn không mất dữ liệu vì bất cứ giá trị nào của short cũng thuộc về int:

short x = 10; int y = x; // chuyển đổi ngầm định

Tuy nhiên, nếu chúng ta thực hiện chuyển đổi ngược lại, chắc chắn chúng ta sẽ bị mất thông tin. Nếu giá trị của số nguyên đó lớn hơn 32.767 thì nó sẽ bị cắt khi chuyển đổi. Trình biên dịch sẽ không thực hiện việc chuyển đổi ngầm định từ số kiểu int sang số kiểu short:

short x; int y = 100; x = y; // Không biên dịch, lỗi !!!

Để không bị lỗi chúng ta phải dùng lệnh gán tường minh, đoạn mã trên được viết lại như sau:

```
short x; int y = 500;
x = (short) y; // Ép kiểu tường minh, trình biên dịch không báo lỗi
```

3. Biến và hằng

Để tạo một biến chúng ta phải khai báo kiểu của biến và gán cho biến một tên duy nhất. Biến có thể được khởi tạo giá trị ngay khi được khai báo, hay nó cũng có thể được gán một giá trị mới vào bất cứ lúc nào trong chương trình. Ví dụ sau minh họa sử dụng biến.

Khởi tạo và gán giá trị đến một biến.

```
---------------------------------------------------------------------------
class MinhHoaC3 { static void Main() { int bien1 = 9;
System.Console.WriteLine("Sau khi khoi tao: bien1 ={0}", bien1); bien1 = 15;
System.Console.WriteLine("Sau khi gan: bien1 ={0}", bien1);
}
```

```
}
-------------------------------------------------------------------------
```
Kết quả:
Sau khi khoi tao: bien1 = 9
Sau khi gan: bien1 = 15

Ngay khi khai báo biến ta đã gán giá trị là 9 cho biến, khi xuất biến này thì biến có giá trị là 9. Thực hiện phép gán biến cho giá trị mới là 15 thì biến sẽ có giá trị là 15 và xuất kết quả là 15.

Gán giá trị xác định cho biến

C# đòi hỏi các biến phải được khởi tạo trước khi được sử dụng. Để kiểm tra luật này chúng ta thay đổi dòng lệnh khởi tạo biến bien1 trong ví dụ 3.1 như sau:
int bien1; và giữ nguyên phần còn lại ta được ví dụ sau:
Sử dụng một biến không khởi tạo.

```
-------------------------------------------------------------------------
class MinhHoaC3 { static void Main() { int bien1;
System.Console.WriteLine("Sau khi khoi tao: bien1 ={0}", bien1); bien1 = 15;
System.Console.WriteLine("Sau khi gan: bien1 ={0}", bien1);
}
}
```

Khi biên dịch đoạn chương trình trên thì trình biên dịch C# sẽ thông báo một lỗi sau:
...error CS0165: Use of unassigned local variable ‘bien1’

Việc sử dụng biến khi chưa được khởi tạo là không hợp lệ trong C#. Ví dụ 2 trên không hợp lệ.

Tuy nhiên không nhất thiết lúc nào chúng ta cũng phải khởi tạo biến. Nhưng để dùng được thì bắt buộc phải gán cho chúng một giá trị trước khi có một lệnh nào tham chiếu đến biến đó. Điều này được gọi là gán giá trị xác định cho biến và C# bắt buộc phải thực hiện điều này.Ví dụ sau minh họa một chương trình đúng.
Biến không được khi tạo nhưng sau đó được gán giá trị.

```
-------------------------------------------------------------------------
class MinhHoaC3 { static void Main() { int bien1; bien1 = 9;
System.Console.WriteLine("Sau khi khoi tao: bien1 ={0}", bien1); bien1 = 15;
System.Console.WriteLine("Sau khi gan: bien1 ={0}", bien1);
}
}
```

Hằng

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

Hằng được phân thành ba loại: giá trị hằng (literal), biểu tượng hằng (symbolic constants), kiểu liệu kê (enumerations). Giá trị hằng: ta có một câu lệnh gán như sau:
x = 100;

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

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

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

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

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

Trong khai báo trên, 32 là một hằng số và DoSoi là một biểu tượng hằng có kiểu nguyên. Ví dụ sau minh họa việc sử dụng những biểu tượng hằng.

Sử dụng biểu tượng hằng.

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

Kết quả:

```
Do dong cua nuoc 0
Do soi cua nuoc 100
---------------------------------------------------------------------------
```

Ví dụ trên tạo ra hai biểu tượng hằng chứa giá trị nguyên: DoSoi và DoDong, theo qui tắc đặt tên hằng thì tên hằng thường được đặt theo cú pháp Pascal, nhưng điều này không đòi hỏi bởi ngôn ngữ nên ta có thể đặt tùy ý.

Việc dùng biểu thức hằng này sẽ làm cho chương trình được viết tăng thêm phần ý nghĩa cùng với sự dễ hiểu. Thật sự chúng ta có thể dùng hằng số là 0 và 100 thay thế cho hai biểu tượng hằng trên, nhưng khi đó chương trình không được dễ hiểu và không được tự nhiên lắm. Trình biên dịch không bao giờ chấp nhận một lệnh gán giá trị mới cho một biểu tượng hằng.

Ví dụ trên có thể được viết lại như sau

```
...
class MinhHoaC3 { static void Main() { const int DoSoi = 100; // Độ C const int DoDong
= 0; // Độ C
System.Console.WriteLine( "Do dong cua nuoc {0}", DoDong );
System.Console.WriteLine( "Do soi cua nuoc {0}", DoSoi );
DoSoi = 200;
}
}
```

Khi đó trình biên dịch sẽ phát sinh một lỗi sau:

```
error CS0131: The left-hand side of an assignment must be a variable, property or
indexer.
```

Kiểu liệt kê

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

Trong ví dụ trên, có hai biểu tượng hằng có quan hệ với nhau:

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

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

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

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

enum NhietDoNuoc {
DoDong = 0, DoNguoi = 20, DoAm = 40, DoNong = 60, DoSoi =
100,
}

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

[thuộc tính] [bổ sung] enum <tên liệt kê> [:kiểu cơ sở]
{danh sách các thành phần liệt kê};

Thành phần thuộc tính và bổ sung là tự chọn sẽ được trình bày trong phần sau của sách.

Trong phần này chúng ta sẽ tập trung vào phần còn lại của khai báo. Một kiểu liệt kê bắt đầu với từ khóa enum, tiếp sau là một định danh cho kiểu liệt kê:

enum NhietDoNuoc

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

enum KichThuoc :uint { Nho = 1, Vua = 2, Lon = 3,
}

Là khai báo một kiểu liệt kê phải kết thúc bằng một danh sách liệt kê, danh sách liệt kê này phải có các hằng được gán, và mỗi thành phần phải phân cách nhau dấu phẩy. Ta viết lại ví dụ minh họa trên như sau.

Sử dụng kiểu liệt kê để đơn giản chương trình.

```
---------------------------------------------------------------------------
class MinhHoaC3 { // Khai báo kiểu liệt kê enum NhietDoNuoc
{
DoDong = 0, DoNguoi = 20, DoAm = 40, DoNong = 60, DoSoi =
100, } static void Main() {
System.Console.WriteLine( "Nhiet do dong: {0}", NhietDoNuoc.DoDong);
System.Console.WriteLine( "Nhiet do nguoi: {0}", NhietDoNuoc.DoNguoi);
System.Console.WriteLine( "Nhiet do am: {0}",
NhietDoNuoc.DoAm); System.Console.WriteLine( "Nhiet do nong: {0}",
NhietDoNuoc.DoNong); System.Console.WriteLine(
"Nhiet do soi: {0}", NhietDoNuoc.DoSoi);
}
}
---------------------------------------------------------------------------
```

Kết quả:

```
Nhiet do dong: 0
Nhiet do nguoi: 20
Nhiet do am: 40
Nhiet do nong: 60
Nhiet do soi: 100
--------------------------------------------------------------------------
```

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

Ta xem thử khai báo sau:

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

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

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

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

Kiểu chuỗi ký tự

Kiểu dữ liệu chuỗi khá thân thiện với người lập trình trong bất cứ ngôn ngữ lập trình nào, kiểu dữ liệu chuỗi lưu giữ một mảng những ký tự.

Để khai báo một chuỗi chúng ta sử dụng từ khoá string tương tự như cách tạo một thể hiện của bất cứ đối tượng nào:

```
string chuoi;
```

Một hằng chuỗi được tạo bằng cách đặt các chuỗi trong dấu nháy đôi:

“Xin chao”

Đây là cách chung để khởi tạo một chuỗi ký tự với giá trị hằng:

```
string chuoi = "Xin chao"
```

Định danh

Định danh là tên mà người lập trình chỉ định cho các kiểu dữ liệu, các phương thức, biến, hằng, hay đối tượng.... Một định danh phải bắt đầu với một ký tự chữ cái hay dấu gạch dưới, các ký tự còn lại phải là ký tự chữ cái, chữ số, dấu gạch dưới.

Theo qui ước đặt tên của Microsoft thì đề nghị sử dụng cú pháp lạc đà (camel notation) bắt đầu bằng ký tự thường để đặt tên cho các biến là cú pháp Pascal (Pascal notation) với ký tự đầu tiên hoa cho cách đặt tên hàm và hầu hết các định danh còn lại. Hầu như Microsoft không còn dùng cú pháp Hungary như iSoNguyen hay dấu gạch dưới

Bien_Nguyen để đặt các định danh.

Các định danh không được trùng với các từ khoá mà C# đưa ra, do đó chúng ta không thể tạo các biến có tên như class hay int được. Ngoài ra, C# cũng phân biệt các ký tự thường và ký tự hoa vì vậy C# xem hai biến bienNguyen và bienguyen là hoàn toàn khác nhau.

4. Biểu thức

Những câu lệnh mà thực hiện việc đánh giá một giá trị gọi là biểu thức. Một phép gán một giá trị cho một biến cũng là một biểu thức:

```
var1 = 24;
```

Trong câu lệnh trên phép đánh giá hay định lượng chính là phép gán có giá trị là 24 cho biến var1. Lưu ý là toán tử gán (‘=’) không phải là toán tử so sánh. Do vậy khi sử dụng toán tử này thì biến bên trái sẽ nhận giá trị của phần bên phải. Các toán tử của ngôn ngữ C# như phép so sánh hay phép gán sẽ được trình bày chi tiết trong mục toán tử của chương này.

Do var1 = 24 là một biểu thức được định giá trị là 24 nên biểu thức này có thể được xem như phần bên phải của một biểu thức gán khác:

var2 = var1 = 24;

Lệnh này sẽ được thực hiện từ bên phải sang khi đó biến var1 sẽ nhận được giá trị là 24 và tiếp sau đó thì var2 cũng được nhận giá trị là 24. Do vậy cả hai biến đều cùng nhận một giá trị là 24. Có thể dùng lệnh trên để khởi tạo nhiều biến có cùng một giá trị như:

a = b = c = d = 24;

5. Khoảng trắng

Trong ngôn ngữ C#, những khoảng trắng, khoảng tab và các dòng được xem như là khoảng trắng (whitespace), giống như tên gọi vì chỉ xuất hiện những khoảng trắng để đại diện cho các ký tự đó. C# sẽ bỏ qua tất cả các khoảng trắng đó, do vậy chúng ta có thể viết như sau:

var1=24; Hay var1 = 24;

và trình biên dịch C# sẽ xem hai câu lệnh trên là hoàn toàn giống nhau.

Tuy nhiên lưu ý là khoảng trắng trong một chuỗi sẽ không được bỏ qua. Nếu chúng ta viết:

System.WriteLine("Xin chao!");

mỗi khoảng trắng ở giữa hai chữ “Xin” và “chao” đều được đối xử bình thường như các ký tự khác trong chuỗi.

Hầu hết việc sử dụng khoảng trắng như một sự tùy ý của người lập trình. Điều cốt yếu là việc sử dụng khoảng trắng sẽ làm cho chương trình dễ nhìn dễ đọc hơn Cũng như khi ta viết một văn bản trong MS Word nếu không trình bày tốt thì sẽ khó đọc và gây mất cảm tình cho người xem. Còn đối với trình biên dịch thì việc dùng hay không dùng khoảng trắng là không khác nhau.

Tuy nhiên, cũng cần lưu ý khi sử dụng khoảng trắng như sau:

int x = 24;

tương tự như:

int x=24;

nhưng không giống như:

intx=24;

Trình biên dịch nhận biết được các khoảng trắng ở hai bên của phép gán là phụ và có thể bỏ qua, nhưng khoảng trắng giữa khai báo kiểu và tên biến thì không phải phụ hay thêm mà bắt buộc phải có tối thiểu một khoảng trắng. Điều này không có gì bất hợp lý, vì khoảng trắng cho phép trình biên dịch nhận biết được từ khoá int và không thể nào nhận được intx.

Tương tự như C/C++, trong C# câu lệnh được kết thúc với dấu chấm phẩy ‘;’. Do vậy có thể một câu lệnh trên nhiều dòng, và một dòng có thể nhiều câu lệnh nhưng nhất thiết là hai câu lệnh phải cách nhau một dấu chấm phẩy.

6. Câu lệnh

Trong C# một chỉ dẫn lập trình đầy đủ được gọi là câu lệnh. Chương trình bao gồm nhiều câu lệnh tuần tự với nhau. Mỗi câu lệnh phải kết thúc với một dấu chấm phẩy, ví dụ như:

int x; // một câu lệnh x = 32; // câu lệnh khác int y =x; // đây cũng là một câu lệnh

Những câu lệnh này sẽ được xử lý theo thứ tự. Đầu tiên trình biên dịch bắt đầu ở vị trí đầu của danh sách các câu lệnh và lần lượt đi từng câu lệnh cho đến lệnh cuối cùng, tuy nhiên chỉ đúng cho trường hợp các câu lệnh tuần tự không phân nhánh.

Có hai loại câu lệnh phân nhánh trong C# là : phân nhánh không có điều kiện (unconditional branching statement) và phân nhánh có điều kiện (conditional branching statement).

Ngoài ra còn có các câu lệnh làm cho một số đoạn chương trình được thực hiện nhiều lần, các câu lệnh này được gọi là câu lệnh lặp hay vòng lặp. Bao gồm các lệnh lặp for, while, do, in, và each sẽ được đề cập tới trong mục tiếp theo.

Sau đây chúng ta sẽ xem xét hai loại lệnh phân nhánh phổ biến nhất trong lập trình C#.

Phân nhánh không có điều kiện

Phân nhánh không có điều kiện có thể tạo ra bằng hai cách: gọi một hàm và dùng từ khoá phân nhánh không điều kiện.

Gọi hàm

Khi trình biên dịch xử lý đến tên của một hàm, thì sẽ ngưng thực hiện hàm hiện thời mà bắt đầu phân nhánh dể tạo một gọi hàm mới. Sau khi hàm vừa tạo thực hiện xong và trả về một giá trị thì trình biên dịch sẽ tiếp tục thực hiện dòng lệnh tiếp sau của hàm ban đầu. ví dụ 3.6 minh họa cho việc phân nhánh khi gọi hàm.

Gọi một hàm.

```
using System; class GoiHam { static void Main() {
Console.WriteLine( "Ham Main chuan bi goi ham Func()..." );
Func(); Console.WriteLine( "Tro lai ham Main()"); } static void Func() {
Console.WriteLine( "---->Toi la ham Func()...");
}
}
```

Kết quả:

```
Ham Main chuan bi goi ham Func()...
---->Toi la ham Func()... Tro lai ham Main()
```

Luồng chương trình thực hiện bắt đầu từ hàm Main xử lý đến dòng lệnh Func(), lệnh Func() thường được gọi là một lời gọi hàm. Tại điểm này luồng chương trình sẽ rẽ nhánh để thực hiện hàm vừa gọi. Sau khi thực hiện xong hàm Func, thì chương trình quay lại hàm Main và thực hiện câu lệnh ngay sau câu lệnh gọi hàm Func.

Từ khoá phân nhánh không điều kiện

Để thực hiện phân nhánh ta gọi một trong các từ khóa sau: **goto, break, continue, return, statementthrow**. Việc trình bày các từ khóa phân nhánh không điều kiện này sẽ được đề cập trong chương tiếp theo. Trong phần này chỉ đề cập chung không đi vào chi tiết.

Phân nhánh có điều kiện

Phân nhánh có điều kiện được tạo bởi các lệnh điều kiện. Các từ khóa của các lệnh này như : **if, else, switch**. Sự phân nhánh chỉ được thực hiện khi biểu thức điều kiện phân nhánh được xác định là đúng.

Câu lệnh if...else

Câu lệnh phân nhánh **if...else** dựa trên một điều kiện. Điều kiện là một biểu thức sẽ được kiểm tra giá trị ngay khi bắt đầu gặp câu lệnh đó. Nếu điều kiện được kiểm tra là đúng, thì câu lệnh hay một khối các câu lệnh bên trong thân của câu lệnh if được thực hiện.

Trong câu điều kiện **if...else** thì else là phần tùy chọn. Các câu lệnh bên trong thân của **else** chỉ được thực hiện khi điều kiện của **if** là sai. Do vậy khi câu lệnh đầy đủ **if...else** được dùng thì chỉ có một trong hai if hoặc else được thực hiện. Ta có cú pháp câu điều kiện **if... else** sau:

if (biểu thức điều kiện) <Khối lệnh thực hiện khi điều kiện đúng>
[else
<Khối lệnh thực hiện khi điều kiện sai>]

Nếu các câu lệnh trong thân của if hay else mà lớn hơn một lệnh thì các lệnh này phải được bao trong một khối lệnh, tức là phải nằm trong dấu khối { }:

if (biểu thức điều kiện) { <Lệnh 1> <Lệnh 2> ... } [else { <lệnh 1> <lệnh 2> .... }]

Như trình bày bên trên do else là phần tùy chọn nên được đặt trong dấu ngoặc vuông
[...]. Minh họa bên dưới cách sử dụng câu lệnh if...else.
Dùng câu lệnh điều kiện if...else.

```
using System; class ExIfElse { static void Main() { int var1 = 10; int var2 = 20; if ( var1 > var2) { Console.WriteLine( "var1: {0} > var2:{1}", var1, var2); } else { Console.WriteLine( "var2: {0} > var1:{1}", var2, var1); } var1 = 30; if ( var1 > var2) { var2 = var1++;
Console.WriteLine( "Gan gia tri var1 cho var2"); Console.WriteLine( "Tang bien var1 len mot "); Console.WritelLine( "Var1 = {0}, var2 = {1}", var1, var2); } else { var1 = var2; Console.WriteLine( "Thiet lap gia tri var1 = var2" ); Console.WriteLine( "var1 = {0}, var2 = {1}", var1, var2 );
}
}
}
```

Kết quả:

```
Gan gia tri var1 cho var2
Tang bien var1 len mot
Var1 = 31, var2 = 30
```

Trong ví dụ trên, câu lệnh **if** đầu tiên sẽ kiểm tra xem giá trị của var1 có lớn hơn giá trị của var2 không. Biểu thức điều kiện này sử dụng toán tử quan hệ lớn hơn (>), các toán tử khác như nhỏ hơn (<), hay bằng (==). Các toán tử này thường xuyên được sử dụng trong lập trình và kết quả trả là giá trị đúng hay sai.

Việc kiểm tra xác định giá trị var1 lớn hơn var2 là sai (vì var1 = 10 trong khi var2 = 20), khi đó các lệnh trong **else** sẽ được thực hiện, và các lệnh này in ra màn hình:

var2: 20 > var1: 10

Tiếp theo đến câu lệnh if thứ hai, sau khi thực hiện lệnh gán giá trị của var1 = 30, lúc này điều kiện if đúng nên các câu lệnh trong khối if sẽ được thực hiện và kết quả là in ra ba dòng sau:

Gan gia tri var1 cho var2
Tang bien var1 len mot
Var1 = 31, var2 = 30

Câu lệnh if lồng nhau

Các lệnh điều kiện if có thể lồng nhau để phục vụ cho việc xử lý các câu điều kiện phức tạp. Việc này cũng thường xuyên gặp khi lập trình. Giả sử chúng ta cần viết một chương trình có yêu cầu xác định tình trạng kết hôn của một công dân dựa vào các thông tin như tuổi, giới tính, và tình trạng hôn nhân, dựa trên một số thông tin như sau:

- Nếu công dân là nam thì độ tuổi có thể kết hôn là 20 với điều kiện là chưa có gia đình.
- Nếu công dân là nữ thì độ tuổi có thể kết hôn là 19 cũng với điều kiện là chưa có gia đình.
- Tất cả các công dân có tuổi nhỏ hơn 19 điều không được kết hôn.

Dựa trên các yêu cầu trên ta có thể dùng các lệnh if lồng nhau để thực hiện. Ví dụ sau sẽ minh họa cho việc thực hiện các yêu cầu trên.

Các lệnh if lồng nhau.

```
using System; class TinhTrangKetHon { static void Main() { int tuoi;
bool coGiaDinh; // 0: chưa có gia đình; 1: đã có gia đình bool gioiTinh; // 0: giới tính nữ; 1: giới tính nam tuoi = 24; coGiaDinh = false; // chưa có gia đình if ( tuoi >= 19) { if ( coGiaDinh == false) { if ( gioiTinh == false) // nu Console.WriteLine(" Nu co the ket hon"); else // nam if (tuoi >19) // phải lớn hơn 19 tuoi mới được kết hôn Console.WriteLine(" Nam co the ket hon"); } else // da co gia dinh Console.WriteLine(" Khong the ket hon nua do da ket hon"); } else // tuoi < 19 Console.WriteLine(" Khong du tuoi ket hon" );
}
}
```

Kết quả:

Nam co the ket hon

Theo trình tự kiểm tra thì câu lệnh if đầu tiên được thực hiện, biểu thức điều kiện đúng do tuổi có giá trị là 24 lớn hơn 19. Khi đó khối lệnh trong if sẽ được thực thi. Ở trong khối này lại xuất hiện một lệnh if khác để kiểm tra tình trạng xem người đó đã có gia đình chưa, kết quả điều kiện if là đúng vì coGiaDinh = false nên biểu thức so sánh coGiaDinh == false sẽ trả về giá trị đúng. Tiếp tục xét xem giới tính của người đó là nam hay nữ, vì chỉ có nam trên 19 tuổi mới được kết hôn. Kết quả kiểm tra là nam nên câu lệnh if thứ ba được thực hiện và xuất ra kết quả : “Nam co the ket hon”.

Câu lệnh switch

Khi có quá nhiều điều kiện để chọn thực hiện thì dùng câu lệnh if sẽ rất rối rắm và dài dòng, Các ngôn ngữ lập trình cấp cao đều cung cấp một dạng câu lệnh switch liệt kê các giá trị và chỉ thực hiện các giá trị thích hợp. C# cũng cung cấp câu lệnh nhảy switch có cú pháp sau:

switch (biểu thức điều kiện) { case <giá trị>: <Các câu lệnh thực hiện> <lệnh nhảy> [default: <Các câu lệnh thực hiện mặc định>] }

Cũng tương tự như câu lệnh if, biểu thức để so sánh được đặt sau từ khóa switch, tuy nhiên giá trị so sánh lại được đặt sau mỗi các từ khóa case. Giá trị sau từ khóa case là các giá trị hằng số nguyên như đã đề cập trong phần trước.

Nếu một câu lệnh **case** được thích hợp tức là giá trị sau **case** bằng với giá trị của biểu thức sau **switch** thì các câu lệnh liên quan đến câu lệnh **case** này sẽ được thực thi. Tuy nhiên phải có một câu lệnh nhảy như **break, goto** để điều khiển nhảy qua các **case** khác.Vì nếu không có các lệnh nhảy này thì khi đó chương trình sẽ thực hiện tất cả các **case** theo sau. Để dễ hiểu hơn ta sẽ xem xét ví dụ dưới đây.

Câu lệnh switch.

```
using System; class MinhHoaSwitch { static void Main() { const int mauDo = 0; const int mauCam = 1; const int mauVang = 2; const int mauLuc = 3; const int mauLam = 4; const int mauCham = 5; const int mauTim = 6; int chonMau = mauLuc; switch ( chonMau ) { case mauDo: Console.WriteLine("Ban cho mau do" ); break; case mauCam: Console.WriteLine( "Ban cho mau cam" ); break; case mauVang: //Console.WriteLine( "Ban chon mau vang"); case mauLuc: Console.WriteLine( "Ban chon mau luc"); break; case mauLam: Console.WriteLine( "Ban chon mau lam"); goto case mauCham; case mauCham: Console.WriteLine( "Ban cho mau cham"); goto case mauTim; case mauTim: Console.WriteLine( "Ban chon mau tim"); goto case mauLuc; default: Console.WriteLine( "Ban khong chon mau nao het"); break; } Console.WriteLine( "Xin cam on!"); } }
```

Trong ví dụ trên liệt kê bảy loại màu và dùng câu lệnh switch để kiểm tra các trường hợp chọn màu.Ở đây chúng ta thử phân tích từg câu lệnh case mà không quan tâm đến giá trị biến chonMau.

Mô tả các trường hợp thực hiện câu lệnh switch.

Giá trị chonMau	Câu lệnh case thực hiện	Kết quả thực hiện
mauDo	case mauDo	Ban chon mau do
mauCam	case mauCam	Ban chon mau cam
mauVang	case mauVangcase mauLuc	Ban chon mau luc
mauLuc	case mauLuc	Ban chon mau luc
mauLam	case mauLam case mauCham case mauTimcase mauLuc	Ban chon mau lam Ban chon mau cham Ban chon mau timBan chon mau luc
mauCham	case mauCham case mauTimcase mauLuc	Ban chon mau chamBan chon mau timBan chon mau luc
mauTim	case mauTimcase mauLuc	Ban chon mau timBan chon mau luc

Trong đoạn ví dụ do giá trị của biến chonMau = mauLuc nên khi vào lệnh switch thì case mauLuc sẽ được thực hiện và kết quả như sau:

Kết quả ví dụ 3.9 Ban chon mau luc
Xin cam on!

Đối với người lập trình C/C++, trong C# chúng ta không thể nhảy xuống một trường hợp case tiếp theo nếu câu lệnh case hiện tại không rỗng. Vì vậy chúng ta phải viết như sau:
case 1:// nhảy xuống case 2:

Như minh họa trên thì trường hợp xử lý case 1 là rỗng, tuy nhiên chúng ta không thể viết như sau:
case 1: DoAnything();
// Trường hợp này không thể nhảy xuống case 2 case 2:
trong đoạn chương trình thứ hai trường hợp case 1 có một câu lệnh nên không thể nhảy xuống được. Nếu muốn trường hợp case1 nhảy qua case 2 thì ta phải sử dụng câu lệnh goto một các tường minh:
case 1: DoAnything(); goto case 2; case 2:

Do vậy khi thực hiện xong các câu lệnh của một trường hợp nếu muốn thực hiện một trường hợp case khác thì ta dùng câu lệnh nhảy goto với nhãn của trường hợp đó:
goto case <giá trị>

Khi gặp lệnh thoát break thì chương trình thoát khỏi switch và thực hiện lệnh tiếp sau khối switch đó.

Nếu không có trường hợp nào thích hợp và trong câu lệnh switch có dùng câu lệnh defalut thì các câu lệnh của trường hợp default sẽ được thực hiện. Ta có thể dùng default để cảnh báo một lỗi hay xử lý một trường hợp ngoài tất cả các trường hợp case trong switch.

Trong ví dụ minh họa câu lệnh switch trước thì giá trị để kiểm tra các trường hợp thích hợp là các hằng số nguyên. Tuy nhiên C# còn có khả năng cho phép chúng ta dùng câu lệnh switch với giá trị là một chuỗi, có thể viết như sau:
switch (chuoi1) { case "mau do": .... break; case "mau cam": ... Break; ... }

Câu lệnh lặp

C# cung cấp một bộ mở rộng các câu lệnh lặp, bao gồm các câu lệnh lặp **for, while** và **do... while**. Ngoài ra ngôn ngữ C# còn bổ sung thêm một câu lệnh lặp foreach, lệnh này mới đối với người lập trình C/C++ nhưng khá thân thiện với người lập trình VB. Cuối cùng là các câu lệnh nhảy như **goto, break, continue, và return**.

Câu lệnh nhảy goto

Lệnh nhảy goto là một lệnh nhảy đơn giản, cho phép chương trình nhảy vô điều kiện tới một vị trí trong chương trình thông qua tên nhãn. Tuy nhiên việc sử dụng lệnh goto thường làm mất đi tính cấu trúc thuật toán, việc lạm dụng sẽ dẫn đến một chương trình nguồn mà giới lập trình gọi là "mì ăn liền" rối như mớ bòng bong vậy. Hầu hết các người lập trình có kinh nghiệm đều tránh dùng lệnh **goto**. Sau đây là cách sử dụng lệnh nhảy goto: Tạo một nhãn **goto** đến nhãn

Nhãn là một định danh theo sau bởi dấu hai chấm (:). Thường thường một lệnh goto gắn với một điều kiện nào đó, ví dụ 3.10 sau sẽ minh họa các sử dụng lệnh nhảy **goto** trong chương trình.
Sử dụng goto.
---

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

```
// nhãy về nhãn lap return 0; } }
--------------------------------------------------------------------------------
Kết quả:
i:0 i:1 i:2 i:3 i:4 i:5 i:6 i:7 i:8 i:9
--------------------------------------------------------------------------------
```

Nếu chúng ta vẽ lưu đồ của một chương trình có sử dụng nhiều lệnh goto, thì ta sẽ thấy kết quả rất nhiều đường chồng chéo lên nhau, giống như là các sợi mì vậy. Chính vì vậy nên những đoạn mã chương trình có dùng lệnh goto còn được gọi là "spaghetti code".

Việc tránh dùng lệnh nhảy **goto** trong chương trình hoàn toàn thực hiện được, có thể dùng vòng lặp while để thay thế hoàn toàn các câu lệnh **goto**.

Vòng lặp while

Ý nghĩa của vòng lặp **while** là: "Trong khi điều kiện đúng thì thực hiện các công việc này". Cú pháp sử dụng vòng lặp **while** như sau:

while (Biểu thức) <Câu lệnh thực hiện>

Biểu thức của vòng lặp while là điều kiện để các lệnh được thực hiện, biểu thức này bắt buộc phải trả về một giá trị kiểu bool là **true/false**. Nếu có nhiều câu lệnh cần được thực hiện trong vòng lặp while thì phải đặt các lệnh này trong khối lệnh. Ví dụ sau minh họa việc sử dụng vòng lặp **while**.

Sử dụng vòng lặp while.

```
--------------------------------------------------------------------------------
using System; public class UsingWhile { public static int Main() { int i = 0; while ( i <
10 ) { Console.WriteLine(" i: {0} ",i); i++; } return 0; } }
--------------------------------------------------------------------------------
Kết quả:
i:0 i:1 i:2 i:3 i:4 i:5 i:6 i:7 i:8 i:9
--------------------------------------------------------------------------------
```

Đoạn chương trình trên cũng cho kết quả tương tự như chương trình minh họa 3.10 dùng lệnh goto. Tuy nhiên chương trình 3.11 rõ ràng hơn và có ý nghĩa tự nhiên hơn. Có thể diễn giải ngôn ngữ tự nhiên đoạn vòng lặp while như sau: "Trong khi i nhỏ hơn 10, thì in ra giá trị của i và tăng i lên một đơn vị".

Vòng lặp while sẽ kiểm tra điều kiện trước khi thực hiện các lệnh bên trong, điều này đảm bảo nếu ngay từ đầu điều kiện sai thì vòng lặp sẽ không bao giờ thực hiện. do vậy nếu khởi tạo biến i có giá trị là 11, thì vòng lặp sẽ không được thực hiện.

Vòng lặp do...while

Đôi khi vòng lặp while không thoả mãn yêu cầu trong tình huống sau, chúng ta muốn chuyển ngữ nghĩa của while là "chạy trong khi điều kiện đúng" thành ngữ nghĩa khác như "làm điều này trong khi điều kiện vẫn còn đúng". Nói cách khác thực hiện một hành động, và sau khi hành động được hoàn thành thì kiểm tra điều kiện. Cú pháp sử dụng vòng lặp do...while như sau:

do <Câu lệnh thực hiện> while (điều kiện)

Ở đây có sự khác biệt quan trọng giữa vòng lặp while và vòng lặp do...while là khi dùng vòng lặp do...while thì tối thiểu sẽ có một lần các câu lệnh trong do...while được thực hiện. Điều này cũng dễ hiểu vì lần đầu tiên đi vào vòng lặp do...while thì điều kiện chưa được kiểm tra.

Vòng lặp for

Vòng lặp for bao gồm ba phần chính:

- Khởi tạo biến đếm vòng lặp
- Kiểm tra điều kiện biến đếm, nếu đúng thì sẽ thực hiện các lệnh bên trong vòng for
- Thay đổi bước lặp.

Cú pháp sử dụng vòng lặp for như sau:

for ([phần khởi tạo] ; [biểu thức điều kiện]; [bước lặp]) <Câu lệnh thực hiện>

Vòng lặp for được minh họa trong ví dụ sau:

Sử dụng vòng lặp for.

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

Kết quả:

```
0
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29
```

Trong đoạn chương trình trên có sử dụng toán tử chia lấy dư modulo, toán tử này sẽ được đề cập đến phần sau. Ý nghĩa lệnh i%10 == 0 là kiểm tra xem i có phải là bội số của 10 không, nếu i là bội số của 10 thì sử dụng lệnh WriteLine để xuất giá trị i và sau đó đưa cursor về đầu dòng sau. Còn ngược lại chỉ cần xuất giá trị của i và không xuống dòng.

Đầu tiên biến i được khởi tạo giá trị ban đầu là 0, sau đó chương trình sẽ kiểm tra điều kiện, do 0 nhỏ hơn 30 nên điều kiện đúng, khi đó các câu lệnh bên trong vòng lặp for sẽ được thực hiện. Sau khi thực hiện xong thì biến i sẽ được tăng thêm một đơn vị (i++).

Có một điều lưu ý là biến i do khai bao bên trong vòng lặp for nên chỉ có phạm vi hoạt động bên trong vòng lặp. Ví dụ 3.14 sau sẽ không được biên dịch vì xuất hiện một lỗi.

Phạm vi của biến khai báo trong vòng lặp.

```
using System; public class UsingFor { public static int Main() { for (int i = 0; i < 30; i++) { if (i %10 ==0)
{ Console.WriteLine("{0} ",i); } else { Console.Write("{0} ",i); } } // Lệnh sau sai do biến i chỉ được khai báo bên trong vòng lặp Console.WriteLine(" Ket qua cuoi cung cua i:{0}",i); return 0; } }
```

Câu lệnh lặp foreach

Vòng lặp foreach cho phép tạo vòng lặp thông qua một tập hợp hay một mảng. Đây là một câu lệnh lặp mới không có trong ngôn ngữ C/C++. Câu lệnh foreach có cú pháp chung như sau:

foreach (<kiểu tập hợp> <tên truy cập thành phần > in <
tên tập hợp>) <Các câu lệnh thực hiện>

Do lặp dựa trên một mảng hay tập hợp nên toàn bộ vòng lặp sẽ duyệt qua tất cả các thành phần của tập hợp theo thứ tự được sắp. Khi duyệt đến phần tử cuối cùng trong tập hợp thì chương trình sẽ thoát ra khỏi vòng lặp foreach.
Minh họa việc sử dụng vòng lặp foreach.

---

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

Kết quả:
1 2 3 4 5 6 7 8 9 10

Câu lệnh nhảy break và continue

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

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

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

Hai lệnh break và continue tạo ra nhiều điểm thoát và làm cho chương trình khó hiểu cũng như là khó duy trì. Do vậy phải cẩn trọng khi sử dụng các lệnh nhảy này.

Ví dụ sẽ được trình bày bên dưới minh họa cách sử dụng lệnh continue và break. Đoạn chương trình mô phỏng hệ thống xử lý tín hiệu giao thông đơn giản. Tín hiệu mô phỏng là các ký tự chữ hoa hay số được nhập vào từ bàn phím, sử dụng hàm ReadLine của lớp
Console để đọc một chuỗi ký tự từ bàn phím.

Thuật toán của chương trình khá đơn giản: Khi nhận tín hiệu ‘0’ có nghĩa là mọi việc bình thường, không cần phải làm bất cứ công việc gì cả, kể cả việc ghi lại các sự kiện. Trong chương trình này đơn giản nên các tín hiệu được nhập từ bàn phím, còn trong ứng dụng thật thì tín hiệu này sẽ được phát sinh theo các mẫu tin thời gian trong cơ sở dữ liệu. Khi nhận được tín hiệu thoát (mô phỏng bởi ký tự ‘T’) thì ghi lại tình trạng và kết thúc xử lý. Cuối cùng, bất cứ tín hiệu nào khác sẽ phát ra một thông báo, có thể là thông báo đến nhân viên cảnh sát chẳng hạn...Trường hợp tín hiệu là ‘X’ thì cũng sẽ phát ra một thông báo nhưng sau vòng lặp xử lý cũng kết thúc.
Sử dụng break và continue.

---

using System; public class TrafficSignal { public static int Main() { string signal = "0"; // Khởi tạo tín hiệu // bắt đầu chu trình xử lý tín hiệu while (signal != "X") { //nhập tín hiệu Console.Write("Nhap vao mot tin hieu: "); signal = Console.ReadLine(); // xuất tín hiệu hiện thời Console.WriteLine("Tin hieu nhan duoc: {0}", signal); // phần xử lý tín hiệu if (signal == "T") { // Tín hiệu thoát được gởi // lưu lại sự kiện và thoát Console.WriteLine("Ngung xu ly! Thoat\n"); break; } if (signal == "0") { // Tín hiệu nhận được bình thường // Lưu lại sự kiện và tiếp tục Console.WriteLine("Tat ca dieu tot!\n"); continue; } // Thực hiện một số hành động nào đó // và tiếp tục Console.WriteLine("---bip bip bip\n"); } return 0; } }

Kết quả: sau khi nhập tuần tự các tín hiệu : “0”, “B”, “T”
Nhap vao mot tin hieu: 0
Tin hieu nhan duoc: 0
Tat ca dieu tot!
Nhap vao mot tin hieu: B Tin hieu nhan duoc: B
---bip bip bip
Nhap vao mot tin hieu: T Tin hieu nhan duoc: T Ngung xu ly! Thoat
---

Điểm chính yếu của đoạn chương trình trên là khi nhập vào tín hiệu “T” thì sau khi thực hiện một số hành động cần thiết chương trình sẽ thoát ra khỏi vòng lặp và không xuất ra câu thông báo bip bip bip. Ngược lại khi nhận được tín hiệu 0 thì sau khi xuất thông báo chương trình sẽ quay về đầu vòng lặp để thực hiện tiếp tục và cũng không xuất ra câu thông báo bip bip bip.

7. Toán tử

Toán tử gán

Đến lúc này toán tử gán khá quen thuộc với chúng ta, hầu hết các chương trình minh họa từ đầu sách đều đã sử dụng phép gán. Toán tử gán hay phép gán làm cho toán hạng bên trái thay đổi giá trị bằng với giá trị của toán hạng bên phải. Toán tử gán là toán tử hai ngôi.

Đây là toán tử đơn giản nhất thông dụng nhất và cũng dễ sử dụng nhất.

Toán tử toán học

Ngôn ngữ C# cung cấp năm toán tử toán học, bao gồm bốn toán tử đầu các phép toán cơ bản. Toán tử cuối cùng là toán tử chia nguyên lấy phần dư. Chúng ta sẽ tìm hiểu chi tiết các phép toán này trong phần tiếp sau.

Các phép toán số học cơ bản (+,-,*,/)

Các phép toán này không thể thiếu trong bất cứ ngôn ngữ lập trình nào, C# cũng không ngoại lệ, các phép toán số học đơn giản nhưng rất cần thiết bao gồm: phép cộng (+), phép trừ (-), phép nhân (*), phép chia (/) nguyên và không nguyên.

Khi chia hai số nguyên, thì C# sẽ bỏ phần phân số, hay bỏ phần dư, tức là nếu ta chia 8/ 3 thì sẽ được kết quả là 2 và sẽ bỏ phần dư là 2, do vậy để lấy được phần dư này thì C# cung cấp thêm toán tử lấy dư sẽ được trình bày trong phần kế tiếp.

Tuy nhiên, khi chia cho số thực có kiểu như float, double, hay decimal thì kết quả chia được trả về là một số thực.

Phép toán chia lấy dư

Để tìm phần dư của phép chia nguyên, chúng ta sử dụng toán tử chia lấy dư (%). VíThật sự phép toán chia lấy dư rất hữu dụng cho người lập trình . Khi chúng ta thực hiện một phép chia dư n cho một số khác, nếu số này là bội số của n thì kết quả của phép chia dư là 0.

20 % 5 = 0 vì 20 là một bội số của 5.

Điều này cho phép chúng ta ứng dụng trong vòng lặp, khi muốn thực hiện một công việc nào đó cách khoảng n lần, ta chỉ cần kiểm tra phép chia dư n, nếu kết quả bằng 0 thì thực hiện công việc. Cách sử dụng này đã áp dụng trong ví dụ minh họa sử dụng vòng lặp for bên trên. Ví dụ sau minh họa sử dụng các phép toán chia trên các số nguyên, thực...

Phép chia và phép chia lấy dư.

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

Kết quả: Integer: 4 float: 4.25 double: 4.25 decimal: 4.25
Modulus: 1

Toán tử tăng và giảm

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

Tính toán và gán trở lại

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

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

Luong = Luong * 2; Luong = Luong – 100.000;
...

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

Mô tả các phép toán tự gán

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

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

Luong += 200.000; Luong *= 2;

Luong -= 100.000;

Kết quả của lệnh thứ nhất là giá trị của Luong sẽ tăng thêm 200.000, lệnh thứ hai sẽ làm cho giá trị Luong nhân đôi tức là tăng gấp 2 lần, và lệnh cuối cùng sẽ trừ bớt 100.000 của Luong.

Do việc tăng hay giảm 1 rất phổ biến trong lập trình nên C# cung cấp hai toán tử đặc biệt là tăng một (++) hay giảm một (--).

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

bienDem++;

Toán tử tăng giảm tiền tố và tăng giảm hậu tố

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

var1 = var2++;

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

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

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

var1 = ++var2;

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

Để hiểu rõ hơn về hai phép toán này chúng ta sẽ xem ví dụ minh họa sau

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

---

using System; class Tester { static int Main() { int valueOne = 10; int valueTwo; valueTwo = valueOne++; Console.WriteLine("Thuc hien tang sau: {0}, {1}", valueOne, valueTwo); valueOne = 20; valueTwo = ++valueOne; Console.WriteLine("Thuc hien tang truoc: {0}, {1}", valueOne, valueTwo); return 0; } }

Kết quả:

Thuc hien tang sau: 11, 10

Thuc hien tang truoc: 21, 21

Toán tử quan hệ

Những toán tử quan hệ được dùng để so sánh giữa hai giá trị, và sau đó trả về kết quả là một giá trị logic kiểu bool (true hay false). Ví dụ toán tử so sánh lớn hơn (>) trả về giá trị là true nếu giá trị bên trái của toán tử lớn hơn giá trị bên phải của toán tử. Do vậy 5 >

2 trả về một giá trị là true, trong khi 2 > 5 trả về giá trị false.

Các toán tử quan hệ trong ngôn ngữ C# được trình bày ở bảng 3.4 bên dưới. Các toán tử trong bảng được minh họa với hai biến là value1 và value2, trong đó value1 có giá trị là

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

Tên toán tử	Kí hiệu	Biểu thức so sánh	Kết quả so sánh
So sánh bằng	==	value1 == 100value1 == 50	truefalse
Không bằng	!=	value2 != 100value2 != 90	falsetrue
Lớn hơn	>	value1 > value2value2 > value1	truefalse
Lớn hơn hay bằng	>=	value2 >= 50	true
Nhỏ hơn	<	value1 < value2value2 < value1	falsetrue
Nhỏ hơn hay bằng	<=	value1 <= value2	false

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

Toán tử logic

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

Bảng dưới liệt kệ ba phép toán logic, bảng này cũng sử dụng hai biến minh họa là x, và y trong đó x có giá trị là 5 và y có giá trị là 7.
Các toán tử logic (giả sử x = 5, y = 7)

Tên toán tử	Ký hiệu	Biểu thức logic	Giá trị	Logic
and	&&	(x == 3) && (y ==7)	false	Cả hai điều kiệnphải đúng
or	||	(x == 3) || (y == 7)	true	Chỉ cần một điềukiện đúng
not	!	! (x == 3)	true	Biểu thức trongngoặc phải sai.

Toán tử and sẽ kiểm tra cả hai điều kiện. Trong bảng trên có minh họa biểu thức logic sử dụng toán tử and:
(x == 3) && (y == 7)

Toàn bộ biểu thức được xác định là sai vì có điều kiện (x == 3) là sai.

Với toán tử or, thì một hay cả hai điều kiện đúng thì đúng, biểu thức sẽ có giá trị là sai khi cả hai điều kiện sai. Do vậy ta xem biểu thức minh họa toán tử or:
(x == 3) || (y == 7)

Biểu thức này được xác định giá trị là đúng do có một điều kiện đúng là (y == 7) là đúng.

Đối với toán tử not, biểu thức sẽ có giá trị đúng khi điều kiện trong ngoặc là sai, và ngược lại, do đó biểu thức:
!(x == 3) có giá trị là đúng vì điều kiện trong ngoặc tức là (x == 3) là sai.

Như chúng ta đã biết đối với phép toán logic and thì chỉ cần một điều kiện trong biểu thức sai là toàn bộ biểu thức là sai, do vậy thật là dư thừa khi kiểm tra các điều kiện còn lại một khi có một điều kiện đã sai. Giả sử ta có đoạn chương trình sau:

int x = 8; if ((x == 5) && (y == 10))

Khi đó biểu thức if sẽ đúng khi cả hai biểu thức con là (x == 5) và (y == 10) đúng. Tuy nhiên khi xét biểu thức thứ nhất do giá trị x là 8 nên biểu thức (x == 5) là sai. Khi đó không cần thiết để xác định giá trị của biểu thức còn lại, tức là với bất kỳ giá trị nào của biểu thức (y== 10) thì toàn bộ biểu thức điều kiện if vẫn sai.

Tương tự với biểu thức logic or, khi xác định được một biểu thức con đúng thì không cần phải xác định các biểu thức con còn lại, vì toán tử logic or chỉ cần một điều kiện đúng là đủ:

int x =8; if ((x == 8) || (y == 10))

Khi kiểm tra biểu thức (x == 8) có giá trị là đúng, thì không cần phải xác định giá trị của biểu thức (y == 10) nữa.

Ngôn ngữ lập trình C# sử dụng logic như chúng ta đã thảo luận bên trên để loại bỏ các tính toán so sánh dư thừa và cũng không logic nữa!

Độ ưu tiên toán tử

Trình biên dịch phải xác định thứ tự thực hiện các toán tử trong trường hợp một biểu thức có nhiều phép toán, giả sử, có biểu thức sau:

var1 = 5+7*3;

Biểu thức trên có ba phép toán để thực hiện bao gồm (=, +,*). Ta thử xét các phép toán theo thứ tự từ trái sang phải, đầu tiên là gán giá trị 5 cho biến var1, sau đó cộng 7 vào 5 là 12 cuối cùng là nhân với 3, kết quả trả về là 36, điều này thật sự có vấn đề, không đúng với mục đích yêu cầu của chúng ta. Do vậy việc xây dựng một trình tự xử lý các toán tử là hết sức cần thiết. Các luật về độ ưu tiên xử lý sẽ bảo trình biên dịch biết được toán tử nào được thực hiện trước trong biểu thức.Tương tự như trong phép toán đại số thì phép nhân có độ ưu tiên thực hiện trước phép toán cộng, do vậy 5+7*3 cho kết quả là 26 đúng hơn kết quả 36. Và cả hai phép toán cộng và phép toán nhân điều có độ ưu tiên cao hơn phép gán. Như vậy trình biên dịch sẽ thực hiện các phép toán rồi sau đó thực hiện phép gán ở bước cuối cùng. Kết quả đúng của câu lệnh trên là biến var1 sẽ nhận giá trị là 26.

Trong ngôn ngữ C#, dấu ngoặc được sử dụng để thay đổi thứ tự xử lý, điều này cũng giống trong tính toán đại số. Khi đó muốn kết quả 36 cho biến var1 có thể viết:

var1 = (5+7) * 3;

Biểu thức trong ngoặc sẽ được xử lý trước và sau khi có kết quả là 12 thì phép nhân được thực hiện.

Bảng dưới liệt kê thứ tự độ ưu tiên các phép toán trong C#.

Thứ tự ưu tiên các toán tử

STT	**Loại toán tử**	**Toán tử**	**Thứ tự**
1	Phép toán cơ bản	(x) x.y f(x) a[x] x++ x—new typeofsizeof checked unchecked	Trái
2		+ - ! ~ ++x –x (T)x	Trái
3	Phép nhân	* / %	Trái
4	Phép cộng	+ -	Trái
5	Dịch bit	<< >>	Trái

6	Quan hệ	< > <= >= is	Trái
7	So sánh bằng	== !=	Phải
8	Phép toán logicAND	&	Trái
9	Phép toán logicXOR	^	Trái
10	Phép toán logic OR	|	Trái
11	Điều kiện AND	&&	Trái
12	Điều kiện OR	||	Trái
13	Điều kiện	?:	Phải
14	Phép gán	= *= /= %= += -= <<= >>= &=^= |=	Phải

Các phép toán được liệt kê cùng loại sẽ có thứ tự theo mục thứ thự của bảng: thứ tự trái tức là độ ưu tiên của các phép toán từ bên trái sang, thứ tự phải thì các phép toán có độ ưu tiên từ bên phải qua trái. Các toán tử khác loại thì có độ ưu tiên từ trên xuống dưới, do vậy các toán tử loại cơ bản sẽ có độ ưu tiên cao nhất và phép toán gán sẽ có độ ưu tiên thấp nhất trong các toán tử.

Toán tử ba ngôi

Hầu hết các toán tử đòi hỏi có một toán hạng như toán tử (++, --) hay hai toán hạng như (+,-,*,/,...). Tuy nhiên, C# còn cung cấp thêm một toán tử có ba toán hạng (?:). Toán tử này có cú pháp sử dụng như sau:
<Biểu thức điều kiện > ? <Biểu thức thứ 1> : <Biểu thức thứ 2>

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

---

using System; class Tester { public static int Main() { int value1; int value2; int maxValue; value1 = 10; value2
= 20; maxValue = value1 > value2 ? value1 : value2;
Console.WriteLine("Gia tri thu nhat {0}, gia tri thu hai
{1}, gia tri lon nhat {2}",value1,value2, maxValue); return 0; } }

Kết quả:
Gia tri thu nhat 10, gia tri thu hai 20, gia tri lon nhat 20

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

gán giá trị value1 cho biến maxValue, còn ngược lại thì gán giá trị value2 cho biến maxValue.

8. Namspace

Việc sử dụng namespace trong khi lập trình là một thói quen tốt, bởi vì công việc này chính là cách lưu các mã nguồn để sử dụng về sau. Ngoài thư viện namespace do MS.NET và các hãng thứ ba cung cấp, ta có thể tạo riêng cho mình các namespace. C# đưa ra từ khóa using đề khai báo sử dụng namespace trong chương trình:

using < Tên namespace >

Để tạo một namespace dùng cú pháp sau:

namespace <Tên namespace> { < Định nghĩa lớp A> < Định nghĩa lớp B > ..... }

Đoạn ví dụ sau minh họa việc tạo một namespace.

Tạo một namespace.

```
---------------------------------------------------------------------------
namespace MyLib { 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 trên tạo ra một namespace có tên là MyLib, bên trong namespace này chứa một lớp có tên là Tester. C# cho phép trong một namespace có thể tạo một namespace khác lồng bên trong và không giới hạn mức độ phân cấp này, việc phân cấp này được minh họa trong ví dụ dưới đây.

Tạo các namespace lồng nhau.

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

Lớp Tester trong ví dụ được đặt trong namespace Demo do đó có thể tạo một lớp Tester khác bên ngoài namespace Demo hay bên ngoài namespace MyLib mà không có bất cứ sự tranh cấp hay xung đột nào. Để truy cập lớp Tester dùng cú pháp sau:

MyLib.Demo.Tester

Trong một namespace một lớp có thể gọi một lớp khác thuộc các cấp namespace khác nhau, ví dụ tiếp sau minh họa việc gọi một hàm thuộc một lớp trong namespace khác.

Gọi một namespace thành viên.

```
---------------------------------------------------------------------------
using System; namespace MyLib { namespace Demo1 { class Example1
{
public static void Show1() { Console.WriteLine("Lop Example1");
}
```

```
} } namespace Demo2 { public class Tester { public static int Main() { Demo1.Example1.Show1(); Demo1.Example2.Show2(); return 0;
}
}
}
}
// Lớp Example2 có cùng namespace MyLib.Demo1 với //lớp Example1 nhưng hai khai báo không cùng một khối. namespace MyLib.Demo1 { class Example2
{ public static void Show2() { Console.WriteLine("Lop Example2");
}
}
}
```

Kết quả:

```
Lop Exemple1
Lop Exemple2
```

Ví dụ trên có hai điểm cần lưu ý là cách gọi một namespace thành viên và cách khai báo các namspace. Như chúng ta thấy trong namespace MyLib có hai namespace con cùng cấp là Demo1 và Demo2, hàm Main của Demo2 sẽ được chương trình thực hiện, và trong hàm Main này có gọi hai hàm thành viên tĩnh của hai lớp Example1 và Example2 của namespace Demo1.

Ví dụ trên cũng đưa ra cách khai báo khác các lớp trong namespace. Hai lớp Example1 và Example2 điều cùng thuộc một namespace MyLib.Demo1, tuy nhiên Example2 được khai báo một khối riêng lẻ bằng cách sử dụng khai báo:

```
namespace MyLib.Demo1 { class Example2 { .... } }
```

Việc khai báo riêng lẻ này có thể cho phép trên nhiều tập tin nguồn khác nhau, miễn sao đảm bảo khai báo đúng tên namspace thì chúng vẫn thuộc về cùng một namespace.

9. Cách chỉ dẫn và biên dịch

Đối với các ví dụ minh họa trong các phần trước, khi biên dịch thì toàn bộ chương trình sẽ được biên dịch. Tuy nhiên, có yêu cầu thực tế là chúng ta chỉ muốn một phần trong chương trình được biên dịch độc lập, ví dụ như khi debug chương trình hoặc xây dựng các ứng dụng...

Trước khi một mã nguồn được biên dịch, một chương trình khác được gọi là chương trình tiền xử lý sẽ thực hiện trước và chuẩn bị các đoạn mã nguồn để biên dịch. Chương trình tiền xử lý này sẽ tìm trong mã nguồn các kí hiệu chỉ dẫn biên dịch đặc biệt, tất cả các chỉ dẫn biên dịch này đều được bắt đầu với dấu rào (#). Các chỉ dẫn cho phép chúng ta định nghĩa các định danh và kiểm tra các sự tồn tại của các định danh đó.

Bài tập:

Viết chương trình in ra màn hình dòng chữ “Day so le:”, sau đó liệt kê 100 số lẻ. Yêu cầu là sử dụng vòng lặp while hoặc for.

Bài tập nâng cao:

Viết chương trình chuyển đổi 1 số nguyên dương thành nhị phân và chuyển 1 số nhị phân thành số nguyên dương.

Những trọng tâm cần chú ý trong bài:

- Biết kiến thức và các chức năng tiên tiến trên C#;
- Hiểu về các kiểu dữ liệu dựng sẵn của C#;
- Hiểu được các cơ chế thực thi các biến, hằng và các biểu thức trên C#;
- Hiểu về khoảng trắng;
- Biết kiến thức về không gian tên (Namespace);
- Biết kiến thức về các toán tử;
- Biết kiến thức về chỉ dẫn biên dịch;
- Tạo và thực thi được ứng dụng đơn giản trên C#;

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:
    - Biết kiến thức và các chức năng tiên tiến trên C#;
    - Hiểu về các kiểu dữ liệu dựng sẵn của C#;
    - Hiểu được các cơ chế thực thi các biến, hằng và các biểu thức trên C#;
    - Hiểu về khoảng trắng;
    - Biết kiến thức về không gian tên (Namespace);
    - Biết kiến thức về các toán tử;
    - Biết kiến thức về chỉ dẫn biên dịch;

+ Về kỹ năng: Tạo và thực thi được ứng dụng đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Tạo và thực thi được ứng dụng đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 2: XÂY DỰNG LỚP ĐỐI TƯỢNG
Mã bài: MĐ 11 - 03

Giới thiệu:

Trong bài học này các sinh viên sẽ được tìm hiểu về cách thức xây dựng một lớp đối tượng

Mục tiêu:

+ Biết được kỹ năng tạo lớp, tạo đối tượng;
+ Biết kiến thức và kỹ năng về các phương thức, các thành phần static;
+ Biết kiến thức và kỹ năng về tham số và các phương thức nạp chồng;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập.

Nội dung chính:

1. Lớp và đối tượng

Định nghĩa lớp

Người lập trình tạo ra các kiểu dữ liệu mới bằng cách xây dựng các lớp đối tượng và đó cũng chính là các vấn đề chúng ta cần thảo luận trong chương này.

Đây là khả năng để tạo ra những kiểu dữ liệu mới, một đặc tính quan trọng của ngôn ngữ lập trình hướng đối tượng. Chúng ta có thể xây dựng những kiểu dữ liệu mới trong ngôn ngữ C# bằng cách khai báo và định nghĩa những lớp. Ngoài ra ta cũng có thể định nghĩa các kiểu dữ liệu với những giao diện (interface) sẽ được bàn trong Chương 8 sau. Thể hiện của một lớp được gọi là những đối tượng (object). Những đối tượng này được tạo trong bộ nhớ khi chương trình được thực hiện.

Sự khác nhau giữa một lớp và một đối tượng cũng giống như sự khác nhau giữa khái niệm giữa loài mèo và một con mèo Mun đang nằm bên chân của ta. Chúng ta không thể đụng chạm hay đùa giỡn với khái niệm mèo nhưng có thể thực hiện điều đó được với mèo Mun, nó là một thực thể sống động, chứ không trừu tượng như khái niệm họ loài mèo.

Một họ mèo mô tả những con mèo có các đặc tính: có trọng lượng, có chiều cao, màu mắt, màu lông,...chúng cũng có hành động như là ăn ngủ, leo trèo,...một con mèo, ví dụ như mèo Mun chẳng hạn, nó cũng có trọng lượng xác định là 5 kg, chiều cao 15 cm, màu mắt đen, lông đen...Nó cũng có những khả năng như ăn ngủ leo trèo,..

Lợi ích to lớn của những lớp trong ngôn ngữ lập trình là khả năng đóng gói các thuộc tính và tính chất của một thực thể trong một khối đơn, tự có nghĩa, tự khả năng duy trì . Ví dụ khi chúng ta muốn sắp nội dung những thể hiện hay đối tượng của lớp điều khiển ListBox trên Windows, chỉ cần gọi các đối tượng này thì chúng sẽ tự sắp xếp, còn việc chúng làm ra sao thì ta không quan tâm, và cũng chỉ cần biết bấy nhiêu đó thôi.

Đóng gói cùng với đa hình (polymorphism) và kế thừa (inheritance) là các thuộc tính chính yếu của bất kỳ một ngôn ngữ lập trình hướng đối tượng nào.

Chương này sẽ trình bày các đặc tính của ngôn ngữ lập trình C# để xây dựng các lớp đối tượng. Thành phần của một lớp, các hành vi và các thuộc tính, được xem như là thành viên của lớp (class member). Tiếp theo chương cũng trình này khái niệm về phương thức (method) được dùng để định nghĩa hành vi của một lớp, và trạng thái của các biến thành viên hoạt động trong một lớp. Một đặc tính mới mà ngôn ngữ C# đưa ra để xây dựng lớp là khái niệm thuộc tính (property), thành phần thuộc tính này hoạt động giống như cách phương thức để tạo một lớp, nhưng bản chất của phương thức này là tạo một lớp giao diện cho bên ngoài tương tác với biến thành viên một cách gián tiếp, ta sẽ bàn sâu vấn đề này trong chương.

Định nghĩa lớp

Để định nghĩa một kiểu dữ liệu mới hay một lớp đầu tiên phải khai báo rồi sau đó mới định nghĩa các thuộc tính và phương thức của kiểu dữ liệu đó. Khai báo một lớp bằng cách sử dụng từ khoá class. Cú pháp đầy đủ của khai báo một lớp như sau:
[Thuộc tính] [Bổ sung truy cập] class <Định danh lớp> [:
Lớp cơ sở] { <Phần thân của lớp: bao gồm định nghĩa các thuộc tính và phương thức hành động > }

Thành phần thuộc tính của đối tượng sẽ được trình bày chi tiết trong chương sau, còn thành phần bổ sung truy cập cũng sẽ được trình bày tiếp ngay mục dưới. Định danh lớp chính là tên của lớp do người xây dựng chương trình tạo ra. Lớp cơ sở là lớp mà đối tượng sẽ kế thừa để phát triển ta sẽ bàn sau. Tất cả các thành viên của lớp được định nghĩa bên trong thân của lớp,phần thân này sẽ được bao bọc bởi hai dấu ({}).

Trong ngôn ngữ C# phần kết thúc của lớp không có đấu chấm phẩy giống như khai báo lớp trong ngôn ngữ C/C++. Tuy nhiên nếu người lập trình thêm vào thì trình biên dịch C# vẫn chấp nhận mà không đưa ra cảnh báo lỗi.

Trong C#, mọi chuyện đều xảy ra trong một lớp. Như các ví dụ mà chúng ta đã tìm hiểu trong chương 3, các hàm điều được đưa vào trong một lớp, kể cả hàm đầu vào của chương trình (hàm Main()):

```
public class Tester { public static int Main() { //.... } }
```

Điều cần nói ở đây là chúng ta chưa tạo bất cứ thể hiện nào của lớp, tức là tạo đối tượng cho lớp Tester. Điều gì khác nhau giữa một lớp và thể hiện của lớp? để trả lới cho câu hỏi này chúng ta bắt đầu xem xét sự khác nhau giữa kiểu dữ liệu int và một biến kiểu int
. Ta có viết như sau:

```
int var1 = 10;
```

tuy nhiên ta không thể viết được

```
int = 10;
```

Ta không thể gán giá trị cho một kiểu dữ liệu, thay vào đó ta chỉ được gán dữ liệu cho một đối tượng của kiểu dữ liệu đó, trong trường hợp trên đối tượng là biến var1.
Khi chúng ta tạo một lớp mới, đó chính là việc định nghĩa các thuộc tính và hành vi của tất cả các đối tượng của lớp. Giả sử chúng ta đang lập trình để tạo các điều khiển trong các ứng dụng trên Windows, các điều khiển này giúp cho người dùng tương tác tốt với Windows, như là ListBox, TextBox, ComboBox,...Một trong những điều khiển thông dụng là ListBox, điều khiển này cung cấp một danh sách liệt kê các mục chọn và cho phép người dùng chọn các mục tin trong đó.
ListBox này cũng có các thuộc tính khác nhau nhu: chiều cao, bề dày, vị trí, và màu sắc thể hiện và các hành vi của chúng như: chúng có thể thêm bới mục tin, sắp xếp,...
Ngôn ngữ lập trình hướng đối tượng cho phép chúng ta tạo kiểu dữ liệu mới là lớp ListBox, lớp này bao bọc các thuộc tính cũng như khả năng như: các thuộc tính height, width, location, color, các phương thức hay hành vi như Add(), Remove(), Sort(),...
Chúng ta không thể gán dữ liệu cho kiểu ListBox, thay vào đó đầu tiên ta phải tạo một đối tượng cho lớp đó:

```
ListBox myListBox;
```

Một khi chúng ta đã tạo một thể hiện của lớp ListBox thì ta có thể gán dữ liệu cho thể hiện đó. Tuy nhiên đoạn lệnh trên chưa thể tạo đối tượng trong bộ nhớ được, ta sẽ bàn tiếp. Bây giờ ta sẽ tìm hiểu cách tạo một lớp và tạo các thể hiện thông qua ví dụ minh họa 4.1. Ví dụ này tạo một lớp có chức năng hiểu thị thời gian trong một ngày. Lớp này có hành vi thể hiện ngày, tháng, năm, giờ, phút, giây hiện hành. Để làm được điều trên

thì lớp này có 6 thuộc tính hay còn gọi là biến thành viên, cùng với một phương thức như sau:
Tạo một lớp Thoigian đơn giản như sau.

```
---------------------------------------------------------------------------
using System; public class ThoiGian { public void ThoiGianHienHanh() { Console.WriteLine("Hien thi thoi gian hien hanh");
}
// Các biến thành viên int Nam; int Thang; int Ngay; int Gio; int Phut; int Giay; } public class Tester { static void Main() { ThoiGian t = new ThoiGian();
t.ThoiGianHienHanh(); }
}
---------------------------------------------------------------------------
```

Kết quả:

```
Hien thi thoi gian hien hanh
---------------------------------------------------------------------------
```

Lớp ThoiGian chỉ có một phương thức chính là hàm ThoiGianHienHanh(), phần thân của phương thức này được định nghĩa bên trong của lớp ThoiGian. Điều này khác với ngôn ngữ C++, C# không đòi hỏi phải khai báo trước khi định nghĩa một phương thức, và cũng không hỗ trợ việc khai báo phương thức trong một tập tin và sau đó định nghĩa ở một tập tin khác.
C# không có các tập tin tiêu đề, do vậy tất cả các phương thức được định nghĩa hoàn toàn bên trong của lớp. Phần cuối của định nghĩa lớp là phần khai báo các biến thành viên: Nam, Thang, Ngay, Gio, Phut, va Giay.
Sau khi định nghĩa xong lớp ThoiGian, thì tiếp theo là phần định nghĩa lớp Tester, lớp này có chứa một hàm khá thân thiện với chúng ta là hàm Main(). Bên trong hàm Main có một thể hiện của lớp ThoiGian được tạo ra và gán giá trị cho đối tượng t. Bởi vì t là thể hiện của đối tượng ThoiGian, nên hàm Main() có thể sử dụng phương thức của t:
t.ThoiGianHienHanh();

Thuộc tính truy cập

Thuộc tính truy cập quyết định khả năng các phương thức của lớp bao gồm việc các phương thức của lớp khác có thể nhìn thấy và sử dụng các biến thành viên hay những phương thức bên trong lớp. Bảng 4.1 tóm tắt các thuộc tính truy cập của một lớp trong C#.

Thuộc tính truy cập

Thuộc tính	Giới hạn truy cập
public	Không hạn chế. Những thành viên được đánh dấupublic có thể được dùng bởi bất kì các phương thức của lớp bao gồm những lớp khác.
private	Thành viên trong một lớp A được đánh dấu là privatethì chỉ được truy cập bởi các phương thức của lớp A.
protected	Thành viên trong lớp A được đánh dấu là protectedthì chỉ được các phương thức bên trong lớp A và những phương thức dẫn xuất từ lớp A truy cập.

internal	Thành viên trong lớp A được đánh dấu là internal thì được truy cập bởi những phương thức của bất cứ lớpnào trong cùng khối hợp ngữ với A.
protected internal	Thành viên trong lớp A được đánh dấu là protected internal được truy cập bởi các phương thức của lớp A, các phương thức của lớp dẫn xuất của A, và bất cứ lớpnào trong cùng khối hợp ngữ của A.

Mong muốn chung là thiết kế các biến thành viên của lớp ở thuộc tính **private**. Khi đó chỉ có phương thức thành viên của lớp truy cập được giá trị của biến. C# xem thuộc tính **private** là mặc định nên ta không khai báo thuộc tính truy cập cho 6 biến nên mặc định chúng là **private**:

// Các biến thành viên private int Nam; int Thang; int
Ngay; int Gio; int Phut; int Giay;

Do lớp Tester và phương thức thành viên ThoiGianHienHanh của lớp ThoiGian được khai báo là public nên bất kỳ lớp nào cũng có thể truy cập được.

Ghi chú: Thói quen lập trình tốt là khai báo tường minh các thuộc tính truy cập của biến thành viên hay các phương thức trong một lớp. Mặc dù chúng ta biết chắc chắn rằng các thành viên của lớp là được khai báo private mặc định. Việc khai báo tường minh này sẽ làm cho chương trình dễ hiểu, rõ ràng và tự nhiên hơn.

Tham số của phương thức

Trong các ngôn ngữ lập trình thì tham số và đối mục được xem là như nhau, cũng tương tự khi đang nói về ngôn ngữ hướng đối tượng thì ta gọi một hàm là một phương thức hay hành vi. Tất cả các tên này điều tương đồng với nhau.

Một phương thức có thể lấy bất kỳ số lượng tham số nào, Các tham số này theo sau bởi tên của phương thức và được bao bọc bên trong dấu ngoặc tròn (). Mỗi tham số phải khai báo kèm với kiểu dữ liệu. ví dụ ta có một khai báo định nghĩa một phương thức có tên là Method, phương thức không trả về giá trị nào cả (khai báo giá trị trả về là void), và có hai tham số là một kiểu int và button:

void Method(int param1, button param2) { //... }

Bên trong thân của phương thức, các tham số này được xem như những biến cục bộ, giống như là ta khai báo biến bên trong phương thức và khởi tạo giá trị bằng giá trị của tham số truyền vào. Ví dụ 4.2 minh họa việc truyền tham số vào một phương thức, trong trường hợp này thì hai tham số của kiểu là int và float.

Truyền tham số cho phương thức.

---

using System; public class Class1 { public void SomeMethod(int p1, float p2)
{
Console.WriteLine("Ham nhan duoc hai tham so: {0} va {1}", p1,p2); } } public class Tester { static void Main() { int var1 = 5; float var2 = 10.5f; Class1 c = new Class1();
c.SomeMethod(var1, var2); }
}

Kết quả:

Ham nhan duoc hai tham so: 5 va 10.5

Phương thức SomeMethod sẽ lấy hai tham số int và float rồi hiển thị chúng ta màn hình bằng việc dùng hàm Console.WriteLine(). Những tham số này có tên là p1 và p2 được xem như là biến cục bộ bên trong của phương thức.

Trong phương thức gọi Main, có hai biến cục bộ được tạo ra là var1 và var2. Khi hai biến này được truyền cho phương thức SomeMethod thì chúng được ánh xạ thành hai tham số p1 và p2 theo thứ tự danh sách biến đưa vào.

Tạo đối tượng

Trong Chương trước có đề cập đến sự khác nhau giữa kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu. Những kiểu dữ liệu chuẩn của C# như int, char, float,… là những kiểu dữ liệu giá trị,và các biến được tạo ra từ các kiểu dữ liệu này được lưu trên stack. Tuy nhiên, với các đối tượng kiểu dữ liệu tham chiếu thì được tạo ra trên heap, sử dụng từ khóa new để tạo một đối tượng:
ThoiGian t = new ThoiGian();
t thật sự không chứa giá trị của đối tượng ThoiGian, nó chỉ chứa địa chỉ của đối tượng được tạo ra trên heap, do vậy t chỉ chứa tham chiếu đến một đối tượng mà thôi.

Bộ khởi dựng

Thử xem lại ví dụ minh họa bài trước, câu lệnh tạo một đối tượng cho lớp ThoiGian tương tự như việc gọi thực hiện một phương thức:
ThoiGian t = new ThoiGian();
Đúng như vậy, một phương thức sẽ được gọi thực hiện khi chúng ta tạo một đối tượng. Phương thức này được gọi là bộ khởi dựng (constructor). Các phương thức này được định nghĩa khi xây dựng lớp, nếu ta không tạo ra thì CLR sẽ thay mặt chúng ta mà tạo phương thức khởi dựng một cách mặc định. Chức năng của bộ khởi dựng là tạo ra đối tượng được xác định bởi một lớp và đặt trạng thái này hợp lệ. Trước khi bộ khởi dựng được thực hiện thì đối tượng chưa được cấp phát trong bộ nhớ. Sau khi bộ khởi dựng thực hiện hoàn thành thì bộ nhớ sẽ lưu giữ một thể hiện hợp lệ của lớp vừa khai báo.
Lớp ThoiGian trong ví dụ đó không định nghĩa bộ khởi dựng. Do không định nghĩa nên trình biên dịch sẽ cung cấp một bộ khởi dựng cho chúng ta. Phương thức khởi dựng mặc định được tạo ra cho một đối tượng sẽ không thực hiện bất cứ hành động nào, tức là bên trong thân của phương thức rỗng. Các biến thành viên được khởi tạo các giá trị tầm thường như thuộc tính nguyên có giá trị là 0 và chuỗi thì khởi tạo rỗng,..Bảng 4.2 sau tóm tắt các giá trị mặc định được gán cho các kiểu dữ liệu cơ bản.

Giá trị mặc định của kiểu dữ liệu cơ bản.

Kiểu dữ liệu	Giá trị mặc định
int, long, byte,…	0
bool	false
char	‘\0’ (null)
enum	0
reference	null

Thường thường, khi muốn định nghĩa một phương thức khởi dựng riêng ta phải cung cấp các tham số để hàm khởi dựng có thể khởi tạo các giá trị khác ngoài giá trị mặc định cho các đối tượng. Quay lại ví dụ 4.1 giả sử ta muốn truyền thời gian hiện hành: năm, tháng, ngày,…để đối tượng có ý nghĩa hơn.
Để định nghĩa một bộ khởi dựng riêng ta phải khai báo một phương thức có tên giống như tên lớp đã khai báo. Phương thức khởi dựng không có giá trị trả về và được khai báo là public. Nếu phương thức khởi dựng này được truyền tham số thì phải khai báo danh sách tham số giống như khai báo với bất kỳ phương thức nào trong một lớp. Ví dụ 1 được viết lại từ ví dụ bài trước và thêm một bộ khởi dựng riêng, phương phức khởi dựng này sẽ nhận một tham số là một đối tượng kiểu DateTime do C# cung cấp.
Định nghĩa một bộ khởi dựng.

```
---------------------------------------------------------------------------
using System; public class ThoiGian { public void ThoiGianHienHanh() {
Console.WriteLine(" Thoi gian hien hanh la :
{0}/{1}/{2}{3}:{4}:{5}", Ngay, Thang, Nam, Gio, Phut,
Giay);
} // Hàm khởi dựng public ThoiGian( System.DateTime dt )
{
Nam = dt.Year;
Thang = dt.Month;
Ngay = dt.Day;
Gio = dt.Hour;
Phut = dt.Minute; Giay = dt.Second;
} // Biến thành viên private int Nam; int Thang; int Ngay; int Gio; int Phut; int Giay; }
public class Tester { static void Main() {
System.DateTime currentTime = System.DateTime.Now; ThoiGian t = new ThoiGian(
currentTime );
t.ThoiGianHienHanh(); }
}
---------------------------------------------------------------------------
Kết quả:
Thoi gian hien hanh la: 5/6/2002 9:10:20
---------------------------------------------------------------------------
```

Trong ví dụ trên phương thức khởi dựng lấy một đối tượng DateTime và khởi tạo tất cả các biến thành viên dựa trên giá trị của đối tượng này. Khi phương thức này thực hiện xong, một đối tượng ThoiGian được tạo ra và các biến của đối tượng cũng đã được khởi tạo. Hàm ThoiGianHienHanh được gọi trong hàm Main() sẽ hiển thị giá trị thời gian lúc đối tượng được tạo ra.
Chúng ta thử bỏ một số lệnh khởi tạo trong phương thức khởi dựng và cho thực hiện chương trình lại thì các biến không được khởi tạo sẽ có giá trị mặc định là 0, do là biến nguyên. Một biến thành viên kiểu nguyên sẽ được thiết lập giá trị là 0 nếu chúng ta không gán nó trong phương thức khởi dựng. Chú ý rằng kiểu dữ liệu giá trị không thể không được khởi tạo, nếu ta không khởi tạo thì trình biên dịch sẽ cung cấp các giá trị mặc định theo bảng 1.
Ngoài ra trong chương trình trên có sử dụng đối tượng của lớp DateTime, lớp DateTime này được cung cấp bởi thư viện System, lớp này cũng cung cấp các biến thành viên public như: Year, Month, Day, Hour, Minute, và Second tương tự như lớp ThoiGian của

chúng ta. Thêm vào đó là lớp này có đưa ra một phương thức thành viên tĩnh tên là Now, phương thức Now sẽ trả về một tham chiếu đến một thể hiện của một đối tượng DateTime được khởi tạo với thời gian hiện hành. Theo như trên khi lệnh :
System.DataTime currentTime = System.DateTime.Now();
được thực hiện thì phương thức tĩnh Now() sẽ tạo ra một đối tượng DateTime trên bộ nhớ heap và trả về một tham chiếu và tham chiếu này được gán cho biến đối tượng currentTime.
Sau khi đối tượng currentTime được tạo thì câu lệnh tiếp theo sẽ thực hiện việc truyền đối tượng currentTime cho phương thức khởi dựng để tạo một đối tượng ThoiGian:
ThoiGian t = new ThoiGian(currentTime);
Bên trong phương thức khởi dựng này tham số dt sẽ tham chiếu đến đối tượng DateTime là đối tượng vừa tạo mà currentTime cũng tham chiếu. Nói cách khác lúc này tham số dt và currentTime cùng tham chiếu đến một đối tượng DateTime trong bộ nhớ. Nhờ vậy phương thức khởi dựng ThoiGian có thể truy cập được các biến thành viên public của đối tượng DateTime được tạo trong hàm Main().
Có một sự nhấn mạnh ở đây là đối tượng DateTime được truyền cho bộ dựng ThoiGian chính là đối tượng đã được tạo trong hàm Main và là kiểu dữ liệu tham chiếu. Do vậy khi thực hiện truyền tham số là một kiểu dữ liệu tham chiếu thì con trỏ được ánh xạ qua chứ hoàn toàn không có một đối tượng nào được sao chép lại.

Khởi tạo biến thành viên

Các biến thành viên có thể được khởi tạo trực tiếp khi khai báo trong quá trình khởi tạo, thay vì phải thực hiện việc khởi tạo các biến trong bộ khởi dựng. Để thực hiện việc khởi tạo này rất đơn giản là việc sử dụng phép gán giá trị cho một biến:
private int Giay = 30; // Khởi tạo
Việc khởi tạo biến thành viên sẽ rất có ý nghĩa, vì khi xác định giá trị khởi tạo như vậy thì biến sẽ không nhận giá trị mặc định mà trình biên dịch cung cấp. Khi đó nếu các biến này không được gán lại trong các phương thức khởi dựng thì nó sẽ có giá trị mà ta đã khởi tạo. Ví dụ 4.4 minh họa việc khởi tạo biến thành viên khi khai báo. Trong ví dụ này sẽ có hai bộ dựng ngoài bộ dựng mặc định mà trình biên dịch cung cấp, một bộ dựng thực hiện việc gán giá trị cho tất cả các biến thành viên, còn bộ dựng thứ hai thì cũng tương tự nhưng sẽ không gán giá trị cho biến Giay.
Minh hoạ sử dụng khởi tạo biến thành viên.

```
---------------------------------------------------------------------------
public class ThoiGian { public void ThoiGianHienHanh() {
System.DateTime now = System.DateTime.Now;
System.Console.WriteLine("\n Hien tai: \t {0}/{1}/{2}
{3}:{4}:{5}",
now.Day, now.Month, now.Year, now.Hour, now.Minute, now.Second);
System.Console.WriteLine(" Thoi Gian:\t
{0}/{1}/{2} {3}:{4}:{5}",
Ngay, Thang, Nam, Gio, Phut, Giay);
} public ThoiGian( System.DateTime dt) {
Nam = dt.Year;
Thang = dt.Month;
Ngay = dt.Day;
Gio = dt.Hour;
Phut = dt.Minute; Giay = dt.Second; // có gán cho biến thành viên Giay
```

```
}
public ThoiGian(int Year, int Month, int Date, int Hour, int Minute) {
Nam = Year;
Thang = Month;
Ngay = Date;
Gio = Hour; Phut = Minute; } private int Nam; private int Thang; private int Ngay; private int Gio; private int Phut; private int Giay = 30 ; // biến được khởi tạo. } public class Tester { static void Main() {
System.DateTime currentTime = System.DateTime.Now; ThoiGian t1 = new ThoiGian( currentTime ); t1.ThoiGianHienHanh(); ThoiGian t2 = new ThoiGian(2001,7,3,10,5); t2.ThoiGianHienHanh(); }
}
```

Kết quả:

```
Hientai: 5/6/2002 10:15”5
Thoigian: 5/6/2002 10:15:5
Hientai: 5/6/2002 10:15”5
Thoigian: 3/7/2001 10:5:30
```

Nếu không khởi tạo giá trị của biến thành viên thì bộ khởi dựng mặc định sẽ khởi tạo giá trị là 0 mặc định cho biến thành viên có kiểu nguyên. Tuy nhiên, trong trường hợp này biến thành viên Giay được khởi tạo giá trị 30:

```
Giay = 30; // Khởi tạo
```

Trong trường hợp bộ khởi tạo thứ hai không truyền giá trị cho biến Giay nên biến này vẫn lấy giá trị mà ta đã khởi tạo ban đầu là 30:

```
ThoiGian t2 = new ThoiGian(2001, 7, 3, 10, 5); t2.ThoiGianHienHanh();
```

Ngược lại, nếu một giá trị được gán cho biến Giay như trong bộ khởi tạo thứ nhất thì giá trị mới này sẽ được chồng lên giá trị khởi tạo.

Trong ví dụ trên lần đầu tiên tạo đối tượng ThoiGian do ta truyền vào đối tượng DateTime nên hàm khởi dựng thứ nhất được thực hiện, hàm này sẽ gán giá trị 5 cho biến Giay. Còn khi tạo đối tượng ThoiGian thứ hai, hàm khởi dựng thứ hai được thực hiện, hàm này không gán giá trị cho biến Giay nên biến này vẫn còn lưu giữ lại giá trị 30 khi khởi tạo ban đầu.

Bộ khởi dựng sao chép

Bộ khởi dựng sao chép thực hiện việc tạo một đối tượng mới bằng cách sao chép tất cả các biến từ một đối tượng đã có và cùng một kiểu dữ liệu. Ví dụ chúng ta muốn đưa một đối tượng ThoiGian vào bộ khởi dựng lớp ThoiGian để tạo một đối tượng ThoiGian mới có cùng giá trị với đối tượng ThoiGian cũ. Hai đối tượng này hoàn toàn khác nhau và chỉ giống nhau ở giá trị biến thành viên sao khi khởi dựng.

Ngôn ngữ C# không cung cấp bộ khởi dựng sao chép, do đó chúng ta phải tự tạo ra. Việc sao chép các thành phần từ một đối tượng ban đầu cho một đối tượng mới như sau:

```
public ThoiGian( ThoiGian tg) { Nam = tg.Nam; Thang = tg.Thang; Ngay = tg.Ngay; Gio = tg.Gio; Phut = tg.Phut; Giay = tg.Giay; }
```

Khi đó ta có thể sao chép từ một đối tượng ThoiGian đã hiện hữu như sau:

```
ThoiGian t2 = new ThoiGian( t1 );
```

Trong đó t1 là đối tượng ThoiGian đã tồn tại, sau khi lệnh trên thực hiện xong thì đối tượng t2 được tạo ra như bản sao của đối tượng t1.

Từ khóa this

Từ khóa this được dùng để tham chiếu đến thể hiện hiện hành của một đối tượng. Tham chiếu this này được xem là con trỏ ẩn đến tất các phương thức không có thuộc tính tĩnh trong một lớp. Mỗi phương thức có thể tham chiếu đến những phương thức khác và các biến thành viên thông qua tham chiếu this này.
Tham chiếu this này được sử dụng thường xuyên theo ba cách:
Sử dụng khi các biến thành viên bị che lấp bởi tham số đưa vào, như trường hợp sau:
public void SetYear(int Nam) { this.Nam = Nam; }
Như trong đoạn mã trên phương thức SetYear sẽ thiết lập giá trị của biến thành viên Nam, tuy nhiên do tham số đưa vào có tên là Nam, trùng với biến thành viên, nên ta phải dùng tham chiếu this để xác định rõ các biến thành viên và tham số được truyền vào. Khi đó this.Nam chỉ đến biến thành viên của đối tượng, trong khi Nam chỉ đến tham số.
Sử dụng tham chiếu this để truyền đối tượng hiện hành vào một tham số của một phương thức của đối tượng khác:
public void Method1(OtherClass otherObject) { // Sử dụng tham chiếu this để truyền tham số là bản // thân đối tượng đang thực hiện. otherObject.SetObject(this); }
Như trên cho thấy khi cần truyền một tham số là chính bản thân của đối tượng đang thực hiện thì ta bắt buộc phải dùng tham chiếu this để truyền.

2. Sử dụng các thành viên static

Chúng ta có thể truy cập đến thành viên tĩnh của một lớp thông qua tên lớp đã được khai báo. Ví dụ chúng ta có một lớp tên là Button và có hai thể hiện của lớp tên là btnUpdate và btnDelete. Và giả sử lớp Button này có một phương thức tĩnh là Show(). Để truy cập phương thức tĩnh này ta viết :
Button.Show();
Đúng hơn là viết:
btnUpdate.Show();
Trong ngôn ngữ C# không cho phép truy cập đến các phương thức tĩnh và các biến thành viên tĩnh thông qua một thể hiện, nếu chúng ta cố làm điều đó thì trình biên dịch C# sẽ báo lỗi, điều này khác với ngôn ngữ C++.
Trong một số ngôn ngữ thì có sự phân chia giữa phương thức của lớp và các phương thức khác (toàn cục) tồn tại bên ngoài không phụ thuộc bất cứ một lớp nào. Tuy nhiên, điều này không cho phép trong C#, ngôn ngữ C# không cho phép tạo các phương thức bên ngoài của lớp, nhưng ta có thể tạo được các phương thức giống như vậy bằng cách tạo các phương thức tĩnh bên trong một lớp.
Phương thức tĩnh hoạt động ít nhiều giống như phương thức toàn cục, ta truy cập phương thức này mà không cần phải tạo bất cứ thể hiện hay đối tượng của lớp chứa phương thức toàn cục. Tuy nhiên, lợi ích của phương thức tĩnh vượt xa phương thức toàn cục vì phương thức tĩnh được bao bọc trong phạm vi của một lớp nơi nó được định nghĩa, do vậy ta sẽ không gặp tình trạng lộn xộn giữa các phương thức trùng tên do chúng được đặt trong namespace.
Chúng ta không nên bị cám dỗ bởi việc tạo ra một lớp chứa toàn bộ các phương thức linh tinh. Điều này có thể tiện cho công việc lập trình nhưng sẽ điều không mong muốn và giảm tính ý nghĩa của việc thiết kế hướng đối tượng. Vì đặc tính của việc tạo các đối tượng là xây dựng các phương thức và hành vi xung quanh các thuộc tính hay dữ liệu của đối tượng.

Gọi một phương thức tĩnh

Như chúng ta đã biết phương thức Main() là một phương thức tĩnh. Phương tĩnh được xem như là phần hoạt động của lớp hơn là của thể hiện một lớp. Chúng cũng không cần có một tham chiếu this hay bất cứ thể hiện nào tham chiếu tới.
Phương thức tĩnh không thể truy cập trực tiếp đến các thành viên không có tính chất tĩnh (nonstatic). Như vậy Main() không thể gọi một phương thức không tĩnh bên trong lớp.
Ta xem lại đoạn chương trình minh họa trong ví dụ bài trước :
using System; public class Class1 { public void
SomeMethod(int p1, float p2) { Console.WriteLine("Ham nhan duoc hai tham so: {0} va {1}", p1,p2); } } public class Tester { static void Main() { int var1 = 5; float var2 = 10.5f; Class1 c = new Class1(); c.SomeMethod(var1, var2); } }
Phương thức SomeMethod() là phương thức không tĩnh của lớp Class1, do đó để truy cập được phương thức của lớp này cần phải tạo một thể hiện là một đối tượng cho lớp Class1.
Sau khi tạo thì có thể thông qua đối tượng c ta có thể gọi được được phương thức SomeMethod().

Sử dụng bộ khởi dựng tĩnh

Nếu một lớp khai báo một bộ khởi tạo tĩnh (static constructor), thì được đảm bảo rằng phương thức khởi dựng tĩnh này sẽ được thực hiện trước bất cứ thể hiện nào của lớp được tạo ra.
Chúng ta không thể điều khiển chính xác khi nào thì phương thức khởi dựng tĩnh này được thực hiện. Tuy nhiên ta biết chắc rằng nó sẽ được thực hiện sau khi chương trình chạy và trước bất kì biến đối tượng nào được tạo ra.
Ta có thể thêm một bộ khởi dựng tĩnh cho lớp ThoiGian như sau:
static ThoiGian() { Ten = "Thoi gian"; } Ở đây không có bất cứ
thuộc tính truy cập nào như public trước bộ khởi dựng tĩnh. Thuộc tính truy cập không cho phép theo sau một phương thức khởi dựng tĩnh. Do phương thức tĩnh nên không thể truy cập bất cứ biến thành viên không thuộc loại tĩnh.
Vì vậy biến thành viên Name bên trên cũng phải được khai báo là tĩnh:
private static string Ten;
Cuối cùng ta thêm một dòng vào phương thức ThoiGianHienHanh() của lớp ThoiGian:
public void ThoiGianHienHanh() {
System.Console.WriteLine(" Ten: {0}", Ten);
System.Console.WriteLine(" Thoi Gian:\t {0}/{1}/{2}
{3}:{4}:{5}",Ngay, Thang, Nam, Gio, Phut, Giay); }
Sau khi thay đổi ta biên dịch và chạy chương trình được kết quả sau:
Ten: Thoi Gian
Thoi Gian: 5/6/2002 18:35:20
Mặc dù chương trình thực hiện tốt, nhưng không cần thiết phải tạo ra bộ khởi dựng tĩnh để phục vụ cho mục đích này. Thay vào đó ta có thể dùng chức năng khởi tạo biến thành viên như sau:
private static string Ten = "Thoi Gian";
Tuy nhiên, bộ khởi tạo tĩnh có hữu dụng khi chúng ta cần cài đặt một số công việc mà không thể thực hiện được thông qua chức năng khởi dựng và công việc cài đặt này chỉ được thực hiện duy nhất một lần.

Sử dụng bộ khởi dựng private

Như đã nói ngôn ngữ C# không có phương thức toàn cục và hằng số toàn cục. Do vậy chúng ta có thể tạo ra những lớp tiện ích nhỏ chỉ để chứa các phương thức tĩnh. Cách thực hiện này luôn có hai mặt tốt và không tốt. Nếu chúng ta tạo một lớp tiện ích như vậy và không muốn bất cứ một thể hiện nào được tạo ra. Để ngăn ngừa việc tạo bất cứ thể hiện của lớp ta tạo ra bộ khởi dựng không có tham số và không làm gì cả, tức là bên trong thân của phương thức rỗng, và thêm vào đó phương thức này được đánh dầu là private. Do không có bộ khởi dựng public, nên không thể tạo ra bất cứ thể hiện nào của lớp.

Sử dụng các thuộc tính tĩnh

Một vấn đề đặt ra là làm sao kiểm soát được số thể hiện của một lớp được tạo ra khi thực hiện chương trình. Vì hoàn toàn ta không thể tạo được biến toàn cục để làm công việc đếm số thể hiện của một lớp.
Thông thường các biến thành viên tĩnh được dùng để đếm số thể hiện đã được được tạo ra của một lớp. Cách sử dụng này được áp dụng trong minh họa sau:
Sử dụng thuộc tính tĩnh để đếm số thể hiện.

```
using System; public class Cat { public Cat() { instance++; } public static void HowManyCats() { Console.WriteLine("{0} cats", instance); } private static int instance =0; } public class Tester { static void Main() {
Cat.HowManyCats();
Cat mun = new Cat();
Cat.HowManyCats();
Cat muop = new Cat();
Cat miu = new Cat(); Cat.HowManyCats();
}
}
```

Kết quả:

```
0       cats
1       cats
3 cats
```

Bên trong lớp Cat ta khai báo một biến thành viên tĩnh tên là instance biến này dùng để đếm số thể hiện của lớp Cat, biến này được khởi tạo giá trị 0. Lưu ý rằng biến thành viên tĩnh được xem là thành phần của lớp, không phải là thành viên của thể hiện, do vậy nó sẽ không được khởi tạo bởi trình biên dịch khi tạo các thể hiện. Khởi tạo tường minh là yêu cầu bắt buộc với các biến thành viên tĩnh. Khi một thể hiện được tạo ra thì bộ dựng của lớp Cat sẽ thực hiện tăng biến instance lên một đơn vị.

3. Huỷ đối tượng

Ngôn ngữ C# cung cấp cơ chế thu dọn (garbage collection) và do vậy không cần phải khai báo tường minh các phương thức hủy. Tuy nhiên, khi làm việc với các đoạn mã không được quản lý thì cần phải khai báo tường minh các phương thức hủy để giải phóng các tài nguyên.

C# cung cấp ngầm định một phương thức để thực hiện điều khiển công việc này, phương thức đó là Finalize() hay còn gọi là bộ kết thúc. Phương thức Finalize này sẽ được gọi bởi cơ chế thu dọn khi đối tượng bị hủy.
Phương thức kết thúc chỉ giải phóng các tài nguyên mà đối tượng nắm giữ, và không tham chiếu đến các đối tượng khác. Nếu với những đoạn mã bình thường tức là chứa các tham chiếu kiểm soát được thì không cần thiết phải tạo và thực thi phương thức Finalize(). Chúng ta chỉ làm điều này khi xử lý các tài nguyên không kiểm soát được.
Chúng ta không bao giờ gọi một phương thức Finalize() của một đối tượng một cách trực tiếp, ngoại trừ gọi phương thức này của lớp cơ sở khi ở bên trong phương thức Finalize() của chúng ta. Trình thu dọn sẽ thực hiện việc gọi Finalize() cho chúng ta.

Cách Finalize thực hiện
Bộ thu dọn duy trì một danh sách những đối tượng có phương thức Finalize. Danh sách này được cập nhật mỗi lần khi đối tượng cuối cùng được tạo ra hay bị hủy. Khi một đối tượng trong danh sách kết thúc của bộ thu dọn được chọn đầu tiên. Nó sẽ được đặt vào hàng đợi (queue) cùng với những đối tượng khác đang chờ kết thúc. Sau khi phương thức Finalize của đối tượng thực thi bộ thu dọn sẽ gom lại đối tượng và cập nhật lại danh sách hàng đợi, cũng như là danh sách kết thúc đối tượng.

Bộ hủy của C#

Cú pháp phương thức hủy trong ngôn ngữ C# cũng giống như trong ngôn ngữ C++. Nhưng về hành động cụ thể chúng có nhiều điểm khác nhau. Ta khao báo một phương thức hủy trong C# như sau:
~Class1() {}
Tuy nhiên, trong ngôn ngữ C# thì cú pháp khai báo trên là một shortcut liên kết đến một phương thức kết thúc Finalize được kết với lớp cơ sở, do vậy khi viết
~Class1() { // Thực hiện một số công việc } Cũng tương tự như viết : Class1.Finalize() { // Thực hiện một số công việc base.Finalize(); }
Do sự tương tự như trên nên khả năng dẫn đến sự lộn xộn nhầm lẫn là không tránh khỏi, nên chúng ta phải tránh viết các phương thức hủy và viết các phương thức Finalize tường minh nếu có thể được.

Phương thức Dispose

Như chúng ta đã biết thì việc gọi một phương thức kết thúc Finalize trong C# là không hợp lệ, vì phương thức này dành cho bộ thu dọn thực hiện. Nếu chúng ta xử lý các tài nguyên không kiểm soát như xử lý các handle của tập tin và ta muốn được đóng hay giải phóng nhanh chóng bất cứ lúc nào, ta có thực thi giao diện IDisposable, phần chi tiết IDisposable sẽ được trình bày chi tiết trong Chương 8. Giao diện IDisposable yêu cầu những thành phần thực thi của nó định nghĩa một phương thức tên là Dispose() để thực hiện công việc dọn dẹp mà ta yêu cầu. Ý nghĩa của phương thức Dispose là cho phép chương trình thực hiện các công việc dọn dẹp hay giải phóng tài nguyên mong muốn mà không phải chờ cho đến khi phương thức Finalize() được gọi.

Khi chúng ta cung cấp một phương thức Dispose thì phải ngưng bộ thu dọn gọi phương thức Finalize() trong đối tượng của chúng ta. Để ngưng bộ thu dọn, chúng ta gọi một phương thức tĩnh của lớp GC (garbage collector) là GC.SuppressFinalize() và truyền tham số là tham chiếu this của đối tượng. Và sau đó phương thức Finalize() sử dụng để gọi phương thức Dispose() như đoạn mã sau:
public void Dispose() { // Thực hiện công việc dọn dẹp //
Yêu cầu bộ thu dọc GC trong thực hiện kết thúc
GC.SuppressFinalize(this); } public override void
Finalize() { Dispose(); base.Finalize(); }

Phương thức Close

Khi xây dựng các đối tượng, chúng ta có muốn cung cấp cho người sử dụng phương thức
Close(), vì phương thức Close có vẻ tự nhiên hơn phương thức Dispose trong các đối tượng có liên quan đến xử lý tập tin. Ta có thể xây dựng phương thức Dispose() với thuộc tính là private và phương thức Close() với thuộc tính public. Trong phương thức Close() đơn giản là gọi thực hiện phương thức Dispose().

Câu lệnh using

Khi xây dựng các đối tượng chúng ta không thể chắc chắn được rằng người sử dụng có thể gọi hàm Dispose(). Và cũng không kiểm soát được lúc nào thì bộ thu dọn GC thực hiện việc dọn dẹp. Do đó để cung cấp khả năng mạnh hơn để kiểm soát việc giải phóng tài nguyên thì C# đưa ra cú pháp chỉ dẫn using, cú pháp này đảm bảo phương thức Dispose() sẽ được gọi sớm nhất có thể được. Ý tưởng là khai báo các đối tượng với cú pháp using và sau đó tạo một phạm vi hoạt động cho các đối tượng này trong khối được bao bởi dấu ({}). Khi khối phạm vi này kết thúc, thì phương thức Dispose() của đối tượng sẽ được gọi một cách tự động.
Sử dụng chỉ dẫn using.

---

using System.Drawing; class Tester { public static void Main() { using (Font Afont = new Font("Arial",10.0f)) { // Đoạn mã sử dụng AFont ..... }// Trình biên dịch sẽ gọi Dispose để giải phóng AFont Font TFont = new Font("Tahoma",12.0f); using (TFont) { // Đoạn mã sử dụng TFont ..... }// Trình biên dịch gọi Dispose để giải phóng TFont
}
}

Còn trong phần khai báo thứ hai, một đối tượng Font được tạo bên ngoài câu lệnh using. Khi quyết định dùng đối tượng này ta đặt nó vào câu lệnh using. Và cũng tương tự như trên khi khối câu lệnh using thực hiện xong thì phương thức Dispose() của font được gọi.

4. Truyền tham số và nạp chồng phương thức

Ngôn ngữ C# đưa ra một bổ sung tham số là ref cho phép truyền các đối tượng giá trị vào trong phương thức theo kiểu tham chiếu. Và tham số bổ sung out trong trường hợp muốn truyền dưới dạng tham chiếu mà không cần phải khởi tạo giá trị ban đầu cho tham số truyền. Ngoài ra ngôn ngữ C# còn hỗ trợ bổ sung params cho phép phương thức chấp nhận nhiều số lượng các tham số.

Truyền tham chiếu

Những phương thức chỉ có thể trả về duy nhất một giá trị, mặc dù giá trị này có thể là một tập hợp các giá trị. Nếu chúng ta muốn phương thức trả về nhiều hơn một giá trị thì cách thực hiện là tạo các tham số dưới hình thức tham chiếu. Khi đó trong phương thức ta sẽ xử lý và gán các giá trị mới cho các tham số tham chiếu này, kết quả là sau khi phương thức thực hiện xong ta dùng các tham số truyền vào như là các kết quả trả về.
Ví dụ 4.7 sau minh họa việc truyền tham số tham chiếu cho phương thức.
Trả giá trị trả về thông qua tham số.

```
using System; public class Time { public void
DisplayCurrentTime() { Console.WriteLine("{0}/{1}/{2}/ {3}:{4}:{5}", Date, Month,
Year, Hour, Minute, Second); } public int GetHour() { return Hour; } public void
GetTime(int h, int m, int s) { h = Hour; m = Minute; s = Second; } public Time(
System.DateTime dt) { Year = dt.Year; Month = dt.Month; Date = dt.Day; Hour =
dt.Hour; Minute = dt.Minute; Second = dt.Second; } private int Year; private int Month;
private int Date; private int Hour; private int Minute; private int Second; } public class
Tester { static void Main() { System.DateTime currentTime = System.DateTime.Now;
Time t = new Time( currentTime); t.DisplayCurrentTime(); int theHour = 0; int
theMinute = 0; int theSecond = 0; t.GetTime( theHour, theMinute, theSecond);
System.Console.WriteLine("Current time: {0}:{1}:{2}",theHour, theMinute,
theSecond); } }
```

Kết quả:
8/6/2002 14:15:20
Current time: 0:0:0

Như ta thấy, kết quả xuất ra ở dòng cuối cùng là ba giá trị 0:0:0, rõ ràng phương thức GetTime() không thực hiện như mong muốn là gán giá trị Hour, Minute, Second cho các tham số truyền vào. Tức là ba tham số này được truyền vào dưới dạng giá trị. Do đó để thực hiện như mục đích của chúng ta là lấy các giá trị của Hour, Minute, Second thì phương thức GetTime() có ba tham số được truyền dưới dạng tham chiếu. Ta thực hiện như sau, đầu tiên, thêm là thêm khai báo ref vào trước các tham số trong phương thức GetTime():

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

Điều thay đổi thứ hai là bổ sung cách gọi hàm GetTime để truyền các tham số dưới dạng tham chiếu như sau:

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

Nếu chúng ta không thực hiện bước thứ hai, tức là không đưa từ khóa ref khi gọi hàm thì trình biên dịch C# sẽ báo một lỗi rằng không thể chuyển tham số từ kiểu int sang **kiểu ref int**.

Cuối cùng khi biên dịch lại chương trình ta được kết quả đúng như yêu cầu. Bằng việc khai báo tham số tham chiếu, trình biên dịch sẽ truyền các tham số dưới dạng các tham chiếu, thay cho việc tạo ra một bản sao chép các tham số này. Khi đó các tham số bên trong GetTime() sẽ tham chiếu đến cùng biến đã được khai báo trong hàm Main(). Như vậy mọi sự thay đổi với các biến này điều có hiệu lực tương tự như là thay đổi trong hàm Main().

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

Truyền tham chiếu với biến chưa khởi tạo

Ngôn ngữ C# bắt buộc phải thực hiện một phép gán cho biến trước khi sử dụng, do đó khi khai báo một biến như kiểu cơ bản thì trước khi có lệnh nào sử dụng các biến này thì phải có lệnh thực hiện việc gán giá trị xác định cho biến. Như trong ví dụ 4.7 trên, nếu chúng ta không khởi tạo biến theHour, theMinute, và biến theSecond trước khi truyền như tham số vào phương thức GetTime() thì trình biên dịch sẽ báo lỗi. Nếu chúng ta sửa lại đoạn mã của ví dụ trước như sau:

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

Việc sử dụng các đoạn lệnh trên không phải hoàn toàn vô lý vì mục đích của chúng ta là nhận các giá trị của đối tượng Time, việc khởi tạo giá trị của các biến đưa vào là không cần thiết. Tuy nhiên khi biên dịch với đoạn mã lệnh như trên sẽ được báo các lỗi sau:

```
Use of unassigned local variable ‘theHour’
Use of unassigned local variable ‘theMinute’
Use of unassigned local variable ‘theSecond’
```

Để mở rộng cho yêu cầu trong trường hợp này ngôn ngữ C# cung cấp thêm một bổ sung tham chiếu là out. Khi sử dụng tham chiếu out thì yêu cầu bắt buộc phải khởi tạo các tham số tham chiếu được bỏ qua. Như các tham số trong phương thức GetTime(), các tham số này không cung cấp bất cứ thông tin nào cho phương thức mà chỉ đơn giản là cơ chế nhận thông tin và đưa ra bên ngoài. Do vậy ta có thể đánh dấu tất cả các tham số tham chiếu này là out, khi đó ta sẽ giảm được công việc phải khởi tạo các biến này trước khi đưa vào phương thức. Lưu ý là bên trong phương thức có các tham số tham chiếu out thì các tham số này phải được gán giá trị trước khi trả về. Ta có một số thay đổi cho phương thức GetTime() như sau:

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

và cách gọi mới phương thức GetTime() trong Main():

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

Tóm lại ta có các cách khai báo các tham số trong một phương thức như sau: kiểu dữ liệu giá trị được truyền vào phương thức bằng giá trị. Sử dụng tham chiếu ref để truyền kiểu dữ liệu giá trị vào phương thức dưới dạng tham chiếu, cách này cho phép vừa sử dụng và có khả năng thay đổi các tham số bên trong phương thức được gọi. Tham chiếu out được sử dụng chỉ để trả về giá trị từ một phương thức. Ví dụ sau sử dụng ba kiểu tham số trên.

Sử dụng tham số.

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

```
public void SetTime(int hr, out int min, ref int sec) { // Nếu số giây truyền vào >30 thì tăng số Minute và Second = 0 if ( sec >=30 ) { Minute++; Second = 0;
} Hour = hr; // thiết lập giá trị hr được truyền vào // Trả về giá trị mới cho min và sec min = Minute; sec = Second; } public Time(System.DateTime dt) {
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
Minute = dt.Minute; Second = dt.Second;
}
// biến thành viên private private int Year; private int Month; private int Date; private int Hour; private int Minute; private int Second; } public class Tester { static void Main() { System.DateTime currentTime = System.DateTime.Now; Time t = new Time(currentTime);
t.DisplayCurrentTime(); int theHour = 3; int theMinute; int theSecond = 20;
t.SetTime( theHour, out theMinute, ref theSecond);
Console.WriteLine("The Minute is now: {0} and {1} seconds
",theMinute, theSecond); theSecond = 45;
t.SetTime( theHour, out theMinute, ref theSecond);
Console.WriteLine("The Minute is now: {0} and {1}
seconds",theMinute, theSecond); }
}
```

Kết quả

```
8/6/2002 15:35:24
The Minute is now: 35 and 24 seconds
The Minute is now: 36 and 0 seconds
```

Phương thức SetTime trên đã minh họa việc sử dụng ba kiểu truyền tham số vào một phương thức. Tham số thứ nhất theHour được truyền vào dạng giá trị, mục đích của tham số này là để thiết lập giá trị cho biến thành viên Hour và tham số này không được sử dụng để về bất cứ giá trị nào.

Tham số thứ hai là theMinute được truyền vào phương thức chỉ để nhận giá trị trả về của biến thành viên Minute, do đó tham số này được khai báo với từ khóa out.

Cuối cùng tham số theSecond được truyền vào với khai báo ref, biến tham số này vừa dùng để thiết lập giá trị trong phương thức. Nếu theSecond lớn hơn 30 thì giá trị của biến thành viên Minute tăng thêm một đơn vị và biến thành viên Second được thiết lập về 0. Sau cùng thì theSecond được gán giá trị của biến thành viên Second và được trả về.

Do hai biến theHour và theSecond được sử dụng trong phương thức SetTime nên phải được khởi tạo trước khi truyền vào phương thức. Còn với biến theMinute thì không cần thiết vì nó không được sử dụng trong phương thức mà chỉ nhận giá trị trả về.

Nạp chồng phương thức

Một ký hiệu (signature) của một phương thức được định nghĩa như tên của phương thức cùng với danh sách tham số của phương thức. Hai phương thức khác nhau khi ký hiệu của chúng khác là khác nhau tức là khác nhau khi tên phương thức khác nhau hay danh sách tham số khác nhau. Danh sách tham số được xem là khác nhau bởi số lượng các

tham số hoặc là kiểu dữ liệu của tham số. Ví dụ đoạn mã sau, phương thức thứ nhất khác phương thức thứ hai do số lượng tham số khác nhau. Phương thức thứ hai khác phương thức thứ ba do kiểu dữ liệu tham số khác nhau:
void myMethod(int p1); void myMethod(int p1, int p2); void myMethod(int p1, string p2);
Một lớp có thể có bất cứ số lượng phương thức nào, nhưng mỗi phương thức trong lớp phải có ký hiệu khác với tất cả các phương thức thành viên còn lại của lớp.
Ví dụ sau minh họa lớp Time có hai phương thức khởi dựng, một phương thức nhận tham số là một đối tượng DateTime còn phương thức thứ hai thì nhận sáu tham số nguyên.
Minh họa nạp chồng phương thức khởi dựng.
---
using System; public class Time { public void DisplayCurrentTime() {
Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}", Date, Month,
Year, Hour, Minute, Second);
}
public Time(System.DateTime dt) {
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
Minute = dt.Minute; Second = dt.Second;
}
public Time(int Year, int Month, int Date, int Hour, int
Minute, int Second) { this.Year = Year; this.Month = Month; this.Date = Date; this.Hour = Hour; this.Minute = Minute; this.Second = Second; } private int Month; private int Date; private int Hour; private int Minute; private int Second; } public class Tester { static void Main() { System.DateTime currentTime = System.DateTime.Now; Time t1 = new Time(currentTime); t1.DisplayCurrentTime(); Time t2 = new Time(2002,6,8,18,15,20); t2.DisplayCurrentTime(); }
}

Kết quả:
2/1/2002 17:50:17
8/6/2002 18:15:20

Như chúng ta thấy, lớp Time trong ví dụ minh họa 4.9 có hai phương thức khởi dựng. Nếu hai phương thức có cùng ký hiệu thì trình biên dịch sẽ không thể biết được gọi phương thức nào khi khởi tạo hai đối tượng là t1 và t2. Tuy nhiên, ký hiệu của hai phương thức này khác nhau vì tham số truyền vào khác nhau, do đó trình biên dịch sẽ xác định được phương thức nào được gọi dựa vào các tham số.
Khi thực hiện nạp chồng một phương thức, bắt buộc chúng ta phải thay đổi ký hiệu của phương thức, số tham số, hay kiểu dữ liệu của tham số. Chúng ta cũng có thể toàn quyền thay đổi giá trị trả về, nhưng đây là tùy chọn. Nếu chỉ thay đổi giá trị trả về thì không phải nạp chồng phương thức mà khi đó hai phương thức khác nhau, và nếu tạo ra hai phương thức cùng ký hiệu nhưng khác nhau kiểu giá trị trả về sẽ tạo ra một lỗi biên dịch.
Nạp chồng phương thức.

```
using System; public class Tester { private int Triple( int val) { return 3*val; } private
long Triple(long val) { return 3*val; } public void Test() { int x = 5; int y = Triple(x);
Console.WriteLine("x: {0} y: {1}", x, y); long lx = 10;
long ly = Triple(lx); Console.WriteLine("lx: {0} ly:{1}", lx, ly); } static void Main() {
Tester t = new Tester();
t.Test(); }
}
```

Kết quả: x: 5 y: 15 lx: 10 ly:30

Trong ví dụ này, lớp Tester nạp chồng hai phương thức Triple(), một phương thức nhận tham số nguyên int, phương thức còn lại nhận tham số là số nguyên long. Kiểu giá trị trả về của hai phương thức khác nhau, mặc dù điều này không đòi hỏi nhưng rất thích hợp trong trường hợp này.

5. Đóng gói dữ liệu với thuộc tính

Thuộc tính là khái niệm cho phép truy cập trạng thái của lớp thay vì thông qua truy cập trực tiếp các biến thành viên, nó sẽ đươc thay thế bằng việc thực thi truy cập thông qua phương thức của lớp.

Đây thật sự là một điều lý tưởng. Các thành phần bên ngoài (client) muốn truy cập trạng thái của một đối tượng và không muốn làm việc với những phương thức. Tuy nhiên, người thiết kế lớp muốn dấu trạng thái bên trong của lớp mà anh ta xây dựng, và cung cấp một cách gián tiếp thông qua một phương thức.

Thuộc tính là một đặc tính mới được giới thiệu trong ngôn ngữ C#. Đặc tính này cung cấp khả năng bảo vệ các trường dữ liệu bên trong một lớp bằng việc đọc và viết chúng thông qua thuộc tính. Trong ngôn ngữ khác, điều này có thể được thực hiện thông qua việc tạo các phương thức lấy dữ liệu (getter method) và phương thức thiết lập dữ liệu (setter method).

Thuộc tính được thiết kế nhắm vào hai mục đích: cung cấp một giao diện đơn cho phép truy cập các biến thành viên, Tuy nhiên cách thức thực thi truy cập giống như phương thức trong đó các dữ liệu được che dấu, đảm bảo cho yêu cầu thiết kế hướng đối tượng.

Để hiểu rõ đặc tính này ta sẽ xem ví dụ bên dưới:

Sử dụng thuộc tính.

```
using System; public class Time { public void
DisplayCurrentTime() { Console.WriteLine("Time\t:
{0}/{1}/{2} {3}:{4}:{5}", date, month, year, hour, minute, second); } public Time(
System.DateTime dt) { year = dt.Year; month = dt.Month; date = dt.Day; hour =
dt.Hour; minute = dt.Minute; second = dt.Second; } Public int Hour; { get { return hour;
} set { hour=value; } } //Biến thành viên private private int year; private int month;
private int date; private int hour; private int minute; private int second; } public class
Tester { static void Main() { System.DateTime currentTime = System.DateTime.Now;
Time t = new Time( currentTime ); t.DisplayCurrentTime(); // Lấy dữ liệu từ thuộc tính
Hour int theHour = t.Hour;
Console.WriteLine(" Retrieved the hour: {0}", theHour); theHour++; t.Hour = theHour;
Console.WriteLine("Updated the hour: {0}", theHour); } }
```

Kết quả:

Time : 2/1/2003 17:55:1
Retrieved the hour: 17
Updated the hour: 18

Để khai báo thuộc tính, đầu tiên là khai báo tên thuộc tính để truy cập, tiếp theo là phần thân định nghĩa thuộc tính nằm trong cập dấu ({}). Bên trong thân của thuộc tính là khai báo hai bộ truy cập lấy và thiết lập dữ liệu:
public int Hour { get { return hour; } set { hour=value; } }
Mỗi bộ truy cập được khai báo riêng biệt để làm hai công việc khác nhau là lấy hay thiết lập giá trị cho thuộc tính. Giá trị thuộc tính có thể được lưu trong cơ sở dữ liệu, khi đó trong phần thân của bộ truy cập sẽ thực hiện các công việc tương tác với cơ sở dữ liệu. Hoặc là giá trị thuộc tính được lưu trữ trong các biến thành viên của lớp như trong ví dụ:
private int hour;

Truy cập lấy dữ liệu (get accessor)

Phần khai báo tương tự như một phương thức của lớp dùng để trả về một đối tượng có kiểu dữ liệu của thuộc tính. Trong ví dụ trên, bộ truy cập lấy dữ liệu get của thuộc tính Hour cũng tương tự như một phương thức trả về một giá trị int. Nó trả về giá trị của biến thành viên hour nơi mà giá trị của thuộc tính Hour lưu trữ:
get { return hour; }
Trong ví dụ này, một biến thành viên cục bộ được trả về, nhưng nó cũng có thể truy cập dễ dàng một giá trị nguyên từ cơ sở dữ liệu, hay thực hiện việc tính toán tùy ý.
Bất cứ khi nào chúng ta tham chiếu đến một thuộc tính hay là gán giá trị thuộc tính cho một biến thì bộ truy cập lấy dữ liệu get sẽ được thực hiện để đọc giá trị của thuộc tính:
Time t = new Time(currentTime); int theHour = t.Hour;
Khi lệnh thứ hai được thực hiện thì giá trị của thuộc tính sẽ được trả về, tức là bộ truy cập lấy dữ liệu get sẽ được thực hiện và kết quả là giá trị của thuộc tính được gán cho biến cục bộ theHour.

Bộ truy cập thiết lập dữ liệu (set accessor)

Bộ truy cập này sẽ thiết lập một giá trị mới cho thuộc tính và tương tự như một phương thức trả về một giá trị void. Khi định nghĩa bộ truy cập thiết lập dữ liệu chúng ta phải sử dụng từ khóa value để đại diện cho tham số được truyền vào và được lưu trữ bởi thuộc tính:
set { hour = value; }
Như đã nói trước, do ta đang khai báo thuộc tính lưu trữ dưới dạng biến thành viên nên trong phần thân của bộ truy cập ta chỉ sử dụng biến thành viên mà thôi. Bộ truy cập thiết lập hoàn toàn cho phép chúng ta có thể viết giá trị vào trong cơ sở dữ liệu hay cập nhật bất cứ biến thành viên nào khác của lớp nếu cần thiết.
Khi chúng ta gán một giá trị cho thuộc tính thì bộ truy cập thiết lập dữ liệu set sẽ được tự động thực hiện và một tham số ngầm định sẽ được tạo ra để lưu giá trị mà ta muốn gán:
theHour++;
t.Hour = theHour;
Lợi ích của hướng tiếp cận này cho phép các thành phần bên ngoài (client) có thể tương tác với thuộc tính một cách trực tiếp, mà không phải hy sinh việc che dấu dữ liệu cũng như đặc tính đóng gói dữ liệu trong thiết kế hướng đối tượng.

Thuộc tính chỉ đọc

Giả sử chúng ta muốn tạo một phiên bản khác cho lớp Time cung cấp một số giá trị static để hiển thị ngày và giờ hiện hành. Ví dụ sau minh họa cho cách tiếp cận này.
Sử dụng thuộc tính hằng static.

---

using System; public class RightNow { // Định nghĩa bộ khởi tạo static cho các biến static static RightNow() { System.DateTime dt = System.DateTime.Now;
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
Minute = dt.Minute; Second = dt.Second;
} // Biến thành viên static public static int Year; public static int Month; public static int Date; public static int Hour; public static int Minute; public static int Second; } public class Tester { static void Main() {
Console.WriteLine("This year: {0}", RightNow.Year.ToString());
RightNow.Year = 2003;
Console.WriteLine("This year: {0}", RightNow.Year.ToString());
}
}

Kết quả:
This year: 2002
This year: 2003

Đoạn chương trình trên hoạt động tốt, tuy nhiên cho đến khi có một ai đó thay đổi giá trị của biến thành viên này. Như ta thấy, biến thành Year trên đã được thay đổi đến 2003. Điều này thực sự không như mong muốn của chúng ta.
Chúng ta muốn đánh dấu các thuộc tính tĩnh này không được thay đổi. Nhưng khai báo hằng cũng không được vì biến tĩnh không được khởi tạo cho đến khi phương thức khởi dựng static được thi hành. Do vậy C# cung cấp thêm từ khóa readonly phục vụ chính xác cho mục đích trên. Với ví dụ trên ta có cách khai báo lại như sau:
public static readonly int Year; public static readonly int Month; public static readonly int Date; public static readonly int Hour; public static readonly int Minute; public static readonly int Second;
Khi đó ta phải bỏ lệnh gán biến thành viên Year, vì nếu không sẽ bị báo lỗi:
// RightNow.Year = 2003; // error
Chương trình sau khi biên dịch và thực hiện như mục đích của chúng ta.

Bài tập:

Viết chương trình tạo 1 lớp “Thú cưng”, khai báo các thông tin cơ bản của một thú cưng. Tiến hành tạo các đối tượng sau: con chó, con mèo, con chuột

Bài tập nâng cao:

Trong lớp “Thú cưng” tạo các hành động như: chơi với thú cưng, cho thú cưng ăn, tắm cho thú cưng

Những trọng tâm cần chú ý trong bài:

- Biết được kỹ năng tạo lớp, tạo đối tượng;
- Biết kiến thức và kỹ năng về các phương thức, các thành phần static;
- Biết kiến thức và kỹ năng về tham số và các phương thức nạp chồng;
- Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:
    - Biết được kỹ năng tạo lớp, tạo đối tượng;
    - Biết kiến thức và kỹ năng về các phương thức, các thành phần static;
    - Biết kiến thức và kỹ năng về tham số và các phương thức nạp chồng;

+ Về kỹ năng: Tạo và sử dụng được lớp để hoàn thành bài tập.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Tạo và sử dụng được lớp để hoàn thành bài tập.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 3: KẾ THỪA – ĐA HÌNH
Mã bài: MĐ 11 - 04

Giới thiệu:
Hai tính năng quan trọng của đối tượng là kế thừa và đa hình
Mục tiêu:
+ Biết các kiến thức về tính kế thừa và đa hình trên C#;
+ Biết các kiến thức về lớp trừu tượng;
+ Hiểu được các kiến thức và kỹ năng về các phương thức, các thành phần static;
+ Hiểu được các kiến thức và kỹ năng về tham số và các phương thức nạp chồng;
+ Nghiêm túc, sáng tạo trong quá trình tiếp thu lý thuyết và áp dụng làm các bài tập.

Nội dung chính:

1. Sự kế thừa

Trong ngôn ngữ C#, quan hệ đặc biệt hóa được thực thi bằng cách sử dụng sự kế thừa. Đây không phải là cách duy nhất để thực thi đặc biệt hóa, nhưng nó là cách chung nhất và tự nhiên nhất để thực thi quan hệ này.
Trong mô hình trước, ta có thể nói ListBox kế thừa hay được dẫn xuất từ Window. Window được xem như là lớp cơ sở, và ListBox được xem như là lớp dẫn xuất. Như vậy, ListBox dẫn xuất tất cả các thuộc tính và hành vi từ lớp Window và thêm những phần đặc biệt riêng để xác nhận ListBox.

1.1. Thực thi kế thừa

Trong ngôn ngữ C# để tạo một lớp dẫn xuất từ một lớp ta thêm dấu hai chấm vào sau tên lớp dẫn xuất và trước tên lớp cơ sở:
public class ListBox : Window
Đoạn lệnh trên khai báo một lớp mới tên là ListBox, lớp này được dẫn xuất từ Window. Chúng ta có thể đọc dấu hai chấm có thể được đọc như là "dẫn xuất từ".
Lớp dẫn xuất sẽ kế thừa tất cả các thành viên của lớp cơ sở, bao gồm tất cả các phương thức và biến thành viên của lớp cơ sở. Lớp dẫn xuất được tự do thực thi các phiên bản của một phương thức của lớp cơ sở. Lớp dẫn xuất cũng có thể tạo một phương thức mới bằng việc đánh dấu với từ khóa new. Ví dụ sau minh họa việc tạo và sử dụng các lớp cơ sở và dẫn xuất.
Sử dụng lớp dẫn xuất.

```
---------------------------------------------------------------------------
using System; public class Window {
// Hàm khởi dựng lấy hai số nguyên chỉ // đến vị trí của cửa sổ trên console public Window( int top, int left)
{ this.top = top; this.left = left; } // mô phỏng vẽ cửa sổ public void DrawWindow() { Console.WriteLine("Drawing Window at {0},{1}",top,lep);
}
// Có hai biến thành viên private do đó // hai biến này sẽ không thấy bên trong lớp // dẫn xuất. private int top; private int left; } // ListBox dẫn xuất từ Window public class ListBox: Window {
// Khởi dựng có tham số
public ListBox(int top, int left,string theContents)
:base(top, left)
// gọi khởi dựng của lớp cơ sở
{ mListBoxContents = theContents; }
// Tạo một phiên bản mới cho phương thức DrawWindow // vì trong lớp dẫn xuất muốn thay đổi hành vi thực hiện // bên trong phương thức này public new void DrawWindow()
{ base.DrawWindow();
Console.WriteLine(" ListBox write: {0}", mListBoxContents);
} // biến thành viên private private string mListBoxContents; } public class Tester { public static void Main() {
//tạo đối tượng cho lớp cơ sở Window w=new Window(5,10);
w.DrawWindow();
//tạo đối tượng cho lớp dẫn xuất
ListBox lb = new ListBox(20,10,"Hello world!"); Lb.DrawWindow();
}
}
---------------------------------------------------------------------------
Kết quả:
Drawing Window at: 5, 10
Drawing Window at: 20, 10
ListBox write: Hello world!
---------------------------------------------------------------------------
```

Ví dụ trên bắt đầu với việc khai báo một lớp cơ sở tên Window. Lớp này thực thi một phương thức khởi dựng và một phương thức đơn giản DrawWindow. Lớp có hai biến thành viên private là top và left, hai biến này do khai báo là private nên chỉ sử dụng bên trong của lớp Window, các lớp dẫn xuất sẽ không truy cập được. ta sẽ bàn tiếp về ví dụ này trong phần tiếp theo.

1.2. Gọi phương thức khởi dựng của lớp cơ sở

Trong ví dụ trên, một lớp mới tên là ListBox được dẫn xuất từ lớp cơ sở Window, lớp ListBox có một phương thức khởi dựng lấy ba tham số. Trong phương thức khởi dựng của lớp dẫn xuất này có gọi phương thức khởi dựng của lớp cơ sở. Cách gọi được thực hiện bằng việc đặt dấu hai chấm ngay sau phần khai báo danh sách tham số và tham chiếu đến lớp cơ sở thông qua từ khóa base:

```
public ListBox( int theTop, int theLeft,string theContents):
base( theTop, theLeft) // gọi khởi tạo lớp cơ sở
```

Bởi vì các lớp không được kế thừa các phương thức khởi dựng của lớp cơ sở, do đó lớp dẫn xuất phải thực thi phương thức khởi dựng riêng của nó. Và chỉ có thể sử dụng phương thức khởi dựng của lớp cơ sở thông qua việc gọi tường minh.
Một điều lưu ý trong ví dụ trên là việc lớp ListBox thực thi một phiên bản mới của phương thức DrawWindow():
public new void DrawWindow()
Từ khóa new được sử dụng ở đây để chỉ ra rằng người lập trình đang tạo ra một phiên bản mới cho phương thức này bên trong lớp dẫn xuất.
Nếu lớp cơ sở có phương thức khởi dựng mặc định, thì lớp dẫn xuất không cần bắt buộc phải gọi phương thức khởi dựng của lớp cơ sở một cách tường minh. Thay vào đó phương thức khởi dựng mặc định của lớp cơ sở sẽ được gọi một cách ngầm định. Tuy nhiên, nếu lớp cơ sở không có phương thức khởi dựng mặc định, thì tất cả các lớp dẫn xuất của nó phải gọi phương thức khởi dựng của lớp cơ sở một cách tường minh thông qua việc sử dụng từ khóa base.
Cũng như thảo luận trong chương 4, nếu chúng ta không khai báo bất cứ phương thức khởi dựng nào, thì trình biên dịch sẽ tạo riêng một phương thức khởi dựng cho chúng ta. Khi mà chúng ta viết riêng các phương thức khởi dựng hay là sử dụng phương thức khởi dựng mặc định do trình biên dịch cung cấp hay không thì phương thức khởi dựng mặc định không lấy một tham số nào hết. Tuy nhiên, lưu ý rằng khi ta tạo bất cứ phương thức khởi dựng nào thì trình biên dịch sẽ không cung cấp phương thức khởi dựng cho chúng ta.

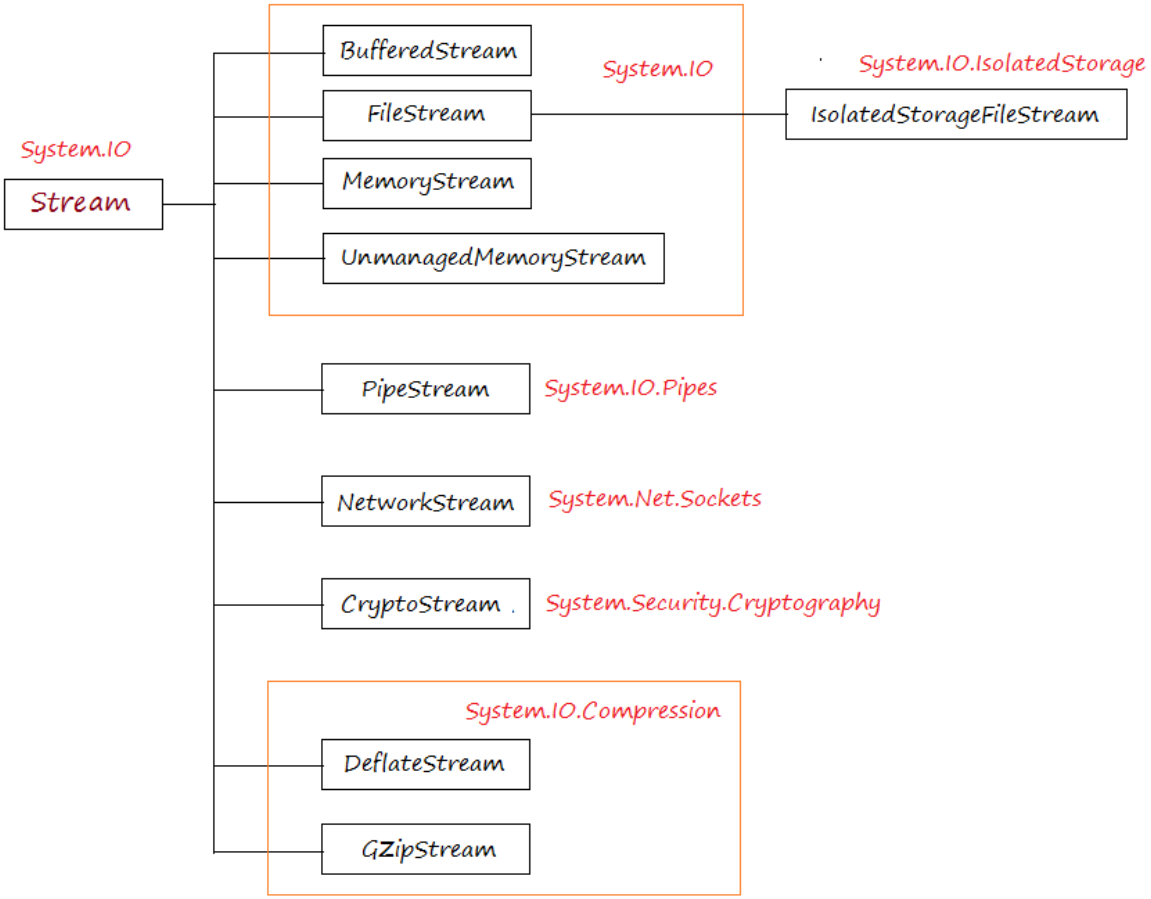
1.3. Gọi phương thức của lớp cơ sở

Trong ví dụ 5.1, phương thức DrawWindow() của lớp ListBox sẽ làm ẩn và thay thế phương thức DrawWindow của lớp cơ sở Window. Khi chúng ta gọi phương thức DrawWindow của một đối tượng của lớp ListBox thì phương thức ListBox.DrawWindow() sẽ được thực hiện, không phải phương thức Window.DrawWindow() của lớp cơ sở Window. Tuy nhiên, ta có thể gọi phương thức DrawWindow() của lớp cơ sở thông qua từ khóa base:
base.DrawWindow(); // gọi phương thức cơ sở
Từ khóa base chỉ đến lớp cơ sở cho đối tượng hiện hành.

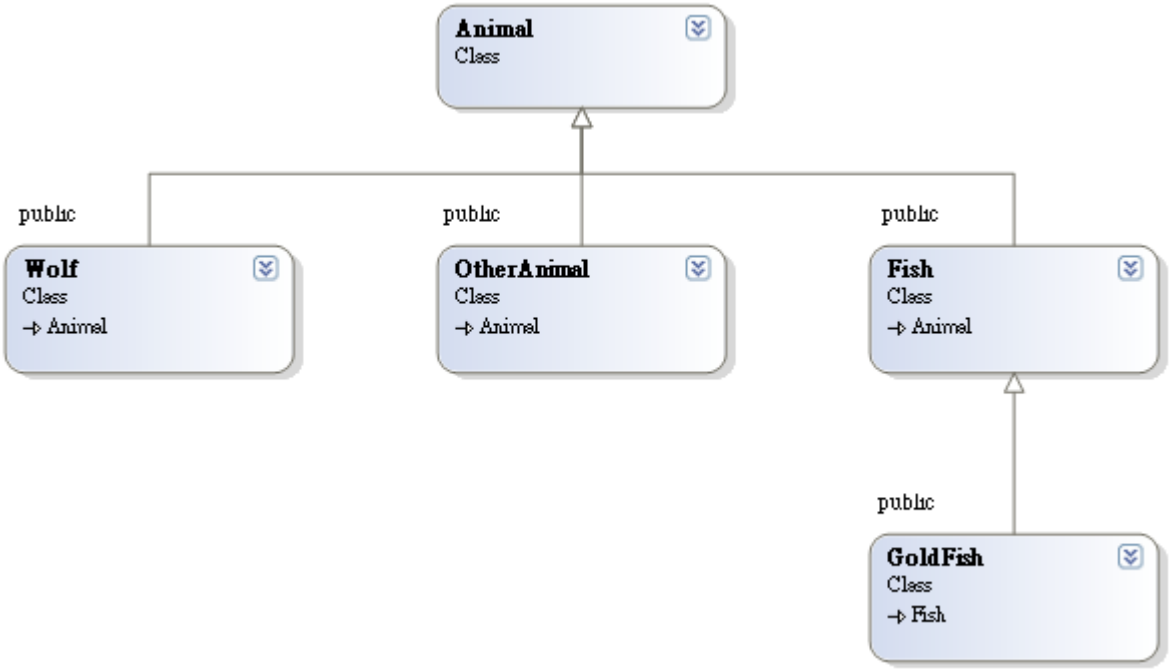
1.4. Điều khiển truy xuất

Khả năng hiện hữu của một lớp và các thành viên của nó có thể được hạn chế thông qua việc sử dụng các bổ sung truy cập: public, private, protected, internal, và protected internal.
Như chúng ta đã thấy, public cho phép một thành viên có thể được truy cập bởi một phương thức thành viên của những lớp khác. Trong khi đó private chỉ cho phép các phương thức thành viên trong lớp đó truy xuất. Từ khóa protected thì mở rộng thêm khả năng của private cho phép truy xuất từ các lớp dẫn xuất của lớp đó. Internal mở rộng khả năng cho phép bất cứ phương thức của lớp nào trong cùng một khối kết hợp (assembly) có thể truy xuất được. Một khối kết hợp được hiểu như là một khối chia xẻ và dùng lại trong CLR. Thông thường, khối này là tập hợp các tập tin vật lý được lưu trữ trong một thư mục bao gồm các tập tin tài nguyên, chương trình thực thi theo ngôn ngữ IL,...
Từ khóa internal protected đi cùng với nhau cho phép các thành viên của cùng một khối assembly hoặc các lớp dẫn xuất của nó có thể truy cập. Chúng ta có thể xem sự thiết kế này giống như là internal hay protected.

Các lớp cũng như những thành viên của lớp có thể được thiết kế với bất cứ mức độ truy xuất nào. Một lớp thường có mức độ truy xuất mở rộng hơn cách thành viên của lớp, còn các thành viên thì mức độ truy xuất thường có nhiều hạn chế. Do đó, ta có thể định nghĩa một lớp MyClass như sau:
public class MyClass { //... Protected int myValue; }
Như trên biến thành viên myValue được khai báo truy xuất protected mặc dù bản thân lớp được khai báo là public. Một lớp public là một lớp sẵn sàng cho bất cứ lớp nào khác muốn tương tác với nó. Đôi khi một lớp được tạo ra chỉ để trợ giúp cho những lớp khác trong một khối assemply, khi đó những lớp này nên được khai báo khóa internal hơn là khóa public.



2. Đa hình



Có hai cách thức khá mạnh để thực hiện việc kế thừa. Một là sử dụng lại mã nguồn, khi chúng ta tạo ra lớp ListBox, chúng ta có thể sử dụng lại một vài các thành phần trong lớp cơ sở như Window.

Tuy nhiên, cách sử dụng thứ hai chứng tỏ được sức mạnh to lớn của việc kế thừa đó là tính đa hình (polymorphism). Theo tiếng Anh từ này được kết hợp từ poly là nhiều và morph có nghĩa là form (hình thức). Do vậy, đa hình được hiểu như là khả năng sử dụng nhiều hình thức của một kiểu mà không cần phải quan tâm đến từng chi tiết.

Khi một tổng đài điện thoại gởi cho máy điện thoại của chúng ta một tín hiệu có cuộc gọi. Tổng đài không quan tâm đến điện thoại của ta là loại nào. Có thể ta đang dùng một điện thoại cũ dùng motor để rung chuông, hay là một điện thoại điện tử phát ra tiếng nhạc số. Hoàn toàn các thông tin về điện thoại của ta không có ý nghĩa gì với tổng đài, tổng đài chỉ biết một kiểu cơ bản là điện thoại mà thôi và diện thoại này sẽ biết cách báo chuông. Còn việc báo chuông như thế nào thì tổng đài không quan tâm. Tóm lại, tổng đài chỉ cần bảo điện thoại hãy làm điều gì đó để reng. Còn phần còn lại tức là cách thức reng là tùy thuộc vào từng loại điện thoại. Đây chính là tính đa hình.

2.1. Kiểu đa hình

Do một ListBox là một Window và một Button cũng là một Window, chúng ta mong muốn sử dụng cả hai kiểu dữ liệu này trong tình huống cả hai được gọi là Window. Ví dụ như trong một form giao diện trên MS Windows, form này chứa một tập các thể hiện của Window. Khi form được hiển thị, nó yêu cầu tất cả các thể hiện của Window tự thực hiện việc tô vẽ. Trong trường hợp này, form không muốn biết thành phần thể hiện là loại nào như Button, CheckBox,...,. Điều quan trọng là form kích hoạt toàn bộ tập hợp này tự thực hiện việc vẽ. Hay nói ngắn gọn là form muốn đối xử với những đối tượng Window này một cách đa hình.

2.2. Phương thức đa hình

Để tạo một phương thức hỗ tính đa hình, chúng ta cần phải khai báo khóa virtual trong phương thức của lớp cơ sở. Ví dụ, để chỉ định rằng phương thức DrawWindow() của lớp Window trong ví dụ 5.1 là đa hình, đơn giản là ta thêm từ khóa virtual vào khai báo như sau:

public virtual void DrawWindow()
Lúc này thì các lớp dẫn xuất được tự do thực thi các cách xử riêng của mình trong phiên bản mới của phương thức DrawWindow(). Để làm được điều này chỉ cần thêm từ khóa **override** để chồng lên phương thức ảo DrawWindow() của lớp cơ sở. Sau đó thêm các đoạn mã nguồn mới vào phương thức viết chồng này.
Trong ví dụ minh họa 5.2 sau, lớp ListBox dẫn xuất từ lớp Window và thực thi một phiên bản riêng của phương thức DrawWindow():
public override void DrawWindow() { base.DrawWindow(); Console.WriteLine("Writing string to the listbox: {0}", listBoxContents); }
Từ khóa **override** bảo với trình biên dịch rằng lớp này thực hiện việc phủ quyết lại phương thức DrawWindow() của lớp cơ sở. Tương tự như vậy ta có thể thực hiện việc phủ quyết phương thức này trong một lớp dẫn xuất khác như Button, lớp này cũng được dẫn xuất từ Window.
Trong phần thân của ví dụ 5.2, đầu tiên ta tạo ra ba đối tượng, đối tượng thứ nhất của Window, đối tượng thứ hai của lớp ListBox và đối tượng cuối cùng của lớp Button. Sau đó ta thực hiện việc gọi phương thức DrawWindow() cho mỗi đối tượng sau:
Window win = new Window(1, 2);
ListBox lb = new ListBox(3, 4, "Stand alone list box"); Button b = new Button(5, 6); win.DrawWindow(); lb.DrawWindow();
b.DrawWindow();
Đoạn chương trình trên thực hiện các công việc như yêu cầu của chúng ta, là từng đối tượng thực hiện công việc tô vẽ của nó. Tuy nhiên, cho đến lúc này thì chưa có bất cứ sự đa hình nào được thực thi. Mọi chuyện vẫn bình thường cho đến khi ta muốn tạo ra một mảng các đối tượng Window, bởi vì ListBox cũng là một Window nên ta có thể tự do đặt một đối tượng ListBox vào vị trí của một đối tượng Window trong mảng trên. Và tương tự ta cũng có thể đặt một đối tượng Button vào bất cứ vị trí nào trong mảng các đối tượng Window, vì một Button cũng là một Window.
Window[] winArray = new Window[3]; winArray[0] = new Window(1, 2); winArray[1] = new ListBox(3, 4, "List box is array"); winArray[2] = new Button(5, 6);
Chuyện gì xảy ra khi chúng ta gọi phương thức DrawWindow() cho từng đối tượng trong mảng winArray.
for(int i = 0; i < 3 ; i++) { winAray[i].DrawWindow(); }
Trình biên dịch điều biết rằng có ba đối tượng Windows trong mảng và phải thực hiện việc gọi phương thức DrawWindow() cho các đối tượng này. Nếu chúng ta không đánh dấu phương thức DrawWindow() trong lớp Window là virtual thì phương thức DrawWindow() trong lớp Window sẽ được gọi ba lần. Tuy nhiên do chúng ta đã đánh dấu phương thức này ảo ở lớp cơ sở và thực thi việc phủ quyết phương thức này ỏ các lớp dẫn xuất.
Khi ta gọi phương thức DrawWindow trong mảng, trình biên dịch sẽ dò ra được chính xác kiểu dữ liệu nào được thực thi trong mảng khi đó có ba kiểu sẽ được thực thi là một Window, một ListBox, và một Button. Và trình biên dịch sẽ gọi chính xác phương thức của từng đối tượng. Đây là điều cốt lõi và tinh hoa của tính chất đa hình. Đoạn chương trình hoàn chỉnh 5.2 minh họa cho sự thực thi tính chất đa hình.
Sử dụng phương thức ảo.
---
using System; public class Window { public Window(int top, int left) { this.top = top; this.left = left; } // phương thức được khai báo ảo public virtual void DrawWindow() {

```
Console.WriteLine( "Window: drawing window at {0}, {1}", top, left ); } // biến thành
viên của lớp protected int top; protected int left;
} public class ListBox : Window { // phương thức khởi dựng có tham số
public ListBox( int top, int left,string contents ): base( top, left) { listBoxContents =
contents; } //thực hiện việc phủ quyết phương thức DrwaWindow public override void
DrawWindow() { base.DrawWindow();
Console.WriteLine(" Writing string to the listbox: {0}", listBoxContents); }
// biến thành viên của ListBox private string
listBoxContents; } public class Button : Window { public Button( int top, int left) : base(
top, left )
{
} // phủ quyết phương thức DrawWindow của lớp cơ sở public override void
DrawWindow() {
Console.WriteLine(" Drawing a button at {0}: {1}", top, left); } } public class Tester {
static void Main() {
Window win = new Window(1,2); ListBox lb = new ListBox( 3, 4, " Stand alone list
box"); Button b = new Button( 5, 6 ); win.DrawWindow(); lb.DrawWindow();
b.DrawWindow(); Window[] winArray = new Window[3]; winArray[0] = new
Window( 1, 2 ); winArray[1] = new ListBox( 3, 4, "List box is array"); winArray[2] =
new Button( 5, 6 ); for( int i = 0; i < 3; i++)
{
winArray[i].DrawWindow(); }
}
}
}
```

Kết quả:

```
Window: drawing window at 1: 2
Window: drawing window at 3: 4
Writing string to the listbox: Stand alone list box
Drawing a button at 5: 6
Window: drawing Window at 1: 2
Window: drawing window at 3: 4
Writing string to the listbox: List box is array
Drawing a button at 5: 6
```

Trong suốt ví dụ này, chúng ta đánh dấu một phương thức phủ quyết mới với từ khóa phủ quyết **override :**

```
public override void DrawWindow()
```

Lúc này trình biên dịch biết cách sử dụng phương thức phủ quyết khi gặp đối tượng mang hình thức đa hình. Trình biên dịch chịu trách nhiệm trong việc phân ra kiểu dữ liệu thật của đối tượng để sau này xử lý. Do đó phương thức ListBox.DrawWindow() sẽ được gọi khi một đối tượng Window tham chiếu đến một đối tượng thật sự là ListBox.

Chúng ta phải chỉ định rõ ràng với từ khóa override khi khai báo một phương thức phủ quyết phương thức ảo của lớp cơ sở. Điều này dễ lầm lẫn với người lập trình C++ vì từ khóa này trong C++ có thể bỏ qua mà trình biên dịch C++ vẫn hiểu.

Từ khóa new và override

Trong ngôn ngữ C#, người lập trình có thể quyết định phủ quyết một phương thức ảo bằng cách khai báo tường minh từ khóa override. Điều này giúp cho ta đưa ra một phiên bản mới của chương trình và sự thay đổi của lớp cơ sở sẽ không làm ảnh hưởng đến chương trình viết trong các lớp dẫn xuất. Việc yêu cầu sử dụng từ khóa override sẽ giúp ta ngăn ngừa vấn đề này.

Bây giờ ta thử bàn về vấn đề này, giả sử lớp cơ sở Window của ví dụ trước được viết bởi một công ty A. Cũng giả sử rằng lớp ListBox và RadioButton đươc viết từ những người lập trình của công ty B và họ dùng lớp cơ sở Window mua được của công ty A làm lớp cơ sở cho hai lớp trên. Người lập trình trong công ty B không có hoặc có rất ít sự kiểm soát về những thay đổi trong tương lai với lớp Window do công ty A phát triển. Khi nhóm lập trình của công ty B quyết định thêm một phương thức Sort( ) vào lớp ListBox:

public class ListBox : Window { public virtual void Sort( ) {...} }

Việc thêm vào vẫn bình thường cho đến khi công ty A, tác giả của lớp cơ sở Window, đưa ra phiên bản thứ hai của lớp Window. Và trong phiên bản mới này những người lập trình của công ty A đã thêm một phương thức Sort( ) vào lớp cơ sở Window:

public class Window { //... public virtual void Sort( ) {...} }

Trong các ngôn ngữ lập trình hướng đối tượng khác như C++, phương thức ảo mới Sort() trong lớp Window bây giờ sẽ hành động giống như là một phương thức cơ sở cho phương thức ảo trong lớp ListBox. Trình biên dịch có thể gọi phương thức Sort( ) trong lớp ListBox khi chúng ta có ý định gọi phương thức Sort( ) trong Window. Trong ngôn ngữ Java, nếu phương thức Sort( ) trong Window có kiểu trả về khác kiểu trả về của phương thức Sort( ) trong lớp ListBox thì sẽ được báo lỗi là phương thức phủ quyết không hợp lệ.

Ngôn ngữ C# ngăn ngừa sự lẫn lộn này, trong C# một phương thức ảo thì được xem như là gốc rễ của sự phân phối ảo. Do vậy, một khi C# tìm thấy một phương thức khai báo là ảo thì nó sẽ không thực hiện bất cứ việc tìm kiếm nào trên cây phân cấp kế thừa. Nếu một phương thức ảo Sort( ) được trình bày trong lớp Window, thì khi thực hiện hành vi của lớp Listbox không thay đổi.

Tuy nhiên khi biên dịch lại, thì trình biên dịch sẽ đưa ra một cảnh báo giống như sau:

…\class1.cs(54, 24): warning CS0114: ‘ListBox.Sort( )’ hides inherited member ‘Window.Sort()’.

To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

Để loại bỏ cảnh báo này, người lập trình phải chỉ rõ ý định của anh ta. Anh ta có thể đánh dấu phương thức ListBox.Sort( ) với từ khóa là new, và nó không phải phủ quyết của bất cứ phương thức ảo nào trong lớp Window:

public class ListBox : Window { public new virtual Sort( ) {...} }

Việc thực hiện khai báo trên sẽ loại bỏ được cảnh báo. Mặc khác nếu người lập trình muốn phủ quyết một phương thức trong Window, thì anh ta cần thiết phải dùng từ khóa **override** để khai báo một cách tường minh:

public class ListBox : Window { public override void Sort(){...} }

3. Lớp trừu tượng

	Dữ liệu	Hành động	
Con chó	Tên Màu Giống	Sủa Vẫy tai Chạy Ăn	
Xe đạp	Bánh răng Bàn đạp Dây xích Bánh xe	Tăng tốc Giảm tốc Chuyển bánh răng ...	

Mỗi lớp con của lớp Window nên thực thi một phương thức DrawWindow() cho riêng mình. Tuy nhiên điều này không thực sự đòi hỏi phải thực hiện một cách bắt buộc. Để yêu cầu các lớp con (lớp dẫn xuất) phải thực thi một phương thức của lớp cơ sở, chúng ta phải thiết kế một phương thức một cách trừu tượng.

Một phương thức trừu tượng không có sự thực thi. Phương thức này chỉ đơn giản tạo ra một tên phương thức và ký hiệu của phương thức, phương thức này sẽ được thực thi ở các lớp dẫn xuất.

Những lớp trừu tượng được thiết lập như là cơ sở cho những lớp dẫn xuất, nhưng việc tạo các thể hiện hay các đối tượng cho các lớp trừu tượng được xem là không hợp lệ. Một khi chúng ta khai báo một phương thức là trừu tượng, thì chúng ta phải ngăn cấm bất cứ việc tạo thể hiện cho lớp này.

Do vậy, nếu chúng ta thiết kế phương thức DrawWindow() như là trừu tượng trong lớp Window, chúng ta có thể dẫn xuất từ lớp này, nhưng ta không thể tạo bất cứ đối tượng cho lớp này. Khi đó mỗi lớp dẫn xuất phải thực thi phương thức DrawWindow(). Nếu lớp dẫn xuất không thực thi phương thức trừu tượng của lớp cơ sở thì lớp dẫn xuất đó cũng là lớp trừu tượng, và ta cũng không thể tạo các thể hiện của lớp này được.

Phương thức trừu tượng được thiết lập bằng cách thêm từ khóa **abstract** vào đầu của phần định nghĩa phương thức, cú pháp thực hiện như sau:

abstract public void DrawWindow( );

Do phương thức không cần phần thực thi, nên không có dấu ({}) mà chỉ có dấu chấm phẩy (;) sau phương thức. Như thế với phương thức DrawWindow() được thiết kế là trừu tượng thì chỉ cần câu lệnh trên là đủ.

Nếu một hay nhiều phương thức được khai báo là trừu tượng, thì phần định nghĩa lớp phải được khai báo là abstract, với lớp Window ta có thể khai báo là lớp trừu tượng như sau:

abstract public void Window

Ví dụ sau minh họa việc tạo lớp Window trừu tượng và phương thức trừu tượng DrawWindow() của lớp Window.

Sử dụng phương thức và lớp trừu tượng.

```
using System; abstract public class Window { // hàm khởi dựng lấy hai tham số public Window( int top, int left) { this.top = top; this.left = left; } // phương thức trừu tượng minh họa việc // vẽ ra cửa sổ abstract public void DrawWindow(); // biến thành viên protected protected int top; protected int left; } // lớp ListBox dẫn xuất từ lớp Window public class ListBox : Window { // hàm khởi dựng lấy ba tham số public ListBox( int top, int left, string contents) : base( top, left) { listBoxContents = contents; } // phủ quyết phương thức trừu tượng DrawWindow() public override void DrawWindow( ) { Console.WriteLine("Writing string to the listbox: {0}", listBoxContents); } // biến private của lớp private string listBoxContents; } // lớp Button dẫn xuất từ lớp Window public class Button : Window { // hàm khởi tạo nhận hai tham số public Button( int top, int left) : base( top, left) { } // thực thi phương thức trừu tượng public override void DrawWindow() {
Console.WriteLine("Drawing button at {0}, {1}\n", top, left); } } public class Tester { static void Main() { Window[] winArray = new Window[3]; winArray[0] = new
ListBox( 1, 2, "First List Box"); winArray[1] = new
ListBox( 3, 4,"Second List Box"); winArray[2] = new
Button( 5, 6); for( int i=0; i <3 ; i++) {
winArray[i].DrawWindow( ); } } } ----------
```

Trong ví dụ trên, lớp Window được khai báo là lớp trừu tượng và do vậy nên chúng ta không thể tạo bất cứ thể hiện nào của lớp Window. Nếu chúng ta thay thế thành viên đầu tiên của mảng:

```
winArray[0] = new ListBox( 1, 2, "First List Box");
```

bằng câu lệnh sau:

```
winArray[0] = new Window( 1, 2);
```

Thì trình biên dịch sẽ báo một lỗi như sau:

Cannot create an instance of the abstract class or interface ‘Window’

Chúng ta có thể tạo được các thể hiện của lớp ListBox và Button, bởi vì hai lớp này đã phủ quyết phương thức trừu tượng. Hay có thể nói hai lớp này đã được xác định (ngược với lớp trừu tượng).

Hạn chế của lớp trừu tượng

Mặc dù chúng ta đã thiết kế phương thức DrawWindow() như một lớp trừu tượng để hỗ trợ cho tất cả các lớp dẫn xuất được thực thi riêng, nhưng điều này có một số hạn chế. Nếu chúng ta dẫn xuất một lớp từ lớp ListBox như lớp DropDownListBox, thì lớp này không được hỗ trợ để thực thi phương thức DrawWindow( ) cho riêng nó.

Khác với ngôn ngữ C++, trong C# phương thức Window.DrawWindow( ) không thể cung cấp một sự thực thi, do đó chúng ta sẽ không thể lấy được lợi ích của phương thức DrawWindow() bình thường dùng để chia xẻ bởi các lớp dẫn xuất.

Cuối cùng những lớp trừu tượng không có sự thực thi căn bản; chúng thể hiện ý tưởng về một sự trừu tượng, điều này thiết lập một sự giao ước cho tất cả các lớp dẫn xuất. Nói cách khác các lớp trừu tượng mô tả một phương thức chung của tất cả các lớp được thực thi một cách trừu tượng.

Ý tưởng của lớp trừu tượng Window thể hiện những thuộc tính chung cùng với những hành vi của tất cả các Window, thậm chí ngay cả khi ta không có ý định tạo thể hiện của chính lớp trừu tượng Window.

Ý nghĩa của một lớp trừu tượng được bao hàm trong chính từ "trừu tượng". Lớp này dùng để thực thi một "Window" trừu tượng, và nó sẽ được biểu lộ trong các thể hiện xác định của Windows, như là Button, ListBox, Frame,...
Các lớp trừu tượng không thể thực thi được, chỉ có những lớp xác thực tức là những lớp dẫn xuất từ lớp trừu tượng này mới có thể thực thi hay tạo thể hiện. Một sự thay đổi việc sử dụng trừu tượng là định nghĩa một giao diện (interface), phần này sẽ được trình bày trong Chương 8 nói về giao diện.

Lớp cô lập (sealed class)

Ngược với các lớp trừu tượng là các lớp cô lập. Một lớp trừu tượng được thiết kế cho các lớp dẫn xuất và cung cấp các khuôn mẫu cho các lớp con theo sau. Trong khi một lớp cô lập thì không cho phép các lớp dẫn xuất từ nó. Để khai báo một lớp cô lập ta dùng từ khóa sealed đặt trước khai báo của lớp không cho phép dẫn xuất. Hầu hết các lớp thường được đánh dấu sealed nhằm ngăn chặn các tai nạn do sự kế thừa gây ra.
Nếu khai báo của lớp Window trong ví dụ 5.3 được thay đổi từ khóa abstract bằng từ khóa sealed (cũng có thể loại bỏ từ khóa trong khai báo của phương thức DrawWindow()). Chương trình sẽ bị lỗi khi biên dịch. Nếu chúng ta cố thử biên dịch chương trình thì sẽ nhận được lỗi từ trình biên dịch:
'ListBox' cannot inherit from sealed class 'Window'
Đây chỉ là một lỗi trong số những lỗi như ta không thể tạo một phương thức thành viên protected trong một lớp khai báo là sealed.

Gốc của tất cả các lớp: Lớp Object

Tất cả các lớp của ngôn ngữ C# của bất cứ kiểu dữ liệu nào thì cũng được dẫn xuất từ lớp System.Object. Thú vị là bao gồm cả các kiểu dữ liệu giá trị.
Một lớp cơ sở là cha trực tiếp của một lớp dẫn xuất. Lớp dẫn xuất này cũng có thể làm cơ sở cho các lớp dẫn xuất xa hơn nữa, việc dẫn xuất này sẽ tạo ra một cây thừa kế hay một kiến trúc phân cấp. Lớp gốc là lớp nằm ở trên cùng cây phân cấp thừa kế, còn các lớp dẫn xuất thì nằm bên dưới. Trong ngôn ngữ C#, lớp gốc là lớp Object, lớp này nằm trên cùng trong cây phân cấp các lớp.
Lớp Object cung cấp một số các phương thức dùng cho các lớp dẫn xuất có thể thực hiện việc phủ quyết. Những phương thức này bao gồm Equals() kiểm tra xem hai đối tượng có giống nhau hay không. Phương thức GetType() trả về kiểu của đối tượng. Và phương thức ToString() trả về một chuỗi thể hiện lớp hiện hành. Sau đây là bảng tóm tắt các phương thức của lớp Object.

Tóm tắt các phương thức của lớp Object

Phương thức	Chức năng
Equal( )	So sánh bằng nhau giữa hai đối tượng
GetHashCode( )	Cho phép những đối tượng cung cấp riêngnhững hàm băm cho sử dụng tập hợp.
GetType( )	Cung cấp kiểu của đối tượng
ToString( )	Cung cấp chuỗi thể hiện của đối tượng
Finalize( )	Dọn dẹp các tài nguyên

MemberwiseClone( )	Tạo một bản sao từ đối tượng.

Ví dụ sau minh họa việc sử dụng phương thức ToString( ) thừa kế từ lớp Object.
Thừa kế từ Object.

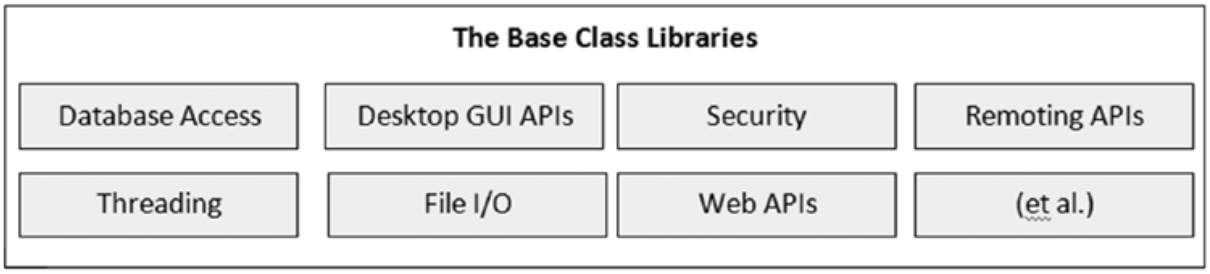
---

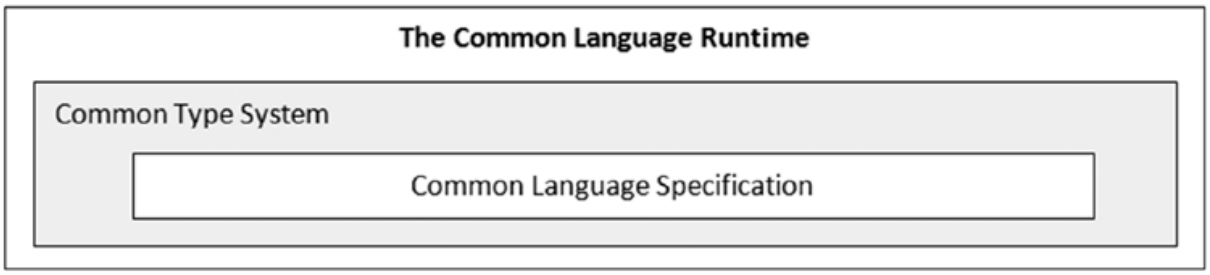
using System; public class SomeClass { public SomeClass(int val) { value = val; } // phủ quyết phương thức
ToString của lớp Object public virtual string ToString() { return value.ToString(); } // biến thành viên private lưu giá trị private int value; } public class Tester { static void Main( ) { int i = 5; Console.WriteLine("The value of i is: {0}", i.ToString()); SomeClass s = new SomeClass(7); Console.WriteLine("The value of s is {0}", s.ToString()); Console.WriteLine("The value of 5 is {0}",5.ToString()); }
}

Kết quả:
The value of i is: 5
The value of s is 7
The value of 5 is 5

Trong tài liệu của lớp Object phương thức ToString() được khai báo như sau:
public virtual string ToString();
Đây là phương thức ảo public, phương thức này trả về một chuỗi và không nhận tham số. Tất cả kiểu dữ liệu được xây dựng sẵn, như kiểu int, dẫn xuất từ lớp Object nên nó cũng có thể thực thi các phương thức của lớp Object.
Lớp SomeClass trong ví dụ trên thực hiện việc phủ quyết phương thức ToString(), do đó phương thức này sẽ trả về giá trị có nghĩa. Nếu chúng ta không phủ quyết phương thức ToString() trong lớp SomeClass, phương thức của lớp cơ sở sẽ được thực thi, và kết quả xuất ra sẽ có thay đổi như sau:
The value of s is SomeClass
Như chúng ta thấy, hành vi mặc định đã trả về một chuỗi chính là tên của lớp đang thể hiện.
Các lớp không cần phải khai báo tường minh việc dẫn xuất từ lớp Object, việc kế thừa sẽ được đưa vào một cách ngầm định. Như lớp SomeClass trên ta không khai báo bất cứ dẫn xuất của lớp nào nhưng C# sẽ tự động đưa lớp Object thành lớp dẫn xuất. Do đó ta mới có thể phủ quyết phương thức ToString() của lớp Object.

4. Các lớp lồng nhau





Các lớp chứa những thành viên, và những thành viên này có thể là một lớp khác có kiểu do người dùng định nghĩa (user-defined type). Do vậy, một lớp Button có thể có một thành viên của kiểu Location, và kiểu Location này chứa thành viên của kiểu dữ liệu Point. Cuối cùng, Point có thể chứa chứa thành viên của kiểu int.

Cho đến lúc này, các lớp được tạo ra chỉ để dùng cho các lớp bên ngoài, và chức năng của các lớp đó như là lớp trợ giúp (helper class). Chúng ta có thể định nghĩa một lớp trợ giúp bên trong các lớp ngoài (outer class). Các lớp được định nghĩa bên trong gọi là các lớp lồng (nested class), và lớp chứa được gọi đơn giản là lớp ngoài.

Những lớp lồng bên trong có lợi là có khả năng truy cập đến tất cả các thành viên của lớp ngoài. Một phương thức của lớp lồng có thể truy cập đến biến thành viên private của lớp ngoài.

Hơn nữa, lớp lồng bên trong có thể ẩn đối với tất cả các lớp khác, lớp lồng có thể là private cho lớp ngoài.

Cuối cùng, một lớp làm lồng bên trong là public và được truy cập bên trong phạm vi của lớp ngoài. Nếu một lớp Outer là lớp ngoài, và lớp Nested là lớp public lồng bên trong lớp Outer, chúng ta có thể tham chiếu đến lớp Tested như Outer.Nested, khi đó lớp bên ngoài hành động ít nhiều giống như một namespace hay một phạm vi.

Đối với người lập trình Java, lớp lồng nhau trong C# thì giống như những lớp nội static (static inner) trong Java. Không có sự tương ứng trong C# với những lớp nội nonstatic (nonstatic inner) trong Java.

Ví dụ sau sẽ thêm một lớp lồng vào lớp Fraction tên là FractionArtist. Chức năng của lớp FractionArtis là vẽ một phân số ra màn hình. Trong ví dụ này, việc vẽ sẽ được thay thế bằng sử dụng hàm WriteLine xuất ra màn hình console.

Sử dụng lớp lồng nhau.

```
using System; using System.Text; public class Fraction { public Fraction( int numerator, int denominator) { this.numerator = numerator; this.denominator = denominator; } public override string ToString() { StringBuilder s = new StringBuilder(); s.AppendFormat("{0}/{1}",numerator, denominator); return s.ToString(); } internal class FractionArtist { public void Draw( Fraction f) { Console.WriteLine("Drawing the numerator {0}",f.numerator); Console.WriteLine("Drawing the denominator {0}",
```

f.denominator); }
} // biến thành viên private private int numerator; private int denominator; } public class Tester { static void Main() {
Fraction f1 = new Fraction(3, 4);
Console.WriteLine("f1: {0}", f1.ToString());
Fraction.FractionArtist fa = new Fraction.FractionArtist(); fa.Draw(f1); }
}

Lớp Fraction trên nói chung là không có gì thay đổi ngoại trừ việc thêm một lớp lồng bên trong và lược đi một số phương thức không thích hợp trong ví dụ này. Lớp lồng bên trong FractionArtist chỉ cung cấp một phương thức thành viên duy nhất, phương thức Draw(). Điều thú vị trong phương thức Draw() truy cập dữ liệu thành viên private là f.numerator và f.denominator. Hai viến thành viên private này sẽ không cho phép truy cập nếu FractionArtist không phải là lớp lồng bên trong của lớp Fraction.
Trong hàm Main() khi khai báo một thể hiện của lớp lồng bên trong, chúng ta phải xác nhận tên của lớp bên ngoài, tức là lớp Fraction:
Fraction.FractionArtist fa = new Fraction.FractionArtist();
Thậm chí khi lớp FractionArtist là public, thì phạm vị của lớp này vẫn nằm bên trong của lớp Fraction.

Bài tập:

Tạo lớp "Vật nuôi", khai báo các thông tin cần thiết cho vật nuôi. Tạo thêm một lớp "Con gà" là lớp kế thừa của lớp "Vật nuôi"

Bài tập nâng cao:

Trong lớp "Con gà" tạo các hành động mang tính đa hình là "Cho ăn" (Mỗi giống gà sẽ có lượng thức ăn khác nhau, loại thức ăn khác nhau).

Những trọng tâm cần chú ý trong bài:

- Biết các kiến thức về tính kế thừa và đa hình trên C#;
- Biết các kiến thức về lớp trừu tượng;
- Hiểu được các kiến thức và kỹ năng về các phương thức, các thành phần static;
- Hiểu được các kiến thức và kỹ năng về tham số và các phương thức nạp chồng;
- Nghiêm túc, sáng tạo trong quá trình tiếp thu lý thuyết và áp dụng làm các bài tập.

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:

- Biết các kiến thức về tính kế thừa và đa hình trên C#;
- Biết các kiến thức về lớp trừu tượng;
- Hiểu được các kiến thức và kỹ năng về các phương thức, các thành phần static;
- Hiểu được các kiến thức và kỹ năng về tham số và các phương thức nạp chồng;

+ Về kỹ năng: Xây dựng được các lớp kế thừa từ lớp cha, tạo các hành động mang tính đa hình.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Xây dựng được các lớp kế thừa từ lớp cha, tạo các hành động mang tính đa hình

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 4: NẠP CHỒNG TOÁN TỬ
Mã bài: MĐ 11 - 05

Giới thiệu:
Một tính năng rất hữu ích trong ngôn ngữ lập trình hướng đối tượng là nạp chồng toán tử.

Mục tiêu:
+ Biết được các kiến thức về toán tử;
+ Biết các kiến thức về sự hỗ trợ nạp chồng toán tử trong các ngôn ngữ .Net khác;
+ Trang bị các kiến thức và kỹ năng về nạp chồng các toán tử trong C#;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Nội dung chính:

1. Sử dụng từ khóa operator

Operators	Overload ability
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using `&` and `|`, which can be overloaded

Trong ngôn ngữ C#, các toán tử là các phương thức tĩnh, giá trị trả về của nó thể hiện kết quả của một toán tử và những tham số là các toán hạng. Khi chúng ta tạo một toán tử cho một lớp là chúng ta đã thực việc nạp chồng (overloaded) những toán tử đó, cũng giống như là chúng ta có thể nạp chồng bất cứ phương thức thành viên nào. Do đó, để nạp chồng toán tử cộng (+) chúng ta có thể viết như sau:
public static Fraction operator + (Fraction lhs, Fraction rhs)
Trong toán tử trên ta có sự qui ước đặt tên của tham số là lhs và rhs. Tham số tên lhs thay thế cho "left hand side" tức là toán hạng bên trái, tương tự tham số tên rhs thay thế cho "right hand side" tức là toán hạng bên phải.
Cú pháp ngôn ngữ C# cho phép nạp chồng một toán tử bằng cách viết từ khóa **operator** và theo sau là toán tử được nạp chồng. Từ khóa **operator** là một bổ sung phương thức (method operator). Như vậy, để nạp chồng toán tử cộng (+) chúng ta có thể viết **operator** +. Khi chúng ta viết:
Fraction theSum = firstFraction + secondFraction;
Thì toán tử nạp chồng + được thực hiện, với firstFraction được truyền vào như là tham số đầu tiên, và secondFraction được truyền vào như là tham số thứ hai. Khi trình biên dịch gặp biểu thức:
firstFraction + secondFraction thì trình biên dịch sẽ chuyển biểu thức vào:
Fraction.operator+(firstFraction, secondFraction)
Kết quả sau khi thực hiện là một đối tượng Fraction mới được trả về, trong trường hợp này phép gán sẽ được thực hiện để gán một đối tượng Fraction cho theSum.
Đối với người lập trình C++, trong ngôn ngữ C# không thể tạo được toán tử nonstatic, và do vậy nên toán tử nhị phân phải lấy hai toán hạng.

2. Hỗ trợ ngôn ngữ .NET khác

Ngôn ngữ C# cung cấp khả năng cho phép nạp chồng toán tử cho các lớp mà chúng ta xây dựng, thậm chí điều này không hoặc đề cập rất ít trong Common Language Specification (CLS). Những ngôn ngữ .NET khác như VB.NET thì không hỗ trợ việc nạp chồng toán tử, và một điều quan trọng để đảm bảo là lớp của chúng ta phải hỗ trợ các phương thức thay thế cho phép những ngôn ngữ khác có thể gọi để tạo ra các hiệu ứng tương tự.

Do đó, nếu chúng ta nạp chồng toán tử (+) thì chúng ta nên cung cấp một phương thức Add() cũng làm cùng chức năng là cộng hai đối tượng. Nạp chồng toán tử có thể là một cú pháp ngắn gọn, nhưng nó không chỉ là đường dẫn cho những đối tượng của chúng ta thiết lập một nhiệm vụ được đưa ra.

3. Sử dụng toán tử

Nạp chồng toán tử có thể làm cho mã nguồn của chúng ta trực quan và những hành động của lớp mà chúng ta xây dựng giống như các lớp được xây dựng sẵn. Tuy nhiên, việc nạp chồng toán tử cũng có thể làm cho mã nguồn phức tạp một cách khó quản lý nếu chúng ta phá vỡ cách thể hiện thông thường để sử dụng những toán tử. Hạn chế việc sử dụng tùy tiện các nạp chồng toán tử bằng những cách sử dụng mới và những cách đặc trưng.

Mặc dù chúng ta có thể hấp dẫn bởi việc sử dụng nạp chồng toán tử gia tăng (++) trong lớp Employee để gọi một phương thức gia tăng mức lương của nhân viên, điều này có thể đem lại rất nhiều nhầm lẫn cho các lớp client truy cập lớp Employee. Vì bên trong của lớp còn có thể có nhiều trường thuộc tính số khác, như số tuổi, năm làm việc,...ta không thể dành toán tử gia tăng duy nhất cho thụôc tính lương được. Cách tốt nhất là sử dụng nạp chồng toán tử một cách hạn chế, và chỉ sử dụng khi nào nghĩa nó rõ ràng và phù hợp với các toán tử của các lớp được xây dựng sẵn.

Khi thường thực hiện việc nạp chồng toán tử so sánh bằng (==) để kiểm tra hai đối tượng xem có bằng nhau hay không. Ngôn ngữ C# nhấn mạnh rằng nếu chúng ta thực hiện nạp chồng toán tử bằng, thì chúng ta phải nạp chồng toán tử nghịch với toán tử bằng là toán tử không bằng (!=). Tương tự, khi nạp chồng toán tử nhỏ hơn (<) thì cũng phải tạo toán tử (>) theo từng cặp. Cũng như toán tử (>=) đi tương ứng với toán tử (<=).

Theo sau là một số luật được áp dụng để thực hiện nạp chồng toán tử:

Định nghĩa những toán tử trong kiểu dữ liệu giá trị, kiểu do ngôn ngữ xây dựng sẵn.

Cung cấp những phương thức nạp chồng toán tử chỉ bên trong của lớp nơi mà những phương thức được định nghĩa.

Sử dụng tên và những kí hiệu qui ước được mô tả trong Common Language Specification (CLS).

Sử dụng nạp chồng toán tử trong trường hợp kết quả trả về của toán tử là thật sự rõ ràng. Thực hiện toán tử trừ (-) giữa một giá trị Time với một giá trị Time khác là một toán tử có ý nghĩa. Tuy nhiên, nếu chúng ta thực hiện toán tử or hay toán tử and giữa hai đối tượng Time thì kết quả hoàn toàn không có nghĩa gì hết.

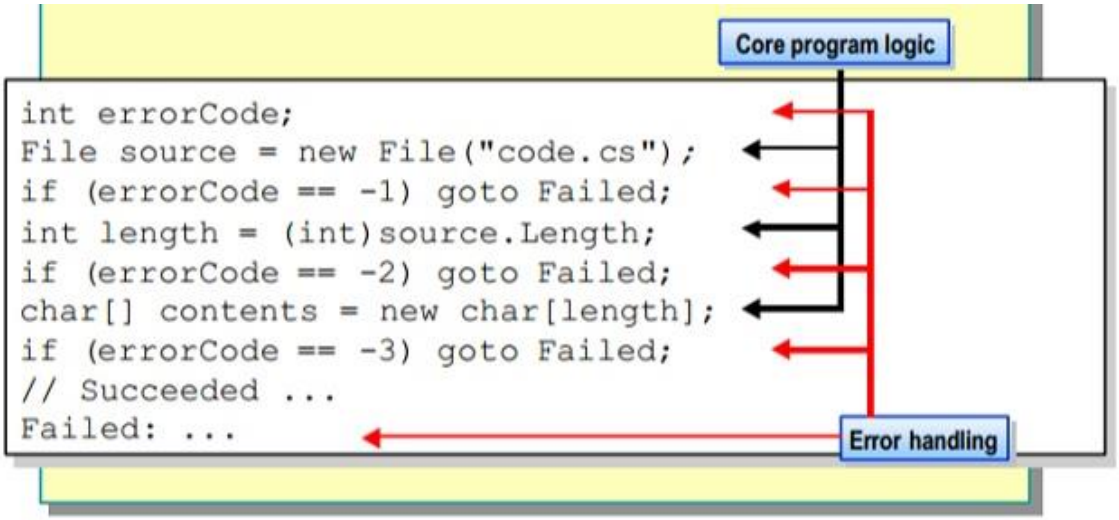
Nạp chồng toán tử có tính chất đối xứng. Ví dụ, nếu chúng ta nạp chồng toán tử bằng (==) thì cũng phải nạp chồng toán tử không bằng (!=). Do đó khi thực hiện toán tử có tính chất đối xứng thì phải thực hiện toán tử đối xứng lại như: < với >, <= với >=.

Phải cung cấp các phương thức thay thế cho toán tử được nạp chồng. Đa số các ngôn ngữ điều không hỗ trợ nạp chồng toán tử. Vì nguyên do này nên chúng ta phải thực thi các phương thức thứ hai có cùng chức năng với các toán tử. Common Language Specification (CLS) đòi hỏi phải thực hiện phương thức thứ hai tương ứng.

Bảng sau trình bày các toán tử cùng với biểu tượng của toán tử và các tên của phương thức thay thế các toán tử.

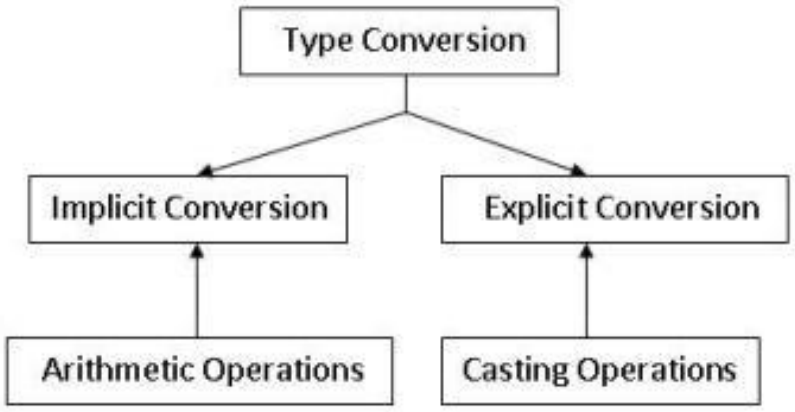
Biểu tượng	Tên phương thức thay thế	Tên toán tử
+	Add	Toán tử cộng
-	Subtract	Toán tử trừ
*	Multiply	Toán tử nhân
/	Divide	Toán tử chia
%	Mod	Toán tử chia lấy dư
^	Xor	Toán tử or loại trừ
&	BitwiseAnd	Toán tử and nhị phân
|	BitwiseOr	Toán tử or nhị phân
&&	And	Toán tử and logic
||	Or	Toán tử or logic
=	Assign	Toán tử gán
<<	LeftShift	Toán tử dịch trái
>>	RightShift	Toán tử dịch phải
==	Equals	Toán tử so sánh bằng
>	Compare	Toán tử so sánh lớn hơn
<	Compare	Toán tử so sánh nhỏ hơn
!=	Compare	Toán tử so sánh không bằng
>=	Compare	Toán tử so sánh lớn hơn haybằng
<=	Compare	Toán tử so sánh nhỏ hơn haybằng
*=	Multiply	Toán tử nhân rồi gán trở lại
-=	Subtract	Toán tử trừ rồi gán trở lại
^=	Xor	Toán tử or loại trừ rồi gán lại
<<=	LeftShift	Toán tử dịch trái rồi gán lại
%=	Mod	Toán tử chia dư rồi gán lại
+=	Add	Toán tử cộng rồi gán lại
&=	BitwiseAnd	Toán tử and rồi gán lại
|=	BitwiseOr	Toán tử or rồi gán lại

4. Toán tử so sánh bằng



Nếu chúng ta nạp chồng toán tử bằng (==), thì chúng ta cũng nên phủ quyết phương thức ảo Equals() được cung cấp bởi lớp object và chuyển lại cho toán tử bằng thực hiện. Điều này cho phép lớp của chúng ta thể tương thích với các ngôn ngữ .NET khác không hỗ trợ tính nạp chồng toán tử nhưng hỗ trợ nạp chồng phương thức. Những lớp FCL không sử dụng nạp chồng toán tử, nhưng vẫn mong đợi lớp của chúng ta thực hiện những phương thức cơ bản này. Do đó ví dụ lớp ArrayList mong muốn chúng ta thực thi phương thức
Equals().
Lớp object thực thi phương thức Equals() với khai báo sau:
public override bool Equals(object 0)
Bằng cách phủ quyết phương thức này, chúng ta cho phép lớp Fraction hành động một cách đa hình với tất cả những lớp khác. Bên trong thân của phương thức Equals() chúng ta cần phải đảm bảo rằng chúng ta đang so sánh với một Fraction khác, và nếu như chúng ta đã thực thi một toán tử so sánh bằng thì có thể định nghĩa phương thức Equals() như sau:
pubic override bool Equals(object o) { if (!(o is
Fraction)) { return false; } return this == (Fraction) o;
}
Toán tử is được sử dụng để kiểm tra kiểu của đối tượng lúc chạy chương trình có tương thích với toán hạng trong trường hợp này là Fraction. Do o là Fraction nên toán tử is sẽ trả về true.

5. Toán tử chuyển đổi



C# cho phép chuyển đổi từ kiểu int sang kiểu long một cách ngầm định, và cũng cho phép chúng ta chuyển từ kiểu long sang kiểu int một cách tường minh. Việc chuyển từ kiểu int sang kiểu long được thực hiện ngầm định bởi vì hiển nhiên bất kỳ giá trị nào của int cũng được thích hợp với kích thước của kiểu long. Tuy nhiên, điều ngược lại, tức là chuyển từ kiểu long sang kiểu int phải được thực hiện một cách tường minh (sử dụng ép kiểu) bởi vì ta có thể mất thông tin khi giá trị của biến kiểu long vượt quá kích thước của int lưu trong bộ nhớ:
int myInt = 5; long myLong; myLong = myInt; // ngầm định myInt = (int) myLong; // tường minh
Chúng ta muốn thực hiện việc chuyển đổi này với lớp Fraction. Khi đưa ra một số nguyên, chúng ta có thể hỗ trợ ngầm định để chuyển đổi thành một phân số bởi vì bất kỳ giá trị nguyên nào ta cũng có thể chuyển thành giá trị phân số với mẫu số là 1 như (24
== 24/1).
Khi đưa ra một phân số, chúng ta muốn cung cấp một sự chuyển đổi tường minh trở lại một số nguyên, điều này có thể hiểu là một số thông tin sẽ bị mất. Do đó, khi chúng ta chuyển phân số 9/4 thành giá trị nguyên là 2.
Từ ngữ ngầm định (implicit) được sử dụng khi một chuyển đổi đảm thành công mà không mất bất cứ thông tin nào của dữ liệu nguyên thủy. Trường hợp ngược lại, tường minh (explicit) không đảm bảo bảo toàn dữ liệu sau khi chuyển đổi do đó việc này sẽ được thực hiện một cách công khai.
Ví dụ sau sẽ trình bày dưới đây minh họa cách thức mà chúng ta có thể thực thi chuyển đổi tường minh và ngầm định, và thực thi một vài các toán tử của lớp Fraction. Trong ví dụ này chúng ta sử dụng hàm Console.WriteLine() để xuất thông điệp ra màn hình minh họa khi phương thức được thi hành. Tuy nhiên cách tốt nhất là chúng ta sử dụng trình bebug để theo dõi từng bước thực thi các lệnh hay nhảy vào từng phương thức được gọi.
Định nghĩa các chuyển đổi và toán tử cho lớp Fraction.
---
using System; public class Fraction
{
public Fraction(int numerator,int denominator) { Console.WriteLine("In Fraction Constructor(int, int) "); this.numerator = numerator; this.denominator = denominator; } public Fraction(int wholeNumber) { Console.WriLine("In Fraction Constructor(int)"); numerator = wholeNumber; denominator = 1; } public static implicit operator Fraction(int theInt) { Console.WriteLine(" In implicit conversion to Fraction"); return new Fraction(theInt); } public static explicit operator int(Fraction theFraction)
{ Console.WriteLine("In explicit conversion to int"); return theFraction.numerator / theFraction.denominator;
}
public static bool operator == (Fraction lhs, Fraction rhs) { Console.WriteLine("In operator =="); if (lhs.numerator == rhs.numerator && lhs.denominator == rhs.denominator) { return true; } // thực hiện khi hai phân số không bằng nhau return false; }
public static bool operator != (Fraction lhs, Fraction rhs) { Console.WriteLine("In operator !="); return !(lhs == rhs); } public override bool Equals(object o) { Console.WriteLine("In method Equals"); if (!(o is Fraction))
{

```
return false; } return this == ( Fraction ) o; }
public static Fraction operator+( Fraction lhs, Fraction rhs ) { Console.WriteLine("In operator +"); if (lhs.denominator == rhs.denominator ) {
return new Fraction( lhs.numerator + rhs.numerator,
lhs.denominator ); } //thực hiện khi hai mẫu số không bằng nhau int firstProduct = lhs.numerator * rhs.denominator; int secondProduct = rhs.numerator * lhs.denominator;
return new Fraction( firstProduct + secondProduct,
lhs.denominator * rhs.denominator); } public override string ToString() {
string s = numerator.ToString() + "/" + denominator.ToString(); return s;
} //biến thành viên lưu tử số và mẫu số private int numerator; private int denominator;
} public class Tester { static void Main() {
Fraction f1 = new Fraction( 3, 4);
Console.WriteLine("f1:{0}",f1.ToString());
Fraction f2 = new Fraction( 2, 4); Console.WriteLine("f2:{0}",f2.ToString());
Fraction f3 = f1 + f2; Console.WriteLine("f1 + f2 = f3:{0}",f3.ToString());
Fraction f4 = f3 + 5; Console.WriteLine("f4 = f3 + 5:{0}",f4.ToString()); Fraction f5 = new Fraction( 2, 4); if( f5 == f2 ) {
Console.WriteLine("f5:{0}==f2:{1}",     f5.ToString(), f2.ToString());
}
}
}
```

Lớp Fraction bắt đầu với hai hàm khởi dựng: một hàm lấy một tử số và mẫu số, còn hàm kia lấy chỉ lấy một số làm tử số. Tiếp sau hai bộ khởi dựng là hai toán tử chuyển đổi.
Toán tử chuyển đổi đầu tiên chuyển một số nguyên sang một phân số:

```
public static implicit operator Fraction( int theInt ) { return new Fraction( theInt); }
```

Sự chuyển đổi này được thực hiện một cách ngầm định bởi vì bất cứ số nguyên nào cũng có thể được chuyển thành một phân số bằng cách thiết lập tử số bằng giá trị số nguyên và mẫu số có giá trị là 1. Việc thực hiện này có thể giao lại cho phương thức khởi dựng lấy một tham số. Toán tử chuyển đổi thứ hai được thực hiện một cách tường minh, chuyển từ một Fraction ra một số nguyên:

```
public static explicit operator int( Fraction theFraction
) { return theFraction.numerator / theFraction.denominator; }
```

Bởi vì trong ví dụ này sử dụng phép chia nguyên, phép chia này sẽ cắt bỏ phần phân chỉ lấy phần nguyên. Do vậy nếu phân số có giá trị là 16/15 thì kết quả số nguyên trả về là 1. Một số các phép chuyển đổi tốt hơn bằng cách sử dụng làm tròn số.
Tiếp theo sau là toán tử so sánh bằng (==) và toán tử so sánh không bằng (!=). Chúng ta nên nhớ rằng khi thực thi toán tử so sánh bằng thì cũng phải thực thi toán tử so sánh không bằng. Chúng ta đã định nghĩa giá trị bằng nhau giữa hai Fraction khi tử số bằng tử số và mẫu số bằng mẫu số. Vi dụ, như hai phân số 3/4 và 6/8 thì không được so sánh là bằng nhau. Một lần nữa, một sự thực thi tốt hơn là tối giản tử số và mẫu số khi đó 6/8 sẽ đơn giản thành ¾ và khi đó so sánh hai phân số sẽ bằng nhau.
Trong lớp này chúng ta cũng thực thi phủ quyết phương thức Equals() của lớp object, do đó đối tượng Fraction của chúng ta có thể được đối xử một cách đa hình với bất cứ đối tượng khác. Trong phần thực thi của phương thức chúng ta ủy thác việc so sánh lại cho toán tử so sánh bằng cách gọi toán tử (==).

Lớp Fraction có thể thực thi hết tất cả các toán tử số học như cộng, trừ, nhân, chia. Tuy nhiên, trong phạm vi nhỏ hẹp của minh họa chúng ta chỉ thực thi toán tử cộng, và thậm chí phép cộng ở đây được thực hiện đơn giản nhất. Chúng ta thử nhìn lại, nếu hai mẫu số bằng nhau thì ta cộng tử số:

```
public static Fraction operator + ( Fraction lhs, Fraction rhs) { if ( lhs.denominator == rhs.denominator) { return
new Fraction( lhs.numerator + rhs.numerator, lhs.denominator); } }
```

Nếu mẫu số không cùng nhau, thì chúng ta thực hiện nhân chéo:

```
int firstProduct = lhs.numerator * rhs.denominator; int secondProduct = rhs.numerator * lhs.denominator;
return new Fraction( firstProduct + secondProduct,
lhs.denominator * rhs.denominator);
```

Cuối cùng là sự phủ quyết phương thức ToString() của lớp object, phương thức mới này thực hiện viết xuất ra nội dung của phân số dưới dạng : tử số / mẫu số:

```
public override string ToString() { string s = numerator.ToString() + "/" + denominator.ToString(); return s; }
```

Chúng ta tạo một chuỗi mới bằng cách gọi phương thức ToString() của numerator. Do numerator là một đối tượng, nên trình biên dịch sẽ ngầm định thực hiện boxing số nguyên numerator và sau đó gọi phương thức ToString(), trả về một chuỗi thể hiện giá trị của số nguyên numerator. Sau đó ta nối chuỗi với "/" và cuối cùng là chuỗi thể hiện giá trị của mẫu số.

Với lớp Fraction đã tạo ra, chúng ta thực hiện kiểm tra lớp này. Đầu tiên chúng ta tạo ra hai phân số 3/4, và 2/4:

```
Fraction f1 = new Fraction( 3, 4); Console.WriteLine("f1:{0}",f1.ToString());
Fraction f2 = new Fraction( 2, 4); Console.WriteLine("f2:{0}",f2.ToString());
```

Kết quả thực hiện các lệnh trên như sau: In Fraction Constructor(int, int) f1: 3/4
In Fraction Constructor(int, int) f2: 2/4

Do trong phương phức khởi dựng của lớp Fraction chúng ta có gọi hàm WriteLine() để xuất ra thông tin bộ khởi dựng nên khi tạo đối tượng (new) thì cũng các thông tin này sẽ được hiển thị.

Dòng tiếp theo trong hàm Main() sẽ gọi toán tử cộng, đây là phương thức tĩnh. Mục đích của toán tử này là cộng hai phân số và trả về một phân số mới là tổng của hai phân số đưa vào:

```
Fraction f3 = f1 + f2; Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString());
```

Hai câu lệnh trên sẽ cho ra kết quả như sau:

```
In operator +
In Fraction Constructor( int, int) f1 + f2 = f3: 5/4
```

Toán tử + được gọi trước sau đó đến phương thức khởi dựng của đối tượng f3. Phương thức khởi dựng này lấy hai tham số nguyên để tạo tử số và mẫu số của phân số mới f3.

Hai câu lệnh tiếp theo cộng một giá trị nguyên vào phân số f3 và gán kết quả mới về cho phân số mới f4:

```
Fraction f4 = f3 + 5; Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString());
```

Kết quả được trình bày theo thứ tự sau:

In implicit conversion to Fraction
In Fraction Construction(int) In operator+ In Fraction Constructor(int, int) f3 + 5 = f4: 25/4

Ghi chú: rằng

Toán tử chuyển đổi ngầm định được gọi khi chuyển 5 thành một phân số. Phân số được tạo ra từ toán tử chuyển đổi ngầm định này gọi phương thức khởi dựng một tham số để tạo phân số mới 5/1. Phân số mới này sẽ được chuyển thành toán hạng trong phép cộng với phân số f3 và kết quả trả về là phân số f4 là tổng của hai phân số trên.
Thử nghiệm cuối cùng là tạo một phân số mới f5, rồi sau đó gọi toán tử nạp chồng so sánh bằng để kiểm tra xem hai phân số có bằng nhau hay không.

Bài tập:

Viết chương trình tạo lớp "Phân số", có phép toán cộng, trừ phân số..

Bài tập nâng cao:

Thêm các phép toán nhân và chia cho lớp "phân số"

Những trọng tâm cần chú ý trong bài:

- Biết được các kiến thức về toán tử;
- Biết các kiến thức về sự hỗ trợ nạp chồng toán tử trong các ngôn ngữ .Net khác;
- Trang bị các kiến thức và kỹ năng về nạp chồng các toán tử trong C#;
- Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:
    - Biết được các kiến thức về toán tử;
    - Biết các kiến thức về sự hỗ trợ nạp chồng toán tử trong các ngôn ngữ .Net khác;
    - Trang bị các kiến thức và kỹ năng về nạp chồng các toán tử trong C#;

+ Về kỹ năng: Tạo và thực thi được ứng dụng có sử dụng nạp chồng toán tử.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Tạo và thực thi được ứng dụng có sử dụng nạp chồng toán tử.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 5: CẤU TRÚC
Mã bài: MĐ 11 - 06

Giới thiệu:
Cấu trúc là một kiểu dữ liệu trừu tượng, rất hữu ích trong việc lập trình cần kiểm soát nhiều thông tin

Mục tiêu:

+ Hiểu các kiến thức về cấu trúc trong C#.Net;
+ Sử dụng các kiến thức để tạo và thực thi cấu trúc vào các bài tập thực hành;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập.

Nội dung chính:

1. Định nghĩa một cấu trúc

Cú pháp để khai báo một cấu trúc cũng tương tự như cách khai báo một lớp:
[thuộc tính] [bổ sung truy cập] struct <tên cấu trúc> [:
danh sách giao diện] { [thành viên của cấu trúc]
}
Ví dụ sau minh họa cách tạo một cấu trúc. Kiểu Location thể hiện một điểm trong không gian hai chiều. Lưu ý rằng cấu trúc Location này được khai báo chính xác như khi thực hiện khai báo với một lớp, ngoại trừ việc sử dụng từ khóa struct. Ngoài ra cũng lưu ý rằng hàm khởi dựng của Location lấy hai số nguyên và gán những giá trị của chúng cho các biến thành viên, x và y. Tọa độ x và y của Location được khai báo như là thuộc tính.
Tạo một cấu trúc.
---
using System; public struct Location { public Location(int xCoordinate, int yCoordinate) { xVal = xCoordinate; yVal = yCoordinate; } public int x { get { return xVal; } set { xVal=value; } } public int y { get { return yVal; } set { yVal=value; } } public override string ToString() { return (String.Format("{0}, {1}", xVal, yVal)); } // thuộc tính private lưu toạ độ x, y private int xVal; private int yVal; } public class Tester { public void myFunc(Location loc) { loc.x = 50; loc.y = 100; Console.WriteLine("Loc1 location: {0}", loc); } static void Main() { Location loc1 = new Location(200, 300); Console.WriteLine("Loc1 location: {0}", loc1); Tester t = new Tester(); t.myFunc(loc1); Console.WriteLine("Loc1 location: {0}", loc1); } }
Không giống như những lớp, cấu trúc không hỗ trợ việc thừa kế. Chúng được thừa kế ngầm định từ lớp object (tương tự như tất cả các kiểu dữ liệu trong C#, bao gồm các kiểu dữ liệu xây dựng sẵn) nhưng không thể kế thừa từ các lớp khác hay cấu trúc khác. Cấu trúc cũng được ngầm định là sealed, điều này có ý nghĩa là không có lớp nào hay bất cứ cấu trúc nào có thể dẫn xuất từ nó. Tuy nhiên, cũng giống như các lớp, cấu trúc có thể thực thi nhiều giao diện. Sau đây là một số sự khác nhau nữa là:
Không có bộ hủy và bộ khởi tạo mặc định tùy chọn: Những cấu trúc không có bộ hủy và cũng không có bộ khởi tạo mặc định không tham số tùy chọn. Nếu chúng ta không cung cấp bất cứ bộ khởi tạo nào thì cấu trúc sẽ được cung cấp một bộ khởi tạo mặc định, khi đó giá trị 0 sẽ được thiết lập cho tất cả các dữ liệu thành viên hay những giá trị mặc định tương ứng cho từng kiểu dữ liệu (bảng 4.2). Nếu chúng ta cung cấp bất cứ bộ khởi dựng nào thì chúng ta phải khởi tạo tất cả các trường trong cấu trúc.
Không cho phép khởi tạo: chúng ta không thể khởi tạo các trường thể hiện (instance fields) trong cấu trúc, do đó đoạn mã nguồn sau sẽ **không hợp lệ**:
private int xVal = 20; private int yVal = 50;
mặc dù điều này thực hiện tốt đối với lớp.

Cấu trúc được thiết kế hướng tới đơn giản và gọn nhẹ. Trong khi các dữ liệu thành viên private hỗ trợ việc che dấu dữ liệu và sự đóng gói. Một vài người lập trình có cảm giác rằng điều này phá hỏng cấu trúc. Họ tạo một dữ liệu thành viên public, do vậy đơn giản thực thi một cấu trúc. Những người lập trình khác có cảm giác rằng những thuộc tính cung cấp một giao diện rõ ràng, đơn giản và việc thực hiện lập trình tốt đòi hỏi phải che dấu dữ liệu thậm chí với dữ liệu rất đơn giản. Chúng ta sẽ chọn cách nào, nói chung là phụ thuộc vào quan nệm thiết kế của từng người lập trình. Dù chọn cách nào thì ngôn ngữ C# cũng hỗ trợ cả hai cách tiếp cận.

2. Tạo cấu trúc

Chúng ta tạo một thể hiện của cấu trúc bằng cách sử dụng từ khóa new trong câu lệnh gán, như khi chúng ta tạo một đối tượng của lớp. Như trong ví dụ trê, lớp Tester tạo một thể hiện của Location như sau:
Location loc1 = new Location(200, 300);
Ở đây một thể hiện mới tên là loc1 và nó được truyền hai giá trị là 200 và 300.

2.1.Cấu trúc là một kiểu giá trị

Phần định nghĩa của lớp Tester trong ví dụ 7.1 trên bao gồm một đối tượng Location là loc1 được tạo với giá trị là 200 và 300. Dòng lệnh sau sẽ gọi thực hiện bộ khởi tạo của cấu trúc Location:
Location loc1 = new Location(200, 300); Sau đó phương tức WriteLine() được gọi:
Console.WriteLine("Loc1 location: {0}", loc1);
Dĩ nhiên là WriteLine chờ đợi một đối tượng, nhưng Location là một cấu trúc (một kiểu giá trị). Trình biên dịch sẽ tự động boxing cấu trúc (cũng giống như trình biên dịch đã làm với các kiểu dữ liệu giá trị khác). Một đối tượng sau khi boxing được truyền vào cho phương thức WriteLine(). Tiếp sau đó là phương thức ToString() được gọi trên đối tượng boxing này, do cấu trúc ngầm định kế thừa từ lớp object, và nó cũng có thể đáp ứng sự đa hình, bằng cách phủ quyết các phương thức như bất cứ đối tượng nào khác.
Loc1 location 200, 300
Tuy nhiên do cấu trúc là kiểu giá trị, nên khi truyền vào trong một hàm, thì chúng chỉ truyền giá trị vào hàm. Cũng như ta thấy ở dòng lệnh kế tiếp, khi đó một đối tượng Location được truyền vào phương thức myFunc():
t.myFunc(loc1);
Trong phương thức myFunc() hai giá trị mới được gán cho x và y, sau đó giá trị mới sẽ được xuất ra màn hình:
Loc1 location: 50, 100
Khi phương thức myFunc() trả về cho hàm gọi (Main()) và chúng ta gọi tiếp phương thức WriteLine() một lần nữa thì giá trị không thay đổi:
Loc1 location: 200, 300
Như vậy cấu trúc được truyền vào hàm như một đối tượng giá trị, và một bản sao sẽ được tạo bên trong phương thức myFunc(). Nếu chúng ta thử đổi khai báo của Location là class như sau:
public class Location
Sau đó chạy lại chương trình thì có kết quả:
Loc1 location: 200, 3000
In myFunc loc: 50, 100
Loc1 location: 50, 100
Lúc này Location là một đối tượng tham chiếu nên khi truyền vào phương thức myFunc() thì việc gán giá trị mới cho x và y điều làm thay đổi đối tượng Location.

2.2. Gọi bộ khởi dựng mặc định

Như đề cập ở phần trước, nếu chúng ta không tạo bộ khởi dựng thì một bộ khởi dựng mặc định ngầm định sẽ được trình biên dịch tạo ra. Chúng ta có thể nhìn thấy điều này nếu bỏ bộ khởi dựng tạo ra:

/*public Location(int xCoordinate , int yCoordinate) { xVal = xCoordinate; yVal = yCoordinate; } */

và ta thay dòng lệnh đầu tiên trong hàm Main() tạo Location có hai tham số bằng câu lệnh tạo không tham số như sau:

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

Bởi vì lúc này không có phương thức khởi dựng nào khai báo, một phương thức khởi dựng ngầm định sẽ được gọi. Kết quả khi thực hiện giống như sau:

Loc1 location 0, 0

In myFunc loc: 50, 100

Loc1 location: 0, 0

Bộ khởi tạo mặc định đã thiết lập tất cả các biến thành viên với giá trị 0.

Đối với lập trình viên C++ lưu ý, trong ngôn ngữ C#, từ khóa new không phải luôn luôn tạo đối tượng trên bộ nhớ heap. Các lớp thì được tạo ra trên heap, trong khi các cấu trúc thì được tạo trên stack. Ngoài ra, khi new được bỏ qua (sẽ bàn tiếp trong phần sau), thì bộ khởi dựng sẽ không được gọi. Do ngôn ngữ C# yêu cầu phải có phép gán trước khi sử dụng, chúng ta phải khởi tạo tường minh tất cả các biến thành viên trước khi sử dụng chúng trong cấu trúc.

2.3. Tạo cấu trúc không gọi new

Bởi vì Location là một cấu trúc không phải là lớp, do đó các thể hiện của nó sẽ được tạo trong stack. Trong ví dụ trên khi toán tử new được gọi:

Location loc1 = new Location(200, 300);

kết quả một đối tượng Location được tạo trên stack.

Tuy nhiên, toán tử new gọi bộ khởi dựng của lớp Location, không giống như với một lớp, cấu trúc có thể được tạo ra mà không cần phải gọi toán tử new. Điều này giống như các biến của các kiểu dữ liệu được xây dựng sẵn (như int, long, char,..) được tạo ra. Ví dụ sau minh họa việc tạo một cấu trúc không sử dụng toán tử new.

Đây là một sự khuyến cáo, trong ví dụ sau chúng ta minh họa cách tạo một cấu trúc mà không phải sử dụng toán tử new bởi vì có sự khác nhau giữa C# và ngôn ngữ C++ và sự khác nhau này chính là cách ngôn ngữ C# đối xử với những lớp khác những cấu trúc. Tuy nhiên, việc tạo một cấu trúc mà không dùng từ khóa new sẽ không có lợi và có thể tạo một chương trình khó hiểu, tiềm ẩn nhiều lỗi, và khó duy trì. Chương trình họa sau sẽ không được khuyến khích.

Tạo một cấu trúc mà không sử dụng new.

---

using System; public struct Location { public Location(int xCoordinate, int yCoordinate) { xVal = xCoordinate; yVal = yCoordinate; } public int x { get { return xVal; } set { xVal=value; } } public int y { get { return yVal; } set { yVal=value; } } public override string ToString() { return (string.Format("{0} ,{1}", xVal, yVal)); } // biến thành viên lưu tọa độ x, y public int xVal; public int yVal; } public class Tester { static void Main() { Location loc1; loc1.xVal = 100; loc1.yVal = 250; Console.WriteLine("loc1"); } }

Trong ví dụ trên chúng ta khởi tạo biến thành viên một cách trực tiếp, trước khi gọi bất cứ phương thức nào của loc1 và trước khi truyền đối tượng cho phương thức

WriteLine():
loc1.xVal = 100; loc2.yVal = 250;
Nếu chúng ta thử bỏ một lệnh gán và biên dịch lại:
static void Main() { Location loc1; loc1.xVal = 100; //loc1.yVal = 250; Console.WriteLine(loc1); }
Chúng ta sẽ nhận một lỗi biên dịch như sau:
Use of unassigned local variable 'loc1'
Một khi mà chúng ta đã gán tất cả các giá trị của cấu trúc, chúng ta có thể truy cập giá trị thông qua thuộc tính x và thuộc tính y:
static void Main() { Location loc1; // gán cho biến thành viên loc1.xVal = 100; loc1.yVal = 250; // sử dụng thuộc tính loc1.x = 300; loc1.y = 400; Console.WriteLine(loc1); }
Hãy cẩn thận với việc sử dụng các thuộc tính. Mặc dù cấu trúc cho phép chúng ta hỗ trợ đóng gói bằng việc thiết lập thuộc tính private cho các biến thành viên. Tuy nhiên bản thân thuộc tính thật sự là phương thức thành viên,và chúng ta không thể gọi bất cứ phương thức thành viên nào cho đến khi chúng ta khởi tạo tất cả các biến thành viên.
Như ví dụ trên ta thiết lập thuộc tính truy cập của hai biến thành viên xVal và yVal là public vì chúng ta phải khởi tạo giá trị của hai biến thành viên này bên ngoài của cấu trúc, trước khi các thuộc tính được sử dụng.

Bài tập:

Tạo cấu trúc "Học sinh" có các thông tin cơ bản như: Họ tên, giới tính, ngày sinh, địa chỉ

Bài tập nâng cao:

Tạo 5 học sinh từ cấu trúc "Học sinh", viết chương trình cho phép thay đổi các thông tin cơ bản của từng học sinh

Những trọng tâm cần chú ý trong bài:

- Hiểu các kiến thức về cấu trúc trong C#.Net;
- Sử dụng các kiến thức để tạo và thực thi cấu trúc vào các bài tập thực hành;
- Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập;

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:
    - Hiểu các kiến thức về cấu trúc trong C#.Net;
    - Sử dụng các kiến thức để tạo và thực thi cấu trúc vào các bài tập thực hành;

+ Về kỹ năng: Tạo và thực thi được cấu trúc đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Tạo và thực thi được ứng dụng đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 6: MẢNG, CHỈ MỤC, TẬP HỢP
Mã bài: MĐ 11 - 08

Giới thiệu:
Kiểu dữ liệu dùng để quản lý dữ liệu phổ biến được dùng hiện nay là mảng, ngoài ra còn có kiểu chỉ mục và tập hợp.

Mục tiêu:
+ Hiểu các kiến thức về mảng và danh sách mảng;
+ Hiểu các kiến thức về bộ chỉ mục và tập hợp.
+ Biết các kiến thức về bộ từ điển dựng sẵn trong C#.
+ Giải quyết được một số bài tập trên mảng, chỉ mục và tập hợp;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập.

Nội dung chính:

1. Mảng

Ngôn ngữ C# cung cấp cú pháp chuẩn cho việc khai báo những đối tượng Array. Tuy nhiên, cái thật sự được tạo ra là đối tượng của kiểu System.Array. Mảng trong ngôn ngữ C# kết hợp cú pháp khai báo mảng theo kiểu ngôn ngữ C và kết hợp với định nghĩa lớp do đó thể hiện của mảng có thể truy cập những phương thức và thuộc tính của System.Array.
Một số các thuộc tính và phương thức của lớp System.Array
Các phương thức và thuộc tính của System.Array

Thành viên	Mô tả
BinarySearch()	Phương thức tĩnh public tìm kiếm một mảng một chiều đãsắp thứ tự.
Clear()	Phương thức tĩnh public thiết lập các thành phần của mảngvề 0 hay null.
Copy()	Phương thức tĩnh public đã nạp chồng thực hiện sao chépmột vùng của mảng vào mảng khác.
CreateInstance()	Phương thức tĩnh public đã nạp chồng tạo một thể hiện mớicho mảng
IndexOf()	Phương thức tĩnh public trả về chỉ mục của thể hiện đầu tiênchứa giá trị trong mảng một chiều
LastIndexOf()	Phương thức tĩnh public trả về chỉ mục của thể hiện cuốicùng của giá trị trong mảng một chiều
Reverse()	Phương thức tĩnh public đảo thứ tự của các thành phần trongmảng một chiều
Sort()	Phương thức tĩnh public sắp xếp giá trị trong mảng mộtchiều.
IsFixedSize	Thuộc tính public giá trị bool thể hiện mảng có kích thướccố định hay không.
IsReadOnly	Thuộc tính public giá trị bool thể hiện mảng chỉ đọc haykhông

IsSynchronized	Thuộc tính public giá trị bool thể hiện mảng có hỗ trợ thread-safe
Length	Thuộc tính public chiều dài của mảng
Rank	Thuộc tính public chứa số chiều của mảng

1.1. Khai báo mảng

Chúng ta có thể khai báo một mảng trong C# với cú pháp theo sau:
<kiểu dữ liệu>[] <tên mảng>
Ta có khai báo như sau:
int[] myIntArray;
Cặp dấu ngoặc vuông ([]) báo cho trình biên dịch biết rằng chúng ta đang khai báo một mảng. Kiểu dữ liệu là kiểu của các thành phần chứa bên trong mảng. Trong ví dụ bên trên. myIntArray được khai báo là mảng số nguyên.
Chúng ta tạo thể hiện của mảng bằng cách sử dụng từ khóa new như sau:
myIntArray = new int[6];
Khai báo này sẽ thiết lập bên trong bộ nhớ một mảng chứa sáu số nguyên.
Dành cho lập trình viên Visual Basic, thành phần đầu tiên luôn bắt đầu 0, không có cách nào thiết lập cận trên và cận dưới của mảng, và chúng ta cũng không thể thiết lập lại kích thước của mảng.
Điều quan trọng để phân biệt giữa bản thân mảng (tập hợp các thành phần) và các thành phần trong mảng. Đối tượng myIntArray là một mảng, thành phần là năm số nguyên được lưu giữ. Mảng trong ngôn ngữ C# là kiểu dữ liệu tham chiếu, được tạo ra trên heap. Do đó myIntArray được cấp trên heap. Những thành phần của mảng được cấp phát dựa trên các kiểu dữ liệu của chúng. Số nguyên là kiểu dữ liệu giá trị, và do đó những thành phần của myIntArray là kiểu dữ liệu giá trị, không phải số nguyên được boxing. Một mảng của kiểu dữ liệu tham chiếu sẽ không chứa gì cả nhưng tham chiếu đến những thành phần được tạo ra trên heap.

1.2. Giá trị mặc định

Khi chúng ta tạo một mảng có kiểu dữ liệu giá trị, mỗi thành phần sẽ chứa giá trị mặc định của kiểu dữ liệu (xem bảng 4.2, kiểu dữ liệu và các giá trị mặc định). Với khai báo:
myIntArray = new int[5];
sẽ tạo ra một mảng năm số nguyên, và mỗi thành phần được thiết lập giá trị mặc định là 0, đây cũng là giá trị mặc định của số nguyên.
Không giống với mảng kiểu dữ liệu giá trị, những kiểu tham chiếu trong một mảng không được khởi tạo giá trị mặc định. Thay vào đó, chúng sẽ được khởi tạo giá trị null. Nếu chúng ta cố truy cập đến một thành phần trong mảng kiểu dữ liệu tham chiếu trước khi chúng được khởi tạo giá trị xác định, chúng ta sẽ tạo ra một ngoại lệ.
Giả sử chúng ta tạo ra một lớp Button. Chúng ta khai báo một mảng các đối tượng Button với cú pháp sau:
Button[] myButtonArray;
và chúng ta tạo thể hiện của mảng như sau:
myButtonArray = new Button[3];
Chúng ta có thể viết ngắn gọn như sau:
Button muButtonArray = new Button[3];
Không giống với ví dụ mảng số nguyên trước, câu lệnh này không tao ra một mảng với những tham chiếu đến ba đối tượng Button. Thay vào đó việc này sẽ tạo ra một mảng myButtonArray với ba tham chiếu null. Để sử dụng mảng này, đầu tiên chúng ta phải

tạo và gán đối tượng Button cho từng thành phần tham chiếu trong mảng. Chúng ta có thể tạo đối tượng trong vòng lặp và sau đó gán từng đối tượng vào trong mảng.

1.3. Truy cập các thành phần trong mảng

Để truy cập vào thành phần trong mảng ta có thể sử dụng toán tử chỉ mục ([]). Mảng dùng cơ sở 0, do đó chỉ mục của thành phần đầu tiên trong mảng luôn luôn là 0. Như ví dụ trước thành phần đầu tiên là myArray[0].
Như đã trình bày ở phần trước, mảng là đối tượng, và do đó nó có những thuộc tính. Một trong những thuộc tính hay sử dụng là Length, thuộc tính này sẽ báo cho biết số đối tượng trong một mảng. Một mảng có thể được đánh chỉ mục từ 0 đến Length –1. Do đó nếu có năm thành phần trong mảng thì các chỉ mục là: 0, 1, 2, 3, 4.
Ví dụ sau minh họa việc sử dụng các khái niệm về mảng từ đầu chương tới giờ. Trong ví dụ một lớp tên là Tester tạo ra một mảng kiểu Employee và một mảng số nguyên. Tạo các đối tượng Employee sau đó in hai mảng ra màn hình.
Làm việc với một mảng.

---

namespace Programming_CSharp { using System; // tạo một lớp đơn giản để lưu trữ trong mảng public class Employee { // bộ khởi tạo lấy một tham số public Employee(int empID
) { this.empID = empID; } public override string
ToString() { return empID.ToString(); } // biến thành viên private private int empID; private int size; } public class Tester { static void Main() { int[] intArray;
Employee[] empArray; intArray = new int[5]; empArray = new Employee[3]; // tạo đối tượng đưa vào mảng for(int i = 0;
i < empArray.Length; i++) { empArray[i] = new
Employee(i+5); } // xuất mảng nguyên for(int i = 0; i <
intArray.Length; i++) {
Console.Write(intArray[i].ToString()+"\t"); } // xuất mảng
Employee for(int i = 0; i < empArray.Length; i++) {
Console.WriteLine(empArray[i].ToString()+"\t"); } } } }

Kết quả:
0 0 0 0 0 5 6 7

Ví dụ bắt đầu với việc định nghĩa một lớp Employee, lớp này thực thi một bộ khởi dựng lấy một tham số nguyên. Phương thức ToString() được kế thừa từ lớp Object được phủ quyết để in ra giá trị empID của đối tượng Employee.
Các kiểu tạo ra là khai báo rồi mới tạo thể hiện của hai mảng. Mảng số nguyên được tự động thiết lập giá trị 0 mặc định cho từng số nguyên trong mảng. Nội dung của mảng Employee được tạo bằng các lệnh trong vòng lặp.
Cuối cùng, nội dung của cả hai mảng được xuất ra màn hình console để đảm bảo kết quả như mong muốn; năm giá trị đầu của mảng nguyên, ba số sau cùng là của mảng Employee.

1.4. Khởi tạo các thành phần trong mảng

Chúng ta có thể khởi tạo nội dung của một mảng ngay lúc tạo thể hiện của mảng bằng cách đặt những giá trị bên trong dấu ngoặc ({}). C# cung cấp hai cú pháp để khởi tạo các thành phần của mảng, một cú pháp dài và một cú pháp ngắn:
int[] myIntArray = new int[5] { 2, 4, 6, 8, 10}; int[] myIntArray = { 2, 4, 6, 8, 10};

Không có sự khác biệt giữa hai cú pháp trên, và hầu hết các chương trình đều sử dụng cú pháp ngắn hơn do sự tự nhiên và lười đánh nhiều lệnh của người lập trình.

Sử dụng từ khóa params

Chúng ta có thể tạo một phương thức rồi sau đó hiển thị các số nguyên ra màn hình console bằng cách truyền vào một mảng các số nguyên và sử dụng vòng lặp foreach để duyệt qua từng thành phần trong mảng. Từ khóa params cho phép chúng ta truyền một số biến của tham số mà không cần thiết phải tạo một mảng.
Trong ví dụ kế tiếp, chúng ta sẽ tạo một phương thức tên DisplayVals(), phương thức này sẽ lấy một số các biến của tham số nguyên:
public void DisplayVals(params int[] intVals)
Phương thức có thể xem mảng này như thể một mảng được tạo ra tường minh và được truyền vào tham số. Sau đó chúng ta có thể tự do lặp lần lượt qua các thành phần trong mảng giống như thực hiện với bất cứ mảng nguyên nào khác:

```
foreach (int i in intVals)
{ Console.WriteLine("DisplayVals: {0}", i);
}
```

Tuy nhiên, phương thức gọi không cần thiết phải tạo tường minh một mảng, nó chỉ đơn giản truyền vào các số nguyên, và trình biên dịch sẽ kết hợp những tham số này vào trong một mảng cho phương thức DisplayVals, ta có thể gọi phương thức như sau:

```
t.DisplayVals(5,6,7,8);
```

và chúng ta có thể tự do tạo một mảng để truyền vào phương thức nếu muốn:

```
int [] explicitArray = new int[5] {1,2,3,4,5};
t.DisplayArray(explicitArray);
```

Ví dụ sau cung cấp tất cả mã nguồn để minh họa sử dụng cú pháp params.
Minh họa sử dụng params.

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; public class
Tester { static void Main() { Tester t = new Tester();
t.DisplayVals(5,6,7,8); int[] explicitArray = new int[5] {1,2,3,4,5};
t.DisplayVals(explicitArray); } public void
DisplayVals( params int[] intVals) { foreach (int i in intVals) {
Console.WriteLine("DisplayVals {0}", i); } } } }
---------------------------------------------------------------------------
```

Kết quả:

```
DisplayVals 5
DisplayVals 6
DisplayVals 7
DisplayVals 8
DisplayVals 1
DisplayVals 2
DisplayVals 3
DisplayVals 4
DisplayVals 5
---------------------------------------------------------------------------
```

2. Câu lệnh foreach

Câu lệnh lặp foreach khá mới với những người đã học ngôn ngữ C, từ khóa này được sử dụng trong ngôn ngữ Visual Basic. Câu lệnh foreach cho phép chúng ta lặp qua tất cả các mục trong một mảng hay trong một tập hợp.
Cú pháp sử dụng lệnh lặp foreach như sau:
foreach (<kiểu dữ liệu thành phần> <tên truy cập> in
<mảng/tập hợp>) { // thực hiện thông qua <tên truy cập> tương ứng với // từng mục trong mảng hay tập hợp }
Do vậy, chúng ta có thể cải tiến ví dụ trước bằng cách thay việc sử dụng vòng lặp for bằng vòng lặp foreach để truy cập đến từng thành phần trong mảng.
Sử dụng foreach.
---
namespace Programming_CSharp { using System; // tạo một lớp đơn giản để lưu trữ trong mảng public class Employee { // bộ khởi tạo lấy một tham số public Employee(int empID
) { this.empID = empID; } public override string
ToString() { return empID.ToString(); } // biến thành viên private private int empID; private int size; } public class Tester { static void Main() { int[] intArray;
Employee[] empArray; intArray = new int[5]; empArray = new Employee[3]; // tạo đối tượng đưa vào mảng for(int i = 0;
i < empArray.Length; i++) { empArray[i] = new
Employee(i+10); } // xuất mảng nguyên foreach (int i in intArray) { Console.Write(i.ToString()+"\t"); } // xuất mảng Employee foreach (Employee e in empArray) { Console.WriteLine(e.ToString()+"\t"); } } } }
Kết quả của ví dụ trên cũng tương tự như ví dụ trước. Tuy nhiên, với việc sử dụng vòng lặp for ta phải xác định kích thước của mảng, sử dụng biến đếm tạm thời để truy cập đến từng thành phần trong mảng:
for (int i = 0 ; i < empArray.Length; i++) { Console.WriteLine(empArray[i].ToString()); }
Thay vào đó ta sử dụng foreach , khi đó vòng lặp sẽ tự động trích ra từng mục tuần tự trong mảng và gán tạm vào một tham chiếu đối tượng khai báo ở đầu vòng lặp:
foreach (Employee e in empArray) { Console.WriteLine(e.ToString()+"\t"); }
Đối tượng được trích từ mảng có kiểu dữ liệu tương ứng. Do đó chúng ta có thể sử dụng bất cứ thành viên public của đối tượng.

3. Mảng đa chiều

Từ đầu chương đến giờ chúng ta chỉ nói đến mảng các số nguyên hay mảng các đối tượng. Tất cả các mảng này đều là mảng một chiều. Mảng một chiều trong đó các thành phần của nó chỉ đơn giản là các đối tượng kiểu giá trị hay đối tượng tham chiếu. Mảng có thể được tổ chức phức tạp hơn trong đó mỗi thành phần là một mảng khác, việc tổ chức này gọi là mảng đa chiều.
Mảng hai chiều được tổ chức thành các dòng và cột, trong đó các dòng là được tính theo hàng ngang của mảng, và các cột được tính theo hàng dọc của mảng.
Mảng ba chiều cũng có thể được tạo ra nhưng thường ít sử dụng do khó hình dung.
Trong mảng ba chiều những dòng bây giờ là các mảng hai chiều.
Ngôn ngữ C# hỗ trợ hai kiểu mảng đa chiều là:
Mảng đa chiều cùng kích thước: trong mảng này mỗi dòng trong mảng có cùng kích thước với nhau. Mảng này có thể là hai hay nhiều hơn hai chiều.

Mảng đa chiều không cùng kích thước: trong mảng này các dòng có thể không cùng kích thước với nhau.

Mảng đa chiều cùng kích thước

Mảng đa chiều cùng kích thước còn gọi là mảng hình chữ nhật (rectanguler array). Trong mảng hai chiều cổ điển, chiều đầu tiên được tính bằng số dòng của mảng và chiều thứ hai được tính bằng số cột của mảng.
Để khai báo mảng hai chiều, chúng ta có thể sử dụng cú pháp theo sau:
<kiểu dữ liệu> [,] <tên mảng>
Ví dụ để khai báo một mảng hai chiều có tên là myRectangularArray để chứa hai dòng và ba cột các số nguyên, chúng ta có thể viết như sau:
int [,] myRectangularArray;
Ví dụ tiếp sau đây minh họa việc khai báo, tạo thể hiện, khởi tạo và in nội dung ra màn hình của một mảng hai chiều. Trong ví dụ này, vòng lặp for được sử dụng để khởi tạo các thành phần trong mảng.
Mảng hai chiều.

---

namespace Programming_CSharp { using System; public class Tester { static void Main() { // khai báo số dòng và số cột của mảng const int rows = 4; const int columns = 3; // khai báo mảng 4x3 số nguyên int [,] rectangularArray = new int[rows, columns]; // khởi tạo các thành phần trong mảng for(int i = 0; i < rows; i++) { for(int j = 0; j < columns; j++) { rectangularArray[i,j] = i+j; } } // xuất nội dung ra màn hình for(int i = 0; i < rows; i++) {
for(int j = 0; j < columns; j++) {
Console.WriteLine("rectangularArray[{0},{1}] = {2}", i, j, rectangularArray[i, j]); } }
} } }

Kết quả:
rectangularArray[0,0] = 0 rectangularArray[0,1] = 1 rectangularArray[0,2] = 2 rectangularArray[1,0] = 1 rectangularArray[1,1] = 2 rectangularArray[1,2] = 3 rectangularArray[2,0] = 2 rectangularArray[2,1] = 3 rectangularArray[2,2] = 4 rectangularArray[3,0] = 3 rectangularArray[3,1] = 4 rectangularArray[3,2] = 5

Trong ví dụ này, chúng ta khai báo hai giá trị:
const int rows = 4; const int columns = 3; hai giá trị này được sử dụng để khai báo số chiều của mảng:
int [,] rectangularArray = new int[rows, columns];
Trong cú pháp này, dấu ngoặc vuông trong int[,] chỉ ra rằng đang khai báo một kiểu dữ liệu là mảng số nguyên, và dấu phẩy (,) chỉ ra rằng đây là mảng hai chiều (hai dấu phẩy khai báo mảng ba chiều, và nhiều hơn nữa). Việc tạo thể hiện thực sự của mảng ở lệnh new int[rows,columns] để thiết lập kích thước của mỗi chiều. Ở đây khai báo và tạo thể hiện được kết hợp với nhau.
Chương trình khởi tạo tất cả các giá trị các thành phần trong mảng thông qua hai vòng lặp for. Lặp thông qua mỗi cột của mỗi dòng. Do đó, thành phần đầu tiên được khởi tạo là rectangularArray[0,0], tiếp theo bởi rectangularArray[0,1] và đến rectangularArray[0,2]. Một khi điều này thực hiện xong thì chương trình sẽ chuyển qua thực hiện tiếp ở dòng tiếp tục: rectangularArray[1,0], rectangularArray[1,1],

rectangularArray[1,2]. Cho đến khi tất cả các cột trong tất cả các dòng đã được duyệt qua tức là tất cả các thành phần trong mảng đã được khởi tạo.
Như chúng ta đã biết, chúng ta có thể khởi tạo mảng một chiều bằng cách sử dụng danh sách các giá trị bên trong dấu ngoặc ({}). Chúng ta cũng có thể làm tương tự với mảng hai chiều. Trong ví dụ sau khai báo mảng hai chiều rectangularArray, và khởi tạo các thành phần của nó thông qua các danh sách các giá trị trong ngoặc, sau đó in ra nội dung của nội dung.
Khởi tạo mảng đa chiều.

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; public class Tester { static void Main() { // khai báo biến lưu số dòng số cột mảng const int rows = 4; const int columns = 3; // khai báo và định nghĩa mảng 4x3 int[,] rectangularArray = { {0,1,2}, {3,4,5}, {6,7,8},{9,10,11} }; // xuất nội dung của mảng for( int i = 0; i < rows; i++) { for(int j = 0; j
< columns; j++) {
Console.WriteLine("rectangularArray[{0},{1}] = {2}", i, j, rectangularArray[i,j]); } } } } }
Kết quả:
rectangularArray[0,0] = 0 rectangularArray[0,1] = 1 rectangularArray[0,2] = 2
rectangularArray[1,0] = 3 rectangularArray[1,1] = 4 rectangularArray[1,2] = 5
rectangularArray[2,0] = 6 rectangularArray[2,1] = 7 rectangularArray[2,2] = 8
rectangularArray[3,0] = 9 rectangularArray[3,1] = 10 rectangularArray[3,2] = 11
---------------------------------------------------------------------------
```

Ví dụ trên cũng tương tự như ví dụ trước, nhưng trong ví dụ này chúng ta thực hiện việc khởi tạo trực tiếp khi tạo các thể hiện:
int[,] rectangularArray = { {0,1,2}, {3,4,5}, {6,7,8},{9,10,11} };
Giá trị được gán thông qua bốn danh sách trong ngoặc móc, mỗi trong số đó là có ba thành phần, bao hàm một mảng 4x3.
Nếu chúng ta viết như sau:
int[,] rectangularArray = { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} };
thì sẽ tạo ra một mảng 3x4.

Mảng đa chiều có kích khác nhau

Cũng như giới thiệu trước kích thước của các chiều có thể không bằng nhau, điều này khác với mảng đa chiều cùng kích thước. Nếu hình dạng của mảng đa chiều cùng kích thước có dạng hình chữ nhật thì hình dạng của mảng này không phải hình chữ nhật vì các chiều của chúng không điều nhau.
Khi chúng ta tạo một mảng đa chiều kích thước khác nhau thì chúng ta khai báo số dòng trong mảng trước. Sau đó với mỗi dòng sẽ giữ một mảng, có kích thước bất kỳ. Những mảng này được khai báo riêng. Sau đó chúng ta khởi tạo giá trị các thành phần trong những mảng bên trong.
Trong mảng này, mỗi chiều là một mảng một chiều. Để khai báo mảng đa chiều có kích thước khác nhau ta sử dụng cú pháp sau, khi đó số ngoặc chỉ ra số chiều của mảng:
<kiểu dữ liệu> [] [] ...
Chúng ta có thể khai báo mảng số nguyên hai chiều khác kích thước tên myJagged-Array như sau:
int [] [] myJaggedArray;

Chúng ta có thể truy cập thành phần thứ năm của mảng thứ ba bằng cú pháp: myJagged-Array[2][4].
Ví dụ sau tạo ra mảng khác kích thước tên myJaggedArray, khởi tạo các thành phần, rồi sau đó in ra màn hình. Để tiết kiệm thời gian, chúng ta sử dụng mảng các số nguyên để các thành phần của nó được tự động gán giá trị mặc định. Và ta chỉ cần gán một số giá trị cần thiết.
Mảng khác chiều.

```
namespace Programming_CSharp { using System; public class Tester { static void Main() { const int rows = 4; // khai báo mảng tối đa bốn dòng int[][] jaggedArray = new int[rows][]; // dòng đầu tiên có 5 phần tử jaggedArray[0] = new int[5]; // dòng thứ hai có 2 phần tử jaggedArray[1]
= new int[2]; // dòng thứ ba có 3 phần tử jaggedArray[2] = new int[3]; // dòng cuối cùng có 5 phần tử jaggedArray[3] = new int[5]; // khởi tạo một vài giá trị cho các thành phần của mảng jaggedArray[0][3] = 15; jaggedArray[1][1] =
12; jaggedArray[2][1] = 9; jaggedArray[2][2] = 99; jaggedArray[3][0] = 10; jaggedArray[3][1] = 11; jaggedArray[3][2] = 12; jaggedArray[3][3] = 13; jaggedArray[3][4] = 14; for(int i = 0; i < 5; i++) {
Console.WriteLine("jaggedArray[0][{0}] = {1}", i, jaggedArray[0][i]); } for(int i = 0; i < 2; i++) { Console.WriteLine("jaggedArray[1][{0}] = {1}", i, jaggedArray[1][i]); } for(int i = 0; i < 3; i++) { Console.WriteLine("jaggedArray[2][{0}] = {1}", i, jaggedArray[2][i]); } for(int i = 0; i < 5; i++) { Console.WriteLine("jaggedArray[3][{0}] = {1}", i, jaggedArray[3][i]); } } } }
```

Kết quả:

```
jaggedArray[0][0] = 0
jaggedArray[0][1] = 0 jaggedArray[0][2] = 0 jaggedArray[0][3] = 15 jaggedArray[0][4] = 0 jaggedArray[1][0] = 0 jaggedArray[1][1] = 12 jaggedArray[2][0] = 0 jaggedArray[2][1] = 9 jaggedArray[2][2] = 99 jaggedArray[3][0] = 10 jaggedArray[3][1] = 11 jaggedArray[3][2] = 12 jaggedArray[3][3] = 13 jaggedArray[3][4] = 14
```

Trong ví dụ này, mảng được tạo với bốn dòng:

```
int[][] jaggedArray = new int[rows][];
```

Chiều thứ hai không xác định. Do sau đó chúng ta có thể khai báo mỗi dòng có kích thước khác nhau. Bốn lệnh sau tạo cho mỗi dòng một mảng một chiều có kích thước khác nhau:

```
// dòng đầu tiên có 5 phần tử jaggedArray[0] = new int[5]; // dòng thứ hai có 2 phần tử jaggedArray[1] = new int[2]; // dòng thứ ba có 3 phần tử jaggedArray[2] = new int[3]; // dòng cuối cùng có 5 phần tử jaggedArray[3] = new int[5];
```

Sau khi tạo các dòng cho mảng xong, ta thực hiện việc đưa các giá trị vào các thành phần của mảng. Và cuối cùng là xuất nội dung của mảng ra màn hình.
Khi chúng ta truy cập các thành phần của mảng kích thước bằng nhau, chúng ta đặt tất cả các chỉ mục của các chiều vào trong cùng dấu ngặc vuông:

```
rectangularArray[i,j]
```

Tuy nhiên với mảng có kích thước khác nhau ta phải để từng chỉ mục của từng chiều trong đấu ngoặc vuông riêng:

```
jaggedArray[i][j]
```

Chuyển đổi mảng

Những mảng có thể chuyển đổi với nhau nếu những chiều của chúng bằng nhau và nếu các kiểu của các thành phần có thể chuyển đổi được. Chuyển đổi tường minh giữa các mảng xảy ra nếu các thành phần của những mảng có thể chuyển đổi tường minh. Và ngược lại, chuyển đổi ngầm định của mảng xảy ra nếu các thành phần của những mảng có thể chuyển đổi ngầm định.

Nếu một mảng chứa những tham chiếu đến những đối tượng tham chiếu, một chuyển đổi có thể được tới một mảng của những đối tượng cơ sở. Ví dụ sau minh họa việc chuyển đổi một mảng kiểu Button đến một mảng những đối tượng.

Chuyển đổi giữa những mảng.

---

namespace Programming_CSharp { using System; // tạo lớp để lưu trữ trong mảng public class Employee { public Employee(int empID) { this.empID = empID; } public override string ToString() { return empID.ToString(); } // biến thành viên private int empID; private int size; } public class Tester { // phương thức này lấy một mảng các object // chúng ta truyền vào mảng các đối tượng Employee // và sau đó là mảng các string, có sự chuyển đổi ngầm // vì cả hai điều dẫn xuất từ lớp object public static void PrintArray(object[] theArray) {

Console.WriteLine("Contents of the Array: {0}", theArray.ToString()); // in ra từng thành phần trong mảng foreach (object obj in theArray) { // trình biên dịch sẽ gọi obj.ToString() Console.WriteLine("Value: {0}", obj); } } static void Main() { // tạo mảng các đối tượng Employee Employee[] myEmployeeArray = new Employee[3]; // khởi tạo các đối tượng của mảng for (int i = 0; i < 3; i++) { myEmployeeArray[i] = new Employee(i+5); } // hiểu thị giá trị của mảng PrintArray(myEmployeeArray); // tạo mảng gồm hai chuỗi string[] array ={ "hello", "world"}; // xuất ra nội dung của chuỗi PrintArray(array); } } }

Kết quả:

Contents of the Array Programming_CSharp.Employee[]

Value: 5

Value: 6

Value: 7

Contents of the Array Programming_CSharp.String[]

Value: hello

Value: world

Ví dụ trên bắt đầu bằng việc tạo một lớp đơn giản Employee như các ví dụ trước. Lớp Tester bây giờ được thêm một phương thức tĩnh PrintArray() để xuất nội dung của mảng, phương thức này có khai báo một tham số là mảng một chiều các đối tượng object:

public static void PrintMyArray(object[] theArray)

object là lớp cơ sở ngầm định cho tất cả các đối tượng trong môi trường .NET, nên nó được khai báo ngầm định cho cả hai lớp string và Employee.

Phương thức PrintArray thực hiện hai hành động. Đầu tiên, là gọi phương thức ToString() của mảng:

Console.WriteLine("Contents of the Array {0}", theArray.ToString());

Tên của kiểu dữ liệu mảng được in ra:

Contents of the Array Programming_CSharp.Employee[] ...

Contents of the Array System.String[]
Sau đó phương thức PrintArray thực hiện tiếp việc gọi phương thức ToString() trong mỗi thành phần trong mảng nhận được. Do ToString() là phương thức ảo của lớp cơ sở object, và chúng ta đã thực hiện phủ quyết trong lớp Employee. Nên phương thức ToString() của lớp Employee được gọi. Việc gọi ToString() có thể không cần thiết, nhưng nếu gọi thì cũng không có hại gì và nó giúp cho ta đối xử với các đối tượng một cách đa hình.

System.Array

Lớp mảng Array chứa một số các phương thức hữu ích cho phép mở rộng những khả năng của mảng và làm cho mảng mạnh hơn những mảng trong ngôn ngữ khác (xem bảng 9.1). Hai phương thức tĩnh hữu dụng của lớp Array là Sort() và Reverse(). Có một cách hỗ trợ đầy đủ cho những kiểu dữ liệu nguyên thủy như là kiểu. Đưa mảng làm việc với những kiểu khác như Button có một số khó khăn hơn. Ví dụ sau minh họa việc sử dụng hai phương thức để thao tác đối tượng chuỗi.
Sử dụng Array.Sort() và Array.Reverse().

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; public class Tester { public static void PrintArray(object[] theArray)
{ foreach( object obj in theArray) {
Console.WriteLine("Value: {0}", obj); }
Console.WriteLine("\n"); } static void Main() { string[] myArray = { "Who", "is","Kitty","Mun" }; PrintArray( myArray ); Array.Reverse( myArray ); PrintArray( myArray
); string[] myOtherArray = { "Chung", "toi", "la",
"nhung","nguoi","lap","trinh", "may", "tinh" };
PrintArray( myOtherArray ); Array.Sort( myOtherArray );
PrintArray( myOtherArray ); } } }
---------------------------------------------------------------------------
```

Kết quả:

```
Value: Who
Value: is
Value: Kitty
Value: Mun
Value: Mun
Value: Kitty
Value: is
Value: Who
Value: Chung
Value: toi
Value:la
Value: nhung
Value: nguoi
Value: lap
Value: trinh
Value: may
Value: tinh
Value: Chung
```

Value: la
Value: lap
Value: may
Value: nguoi
Value: nhung
Value: tinh
Value: toi
Value: trinh

Ví dụ bắt đầu bằng việc tạo mảng myArray, mảng các chuỗi với các từ sau:
“Who”, “is”, “Kitty”, ”Mun”
mảng này được in ra, sau đó được truyền vào cho hàm Array.Reverse(), kết quả chúng ta thấy là kết quả của chuỗi như sau:
Value: Mun
Value: Kitty
Value: is
Value: Who
Tương tự như vậy, ví dụ cũng tạo ra mảng thứ hai, myOtherArray, chứa những từ sau:
“Chung”, “toi”, “la”, “nhung”,”nguoi”,”lap”,”trinh”, “máy”, “tính”
Sau khi gọi phương thức Array.Sort() thì các thành phần của mảng được sắp xếp lại theo thứ tự alphabe: Value: Chung
Value: la
Value: lap
Value: may
Value: nguoi
Value: nhung
Value: tinh
Value: toi
Value: trinh

4. Bộ chỉ mục và giao diện tập hợp

Đôi khi chúng ta chúng ta mong muốn truy cập một tập hợp bên trong một lớp như thể bản thân lớp là một mảng. Ví dụ, giả sử chúng ta tạo một điều khiển kiểu ListBox tên là myListBox chứa một danh sách các chuỗi lưu trữ trong một mảng một chiều, một biến thành viên private myStrings. Một List Box chứa các thuộc tính thành viên và những phương thức và thêm vào đó mảng chứa các chuỗi của nó. Tuy nhiên, có thể thuận tiện hơn nếu có thể truy cập mảng ListBox với chỉ mục như thể ListBox là một mảng thật sự. Ví dụ, ta có thể truy cập đối tượng ListBox được tạo ra như sau:
string theFirstString = myListBox[0]; string theLastString = myListBox[myListBox.Length - 1];
Bộ chỉ mục là một cơ chế cho phép các thành phần client truy cập một tập hợp chứa bên trong một lớp bằng cách sử dụng cú pháp giống như truy cập mảng ([]). Chỉ mục là một loại thuộc tính đặc biệt và bao gồm các phương thức get() và set() để xác nhận những hành vi của chúng.
Chúng ta có thể khai báo thuộc tính chỉ mục bên trong của lớp bằng cách sử dụng cú pháp như sau:
<kiểu dữ liệu> this [<kiểu dữ liệu> <đối mục>]
{ get; set; }

Kiểu trả về được quyết định bởi kiểu của đối tượng được trả về bởi bộ chỉ mục, trong khi đó kiểu của đối mục được xác định bởi kiểu của đối mục dùng để làm chỉ số vào trong tập hợp chứa đối tượng đích. Mặc dù kiểu của chỉ mục thường dùng là các kiểu nguyên, chúng ta cũng có thể dùng chỉ mục cho tập hợp bằng các kiểu dữ liệu khác, như kiểu chuỗi. Chúng ta cũng có thể cung cấp bộ chỉ mục với nhiều tham số để tạo ra mảng đa chiều.
Từ khóa this tham chiếu đến đối tượng nơi mà chỉ mục xuất hiện. Như một thuộc tính bình thường, chúng ta cũng có thể định nghĩa phương thức get() và set() để xác định đối tượng nào trong mảng được yêu cầu truy cập hay thiết lập.
Ví dụ sau khai báo một điều khiển ListBox, tên là ListBoxTest, đối tượng này chứa một mảng đơn giản (myStrings) và một chỉ mục để truy cập nội dung của mảng.
Đối với lập trình C++, bộ chỉ mục đưa ra giống như việc nạp chồng toán tử chỉ mục ([]) trong ngôn ngữ C++. Toán tử chỉ mục không được nạp chồng trong ngôn ngữ C#,và được thay thế bởi bộ chỉ mục.
Sử dụng bộ chỉ mục.

---

namespace Programming_CSharp { using System; // tạo lớp
ListBox public class ListBoxTest { // khởi tạo ListBox với
một chuỗi public ListBoxTest(params string[] initialStrings) { // cấp phát không gian cho chuỗi strings = new String[256]; // copy chuỗi truyền từ tham số foreach (string s in initialStrings) { strings[ctr++] = s; } } // thêm một chuỗi public void Add(string theString) { if (ctr
>= strings.Length) { // xử lý khi chỉ mục sai } else strings[ctr++] = theString; } // thực hiện bộ truy cập public string this[int index] { get { if (index < 0 || index >= strings.Length) { // xử lý chỉ mục sai } return strings[index]; } Set { if (index >= ctr) { // xử lý lỗi chỉ mục không tồn tại } else strings[index] = value; // lấy số lượng chuỗi được lưu giữ public int GetNumEntries() { return ctr; } // các biến thành viên lưu giữ mảng cho bộ chỉ mục private string[] strings; private int ctr = 0; } // lớp thực thi public class Tester { static void Main() {
// tạo một đối tượng ListBox mới và khởi tạo ListBoxTest lbt = new ListBoxTest("Hello","World"); // thêm một số chuỗi vào lbt lbt.Add("Who"); lbt.Add("is"); lbt.Add("Ngoc"); lbt.Add("Mun"); // dùng bộ chỉ mục string strTest = "Universe"; lbt[1] = strTest; // truy cập và
xuất tất cả các chuỗi for(int i = 0; i < lbt.GetNumEntries(); i++) { Console.WriteLine("lbt[{0}]: {1}", i, lbt[i]); } } } }

Kết quả: lbt[0]: Hello lbt[1]: Universe lbt[2]: Who
lbt[3]: is lbt[4]: Ngoc lbt[5]: Mun

Trong chương trình trên, đối tượng ListBox lưu giữ một mảng các chuỗi myStrings và một biến thành viên ctr đếm số chuỗi được chứa trong mảng myStrings.
Chúng ta khởi tạo một mảng tối đa 256 chuỗi như sau:
myStrings = new String[256];
Phần còn lại của bộ khởi dựng là thêm các chuỗi được truyền vào tham số, và đơn giản dùng lệnh lặp foreach để lấy từng thành phần trong mảng tham số đưa vào myStrings
Ghi chú: Nếu chúng ta không biết số lượng bao nhiêu tham số được truyền vào phương thức, chúng ta sử dụng từ khóa params như đã mô tả trong phần trước của chương.

Phương thức Add() của ListBoxTest không làm gì khác hơn là thêm một chuỗi mới vào bên trong mảng myStrings.
Tuy nhiên phương thức quan trọng của ListBoxTest là bộ chỉ mục. Một bộ chỉ mục thì không có tên nên ta dùng từ khóa this:
public string this [int index]
Cú pháp của bộ chỉ mục cũng tương tự như những thuộc tính. Chúng có thể có phương thức get() hay set() hay cả hai phương thức. Phương thức get() được thực thi đầu tiên bằng cách kiểm tra giá trị biên của chỉ mục và giả sử chỉ mục đòi hỏi hợp lệ, thì phương thức trả về giá trị đòi hỏi:
get { if (index < 0 || index >= myStrings.Length) { // xử lý chỉ mục sai } return myStrings[index]; }
Đối với phương thức set() thì đầu tiên nó sẽ kiểm trả xem chỉ mục của đối tượng cần lấy có vượt quá số lượng của các đối tượng trong mảng hay không. Nếu giá trị chỉ mục hợp lệ tức là tồn tại một đối tượng có chỉ mục tương đương, phương thức sẽ bắt đầu thiết lập lại giá trị của đối tượng này. Từ khóa value được sử dụng để tham chiếu đến tham số đưa vào trong phép gán của thuộc tính:
set { if (index >= ctr) { // chỉ mục không tồn tại } }
Do vậy, nếu chúng ta viết:
myStrings[10] = "Hello C#";
trình biên dịch sẽ gọi phương thức set() của bộ chỉ mục trên đối tượng và truyền vào một chuỗi “Hello C#” như là một tham số ngầm định tên là value.

Bộ chỉ mục và phép gán

Trong ví dụ 9.9, chúng ta không thể gán đến một chỉ mục nếu nó không có giá trị. Do đó, nếu chúng ta viết:
lbt[10] = "ah!";
Chúng ta có thể viết điều kiện ràng buộc bên trong phương thức set(), lưu ý rằng chỉ mục mà chúng ta truyền vào là 10 lớn hơn bộ đếm số đối tượng hiện thời là 6.
Dĩ nhiên, chúng ta có thể sử dụng phương thức set() cho phép gán, đơn giản là phải xử lý những chỉ mục mà ta nhận được. Để làm được điều này, chúng ta phải thay đổi phương thức set() để kiểm tra giá trị Length của bộ đệm hơn là giá trị hiện thời của bộ đếm số đối tượng.
Nếu một giá trị được nhập vào cho chỉ mục chưa có giá trị, chúng ta có thể cập nhật bộ đếm như sau:
set { if (index >= strings.Length) { // chỉ mục vượt quá số tối đa của mảng } else { strings[index] = value; if (ctr < index+1) ctr = index+1; } }
Điều này có thể cho phép chúng ta tạo một mảng phân mảng các giá trị, khi đó ta có thể gán cho đối tượng có chỉ mục thứ 10 mà không cần phải có phép gán với đối tượng trước có chỉ mục là 9. Điều này hoàn toàn thực hiện tốt vì ban đầu chúng ta đã cấp phát mảng 256 các phần tử. Do đó chỉ cần truy cập đến đối tượng có chỉ mục từ 0 đến 255 là hợp lệ. Khi đó ta có thể viết:
lbt[10] = “ah!”;
Kết quả thực hiện tương tự như sau:
lbt[0]: Hello
lbt[1]: Universe lbt[2]: Who
lbt[3]: is lbt[4]: Ngoc lbt[5]: Mun lbt[6]: lbt[7]: lbt[8]: lbt[9]:
lbt[10]: “ah!”

Sử dụng kiểu chỉ số khác

Ngôn ngữ C# không đòi hỏi chúng ta phải sử dụng giá trị nguyên làm chỉ mục trong một tập hợp. Khi chúng ta tạo một lớp có chứa một tập hợp và tạo một bộ chỉ mục, bộ chỉ mục này có thể sử dụng kiểu chuỗi làm chỉ mục hay những kiểu dữ liệu khác ngoài kiểu số nguyên thường dùng.

Trong trường hợp lớp ListBox trên, chúng ta muốn dùng giá trị chuỗi làm chỉ mục cho mảng string. Ví dụ sau sử dụng chuỗi làm chỉ mục cho lớp ListBox. Bộ chỉ mục gọi phương thức findString() để lấy một giá trị trả về là một số nguyên dựa trên chuỗi được cung cấp. Lưu ý rằng ở đây bộ chỉ mục được nạp chồng và bộ chỉ mục từ ví dụ trước vẫn còn tồn tại.

Nạp chồng chỉ mục.

---

namespace Programming_CSharp { using System; // tạo lớp List Box public class ListBoxTest
{
// khởi tạo với những chuỗi public ListBoxTest(params string[] initialStrings) { // cấp phát chuỗi strings = new String[256]; // copy các chuỗi truyền vào foreach(string s in initialStrings) { strings[ctr++] = s; }
} // thêm một chuỗi vào cuối danh sách public void Add(string theString) { strings[ctr] = theString; ctr++; } // bộ chỉ mục public string this [int index] { get {
if (index < 0 || index >= strings.Length) { // chỉ mục không hợp lệ
} } set { strings[index] = value; } private int findString(string searchString) { for(int i = 0; i < strings.Length; i++) { if (strings[i].StartsWith(searchString)) { return i; } } return -1; } // bộ chỉ mục dùng chuỗi public string this [string index]
{ get { if (index.Length == 0) { / /xử lý khi chuỗi rỗng } return this[findString(index)]; } set { strings[findString(index)] = value; }
} //lấy số chuỗi trong mảng public int GetNumEntries() { return ctr; } // biến thành viên lưu giữ mảng các chuỗi private string[] strings;
// biến thành viên lưu giữa số chuỗi trong mảng private int ctr = 0; } public class Tester { static void Main() {
// tạo đối tượng List Box và sau đó khởi tạo ListBoxTest lbt = new ListBoxTest("Hello","World"); // thêm các chuỗi vào lbt.Add("Who"); lbt.Add("is"); lbt.Add("Ngoc"); lbt.Add("Mun"); // truy cập bộ chỉ mục string str = "Universe"; lbt[1] = str; lbt["Hell"] = "Hi"; //lbt["xyzt"] = "error!"; // lấy tất cả các chuỗi ra for(int i = 0; i < lbt.GetNumEntries();i++) { Console.WriteLine("lbt[{0}] = {1}", i, lbt[i]);
}
}
}
}

Kết quả: lbt[0]: Hi lbt[1]: Universe lbt[2]: Who
lbt[3]: is lbt[4]: Ngoc lbt[5]: Mun

Ví dụ trên thì tương tự như ví dụ trước ngoại trừ việc thêm vào một chỉ mục được nạp chồng lấy tham số chỉ mục là chuỗi và phương thức findString() tạo ra để lấy chỉ mục nguyên từ chuỗi.

Phương thức findString() đơn giản là lặp mảng strings cho đến khi nào tìm được chuỗi có ký tự đầu tiên trùng với ký tự đầu tiên của chụỏi tham số. Nếu tìm thấy thì trả về chỉ mục của chuỗi, trường hợp ngược lại trả về -1.
Như chúng ta thấy trong hàm Main(), lệnh truy cập chỉ mục thứ hai dùng chuỗi làm tham số chỉ mục, như đã làm với số nguyên trước:
lbt["hell"] = "Hi";
Khi đó nạp chồng chỉ mục sẽ được gọi, sau khi kiểm tra chuỗi hợp lệ tức là không rỗng, chuỗi này sẽ được truyền vào cho phương thức findString(), kết quả findString() trả về là một chỉ mục nguyên, số nguyên này sẽ được sử dụng làm chỉ mục:
return this[findString(index)];
Ví dụ trên tồn tại lỗi khi một chuỗi truyền vào không phù hợp với bất cứ chuỗi nào trong mảng, khi đó giá trị trả về là –1. Sau đó giá trị này được dùng làm chỉ mục vào chuỗi mảng strings. Điều này sẽ tạo ra một ngoại lệ (System.NullReferenceException).
Trường hợp này xảy ra khi chúng ta bỏ đấu comment của lệnh:
lbt["xyzt"] = "error!";
Các trường hợp phát sinh lỗi này cần phải được loại bỏ, đây có thể là bài tập cho chúng ta làm thêm và việc này hết sức cần thiết.

Giao diện tập hợp

Môi trường .NET cung cấp những giao diện chuẩn cho việc liệt kê, so sánh, và tạo các tập hợp. Một số các giao diện trong số đó được liệt kê trong bảng 9.2 sau:
Giao diện cho tập hợp

Giao diện	Mục đích
IEnumerable	Liệt kê thông qua một tập hợp bằng cách sử dụngforeach.
ICollection	Thực thi bởi tất cả các tập hợp để cung cấp phương thức CopyTo() cũng như các thuộc tính Count,ISReadOnly, ISSynchronized, và SyncRoot.
IComparer	So sánh giữa hai đối tượng lưu giữ trong tập hợp đểsắp xếp các đối tượng trong tập hợp.
IList	Sử dụng bởi những tập hợp mảng được chỉ mục
IDictionary	Dùng trong các tập hợp dựa trên khóa và giá trị nhưHashtable và SortedList.
IDictionaryEnumerator	Cho phép liệt kê dùng câu lệnh foreach qua tập hợphỗ trợ IDictionary.

Giao diện Ienumerable

Chúng ta có thể hỗ trợ cú pháp foreach trong lớp ListBoxTest bằng việc thực thi giao diện IEnumerator. Giao diện này chỉ có một phương thức duy nhất là GetEnumerator(), công việc của phương thức là trả về một sự thực thi đặc biệt của IEnumerator. Do vậy, ngữ nghĩa của lớp Enumerable là nó có thể cung cấp một Enumerator:
public IEnumerator GetEnumerator() { return (IEnumerator) new ListBoxEnumerator(this); }
Enumerator phải thực thi những phương thức và thuộc tính IEnumerator. Chúng có thể được thực thi trực tiếp trong lớp chứa (trong trường hợp này là lớp ListBoxTest) hay bởi một lớp phân biệt khác. Cách tiếp cận thứ hai thường được sử dụng nhiều hơn, do chúng được đóng gói trong lớp Enumerator hơn là việc phân vào trong các lớp chứa.
Do lớp Enumerator được xác định cho lớp chứa, vì theo như trên thì lớp ListBoxEnumerator phải biết nhiều về lớp ListBoxTest. Nên chúng ta phải tạo cho nó một sự thực thi riêng chứa bên trong lớp ListBoxTest. Lưu ý rằng phương thức GetEnumerator truyền đối tượng List- BoxTest hiện thời (this) cho enumerator. Điều này cho phép enumerator có thể liệt kê được các thành phần trong tập hợp của đối tượng ListBoxTest. Ở đây lớp thực thi Enumerator là ListBoxEnumerator, đây là một lớp private được định nghĩa bên trong lớp ListBoxTest. Lớp này thực thi đơn giản bao gồm một thuộc tính public, và hai phương thức public là MoveNext(), và Reset(). Đối tượng ListBoxTest được truyền vào như một đối mục của bộ khởi tạo. Ở đây nó được gán cho biến thành viên myLBT. Trong hàm khởi tạo này cũng thực hiện thiết lập giá trị biến thành viên index là -1, chỉ ra rằng chưa bắt đầu thực hiện việc enumerator đối tượng:
public ListBoxEnumerator(ListBoxTest lbt) { this.lbt = lbt; index = -1; }
Phương thức MoveNext() gia tăng index và sau đó kiểm tra để đảm bảo rằng việc thực hiện không vượt quá số phần tử trong tập hợp của đối tượng:

public bool MoveNext() { index++; if (index >= lbt.strings.Length) return false; else return true; }
Phương thức IEnumerator.Reset() không làm gì cả nhưng thiết lập lại giá trị của index là -1. Thuộc tính Current trả về đối tượng chuỗi hiện hành. Đó là tất cả những việc cần làm cho lớp ListBoxTest thực thi một giao diện IEnumerator. Câu lệnh foreach sẽ được gọi để đem về một enumerator, và sử dụng nó để liệt kê lần lượt qua các thành phần trong mảng. Sau đây là toàn bộ chương trình minh họa cho việc thực thi trên.
Tạo lớp ListBox hỗ trợ enumerator.

```
namespace Programming_CSharp { using System; using System.Collections; // tạo một control đơn giản public class ListBoxTest: IEnumerable
{
// lớp thực thi riêng ListBoxEnumerator private class ListBoxEnumerator : IEnumerator { public ListBoxEnumerator(ListBoxTest lbt) { this.lbt = lbt; index = -1; } // gia tăng index và đảm bảo giá trị này hợp lệ public bool MoveNext() { index++; if (index >= lbt.strings.Length) return false; else return true; } public void Reset() { index = -1; } public object Current
{ get { return( lbt[index]); } } private ListBoxTest lbt; private int index; } // trả về Enumerator public IEnumerator GetEnumerator() { return (IEnumerator) new ListBoxEnumerator(this); } // khởi tạo listbox với chuỗi public ListBoxTest (params string[] initStr) { strings = new String[10]; // copy từ mảng chuỗi tham số foreach (string s in initStr) { strings[ctr++] = s;
} } public void Add(string theString) { strings[ctr] = theString; ctr++; } // cho phép truy cập giống như mảng public string this[int index] { get { if ( index < 0 || index >= strings.Length) {
// xử lý index sai } return strings[index]; } set { strings[index] = value;
}
} // số chuỗi nắm giữ public int GetNumEntries() { return ctr; } private string[] strings; private int ctr = 0; } public class Tester { static void Main() { ListBoxTest lbt = new ListBoxTest("Hello", "World"); lbt.Add("What"); lbt.Add("Is"); lbt.Add("The"); lbt.Add("C"); lbt.Add("Sharp"); string subst = "Universe"; lbt[1] = subst;
// truy cập tất cả các chuỗi int count =1;
foreach (string s in lbt) { Console.WriteLine("Value {0}: {1}",count, s); count++; }
}
}
}
```

Kết quả:

```
Value 1: Hello
Value 2: Universe
Value 3: What
Value 4: Is
Value 5: The
Value 6: C Value 7: Sharp
Value 8:
Value 9:
Value 10:
```

Chương trình thực hiện bằng cách tạo ra một đối tượng ListBoxTest mới và truyền hai chuỗi vào cho bộ khởi dựng. Khi một đối tượng được tạo ra thì mảng của chuỗi được định nghĩa có kích thước 10 chuỗi. Năm chuỗi sau được đưa vào bằng phương thức Add(). Và chuỗi thứ hai sau đó được cập nhật lại giá trị mới. Sự thay đổi lớn nhất của chương trình trong phiên bản này là câu lệnh foreach được gọi để truy cập từng chuỗi trong ListBox. Vòng lặp foreach tự động sử dụng giao diện IEnumerator bằng cách gọi phương thức GetEnumerator(). Một đối tượng ListBoxEnumerator được tạo ra và giá trị index = -1 được thiết lập trong bộ khởi tạo. Vòng lặp foreach sau đó gọi phương thức MoveNext(), khi đó index sẽ được gia tăng đến 0 và trả về true. Khi đó foreach sử dụng thuộc tính Current để nhận lại chuỗi hiện hành. Thuộc tính Current gọi chỉ mục của ListBox và nhận lại chuỗi được lưu trữ tại vị trí 0. Chuỗi này được gán cho biến s được định nghĩa trong vòng lặp, và chuỗi này được hiển thị ra màn hình console. Vòng lặp tiếp tục thực hiện tuần tự từngt bước: MoveNext(), Current(), hiển thị chuỗi cho đến khi tất cả các chuỗi trong list box được hiển thị. Trong minh họa này chúng ta khai báo mảng chuỗi có 10 phần tử, nên trong kết quả ta thấy chuỗi ở vị trí 8, 9, 10 không có nội dung.

Giao diện ICollection

Một giao diện quan trọng khác cho những mảng và cho tất cả những tập hợp được cung cấp bởi .NET Framework là ICollection. ICollection cung cấp bốn thuộc tính: Count, IsReadOnly, IsSynchronized, và SyncRoot. Ngoài ra ICollection cũng cung cấp một phương thức CopyTo(). Thuộc tính thường được sử dụng là Count, thuộc tính này trả về số thành phần trong tập hợp:

```
for(int i = 0; i < myIntArray.Count; i++) { //... }
```

Ở đây chúng ta sử dụng thuộc tính Count của myIntArray để xác định số đối tượng có thể được sử dụng trong mảng.

Giao diện IComparer

Giao diện IComparer cung cấp phương thức Compare(), để so sánh hai phần tử trong một tập hợp có thứ tự. Phương thức Compare() thường được thực thi bằng cách gọi phương thức CompareTo() của một trong những đối tượng. CompareTo() là phương thức có trong tất cả đối tượng thực thi IComparable. Nếu chúng ta muốn tạo ra những lớp có thể được sắp xếp bên trong một tập hợp thì chúng ta cần thiết phải thực thi IComparable.

.NET Framework cung cấp một lớp Comparer thực thi IComparable và cung cấp một số thực thi cần thiết. Phần danh sách mảng sau sẽ đi vào chi tiết việc thực thi IComparable.

5. Danh sách mảng, hàng đợi, ngăn xếp

Việc sử dụng mảng có kích thước cố định là không thích hợp cũng như là chúng ta không thể đoán trước được kích thước của mảng cần thiết.

Lớp ArrayList là một kiểu dữ liệu mảng mà kích thước của nó được gia tăng một cách động theo yêu cầu. ArrayList cung cấp một số phương thức và những thuộc tính cho những thao tác liên quan đến mảng. Một vài phương thức và thuộc tính quan trọng của ArrayList được liệt kê trong bảng 9.3 như sau:

Các phương thức và thuộc tính của ArrayList

Phương thức-thuộc tính	Mục đích

Adapter()	Phương thức static tạo một wrapper ArrayList cho đốitượng IList
FixedSize()	Phương thức static nạp chồng trả về sanh sách đối tượng như là một wrapper. Danh sách có kích thước cố định, các thành phần của nó có thể được sửa chữa nhưng không thểthêm hay xóa.
ReadOnly()	Phương thức static nạp chồng trả về danh sách lớp như là một wrapper, chỉ cho phép đọc.
Repeat()	Phương thức static trả về một ArrayList mà những thànhphần của nó được sao chép với giá trị xác định.
Synchronized()	Phương thức static trả về danh sách wrapper được thread-safe
Capacity	Thuộc tính để get hay set số thành phần trong ArrayList.
Count	Thuộc tính nhận số thành phần hiện thời trong mảng
IsFixedSize	Thuộc tính kiểm tra xem kích thước của ArrayList có cốđịnh hay không
IsReadOnly	Thuộc tính kiểm tra xem ArrayList có thuộc tính chỉ đọc hay không.
IsSynchronized	Thuộc tính kiểm tra xem ArrayList có thread-safe haykhông
Item()	Thiết lập hay truy cập thành phần trong mảng tại vị trí xácđịnh. Đây là bộ chỉ mục cho lớp ArrayList.
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồngbộ truy cập đến ArrayList
Add()	Phương thức public để thêm một đối tượng vào ArrayList
AddRange()	Phương thức public để thêm nhiều thành phần của mộtICollection vào cuối của ArrayList
BinarySearch()	Phương thức nạp chồng public sử dụng tìm kiếm nhị phân để định vị một thành phần xác định trong ArrayList đượcsắp xếp.
Clear()	Xóa tất cả các thành phần từ ArrayList
Clone()	Tạo một bản copy
Contains()	Kiểm tra một thành phần xem có chứa trong mảng haykhông
CopyTo()	Phương thức public nạp chồng để sao chép một ArrayListđến một mảng một chiều.
GetEnumerator()	Phương thức public nạp chồng trả về một enumerator dùngđể lặp qua mảng
GetRange()	Sao chép một dãy các thành phần đến một ArrayList mới

IndexOf()	Phương thức public nạp chồng trả về chỉ mục vị trí đầu tiênxuất hiện giá trị
Insert()	Chèn một thành phần vào trong ArrayList
InsertRange(0	Chèn một dãy tập hợp vào trong ArrayList
LastIndexOf()	Phương thức public nạp chồng trả về chỉ mục trị trí cuối cùng xuất hiện giá trị.
Remove()	Xóa sự xuất hiện đầu tiên của một đối tượng xác định.
RemoveAt()	Xóa một thành phần ở vị trí xác định.
RemoveRange()	Xóa một dãy các thành phần.
Reverse()	Đảo thứ tự các thành phần trong mảng.
SetRange()	Sao chép những thành phần của tập hợp qua dãy nhữngthành phần trong ArrayList.
Sort()	Sắp xếp ArrayList.
ToArray()	Sao chép những thành phần của ArrayList đến một mảngmới.
TrimToSize()	Thiết lập kích thước thật sự chứa các thành phần trongArrayList

Khi tạo đối tượng ArrayList, không cần thiết phải định nghĩa số đối tượng mà nó sẽ chứa. Chúng ta thêm vào ArrayList bằng cách dùng phương thức Add(), và danh sách sẽ quan lý những đối tượng bên trong mà nó lưu giữ. Ví dụ sau minh họa sử dụng ArrayList.

Sử dụng ArrayList.

```
namespace Programming_CSharp { using System; using System.Collections; // một lớp đơn giản để lưu trữ trong mảng public class Employee { public Employee(int empID) { this.empID = empID; } public override string ToString() { return empID.ToString(); } public int EmpID { get { return empID;
{ set } empID = value; } } private int empID; } public class Tester { static void Main()
{
ArrayList empArray = new ArrayList(); ArrayList intArray = new ArrayList(); // đưa vào mảng for( int i = 0; i < 5; i++) { empArray.Add( new Employee(i+100)); intArray.Add( i*5 ); } // in tất cả nội dung for(int i = 0; i < intArray.Count; i++) { Console.Write("{0}",intArray[i].ToString());
}
Console.WriteLine("\n"); // in tất cả nội dung của mảng for(int i = 0; i < empArray.Count; i++) { Console.Write("{0} ",empArray[i].ToString());
}
Console.WriteLine("\n");
Console.WriteLine("empArray.Count: {0}", empArray.Count);
Console.WriteLine("empArray.Capacity: {0}",
empArray.Capacity); }
}
}
```

Kết quả:
0 5 10 15 20 100 101 102 103 104 empArray.Count: 5 empArray.Capacity: 16

Với lớp Array phải định nghĩa số đối tượng mà mảng sẽ lưu giữ. Nếu cố thêm các thành phần vào trong mảng vượt quá kích thước mảng thì lớp mảng sẽ phát sinh ra ngoại lệ. Với ArrayList thì không cần phải khai báo số đối tượng mà nó lưu giữ. ArrayList có một thuộc tính là Capacity, đưa ra số thành phần mà ArrayList có thể lưu trữ:
public int Capacity {virtual get; virtual set;}
Mặc định giá trị của Capacity là 16, nếu khi thêm thành phần thứ 17 vào thì Capacity tự động nhân đôi lên là 32. Nếu chúng ta thay đổi vòng lặp như sau:
for(int i = 0; i < 17; i++)
thì kết quả giống như sau:
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 empArray.Capacity: 32
Chúng ta có thể làm bằng tay để thay đổi giá trị của Capacity bằng hay lớn hơn giá trị Count. Nếu thiết lập giá trị của Capacity nhỏ hơn giá trị của Count, thì chương trình sẽ phát sinh ra ngoại lệ có kiểu như sau ArgumentOutOfRangeException.

Thực thi IComparable

Giống như tất cả những tập hợp, ArrayList cũng thực thi phương thức Sort() để cho phép chúng ta thực hiện việc sắp xếp bất cứ đối tượng nào thực thi IComparable. Trong ví dụ kế tiếp sao, chúng ta sẽ bổ sung đối tượng Employee để thực thi IComparable:
public class Employee: IComparable
Để thực thi giao diện IComparable, đối tượng Employee phải cung cấp một phương thức CompareTo(): public int CompareTo(Object o) { Employee r =
(Employee) o; return this.empID.CompareTo(r.empID); }
Phương thức CompareTo() lấy một đối tượng làm tham số, đối tượng Employee phải so sánh chính nó với đối tượng này và trả về -1 nếu nó nhỏ hơn đối tượng này, 1 nếu nó lớn hơn, và cuối cùng là giá trị 0 nếu cả hai đối tượng bằng nhau. Việc xác định thứ tự của Employee thông qua thứ tự của empID là một số nguyên. Do vậy việc so sánh sẽ được ủy quyền cho thành viên empID, đây là số nguyên và nó sẽ sử dụng phương thức so sánh mặc định của kiểu dữ liệu nguyên. Điều này tương đương với việc so sánh hai số nguyên. Lúc này chúng ta có thể thực hiện việc so sánh hai đối tượng Employee. Để thấy được cách sắp xếp, chúng ta cần thiết phải thêm vào các số nguyên vào trong mảng Employee, các số nguyên này được lấy một cách ngẫu nhiên. Để tạo một giá trị ngẫu nhiên, chúng ta cần thiết lập một đối tượng của lớp Random, lớp này sẽ trả về một số giả số ngẫu nhiên. Phương thức Next() được nạp chồng, trong đó một phiên bản cho phép chúng ta truyền vào một số nguyên thể hiện một số ngẫu nhiên lớn nhất mong muốn. Trong trường hợp này chúng ta đưa vào số 10 để tạo ra những số ngẫu nhiên từ 0 đến 10:
Random r = new Random();
r.Next(10);
Ví dụ minh họa sau tạo ra một mảng các số nguyên và một mảng Employee, sau đó đưa vào những số ngẫu nhiên, rồi in kết quả. Sau đó sắp xếp cả hai mảng và in kết quả cuối cùng.
Sắp xếp mảng số nguyên và mảng Employee.

```
namespace Programming_CSharp { using System; using System.Collections; // một lớp đơn giản để lưu trữ trong mảng public class Employee : IComparable { public Employee(int empID) { this.empID = empID;
}
public override string ToString() { return empID.ToString(); } public int EmpID { get { return empID; } set { empID = value; }
}
// So sánh được delegate cho Employee
// Employee sử dụng phương thức so sánh // mặc định của số nguyên public int CompareTo(Object o) { Employee r = (Employee) o; return this.empID.CompareTo(r.empID);
} private int empID; } public class Tester { static void Main() {
ArrayList empArray = new ArrayList();
ArrayList intArray = new ArrayList();
Random r = new Random(); // đưa vào mảng for( int i = 0; i < 5; i++) { empArray.Add( new Employee(r.Next(10)+100)); intArray.Add( r.Next(10) ); } // in tất cả nội dung for(int i = 0; i < intArray.Count; i++) { Console.Write("{0}",intArray[i].ToString());
}
Console.WriteLine("\n");
// in tất cả nội dung của mảng for(int i = 0; i < empArray.Count; i++) { Console.Write("{0} ",empArray[i].ToString());
}
Console.WriteLine("\n"); // sắp xếp và hiển thị mảng nguyên intArray.Sort(); for(int i = 0; i < intArray.Count; i++) { Console.Write("{0} ", intArray[i].ToString());
}
Console.WriteLine("\n");
// sắp xếp lại mảng
Employee empArray.Sort();
// hiển thị tất cả nội dung của mảng Employee for(int i = 0; i < empArray.Count; i++)
{ Console.Write("{0}", empArray[i].ToString());
} Console.WriteLine("\n");
}
}
}
```

Kết quả:

```
8 5 7 3 3
105 103 107 104 102
3 3 5 7 8
102 103 104 105 107
```

Kết quả chỉ ra rằng mảng số nguyên và mảng Employee được tạo ra với những số ngẫu nhiên, và sau đó chúng được sắp xếp và được hiển thị lại giá trị mới theo thứ tự sau khi sắp xếp.

Thực thi IComparer

Khi chúng ta gọi phương thức Sort() trong ArrayList thì phương thức mặc định của IComparer được gọi, nó sử dụng phương pháp QuickSort để gọi thực thi IComparable phương thức CompareTo() trong mỗi thành phần của ArrayList.

Chúng ta có thể tự do tạo một thực thi của IComparer riêng, điều này cho phép ta có thể tùy chọn cách thực hiện việc sắp xếp các thành phần trong mảng. Trong ví dụ minh họa tiếp sau đây, chúng ta sẽ thêm trường thứ hai vào trong Employee là yearsOfSvc. Và Employee có thể được sắp xếp theo hai loại là empID hoặc là yearsOfSvc.

Để thực hiện được điều này, chúng ta cần thiết phải tạo lại sự thực thi của IComparer để truyền cho phương thức Sort() của mảng ArrayList. Lớp IComparer EmployeeComparer biết về những đối tượng Employee và cũng biết cách sắp xếp chúng. EmployeeComparer có một thuộc tính, WhichComparision có kiểu là Employee.EmployeeComparer.ComparisionType:

```
public Employee.EmployeeComparer.ComparisionType WhichComparision
{
get { return whichComparision; } set { wichComparision = value; }
}
```

ComparisionType là kiểu liệt kê với hai giá trị, empID hay yearsOfSvc, hai giá trị này chỉ ra rằng chúng ta muốn sắp xếp theo ID hay số năm phục vụ:

```
public enum ComparisionType {
EmpID, Yrs
};
```

Trước khi gọi Sort(), chúng ta sẽ tạo thể hiện của EmployeeComparer và thiết lập giá trị cho thuộc tính kiểu ComparisionType:

```
Employee.EmployeeComparer c = Employee.GetComparer();
c.WhichComparision = Employee.EmployeeComparer.ComparisionType.EmpID; empArray.Sort(c);
```

Khi chúng ta gọi Sort() thì ArrayList sẽ gọi phương thức Compare() trong EmployeeComparer, đến lượt nó sẽ ủy quyền việc so sánh cho phương thức Employee.CompareTo(),và truyền vào thuộc tính WhichComparision của nó:

```
Compare(object lhs, object rhs) { Employee l = (Employee)
lhs; Employee r = (Employee) rhs; return
l.CompareTo(r.WhichComparision); }
```

Đối tượng Employee phải thực thi một phiên bản riêng của CompareTo() để thực hiện việc so sánh:

```
public int CompareTo(Employee rhs,
Employee.EmployeeComparer.ComparisionType which) { switch
(which) { case
Employee.EmployeeComparer.ComparisionType.EmpID: return
this.empID.CompareTo( rhs.empID); case
Employee.EmployeeComparer.ComparisionType.Yrs: return
this.yearsOfSvc.CompareTo(rhs.yearsOfSvc); } return 0; }
```

Sau đây là ví dụ sau thể hiện đầy đủ việc thực thi IComparer để cho phép thực hiện sắp xếp theo hai tiêu chuẩn khác nhau. Trong ví dụ này mảng số nguyên được xóa đi để làm cho đơn giản hóa ví dụ.

Sắp xếp mảng theo tiêu chuẩn ID và năm công tác.

```
----------namespace Programming_CSharp { using System; using System.Collections; //lớp đơn giản để lưu trữ trong mảng public class Employee : IComparable { public Employee(int empID) { this.empID = empID;
} public Employee(int empID, int yearsOfSvc) { this.empID = empID; this.yearsOfSvc = yearsOfSvc; } public override string ToString() {
return "ID: "+empID.ToString() + ". Years of Svc: "+
yearsOfSvc.ToString(); }
// phương thức tĩnh để nhận đối tượng Comparer public static EmployeeComparer GetComparer() { return new Employee.EmployeeComparer(); } public int CompareTo(Object rhs) { Employee r = (Employee) rhs; return this.empID.CompareTo(r.empID); }
// thực thi đặc biệt được gọi bởi custom comparer
public int CompareTo(Employee rhs, Employee.EmployeeComparer.ComparisionType which) { switch (which) { case Employee.EmployeeComparer.ComparisionType.EmpID:
return this.empID.CompareTo( rhs.empID); case Employee.EmployeeComparer.ComparisionType.Yrs: return this.yearsOfSvc.CompareTo( rhs.yearsOfSvc); } return 0; } // lớp bên trong thực thi IComparer public class EmployeeComparer : IComparer { // định nghĩa kiểu liệt kê public enum ComparisionType {
EmpID, Yrs
}; // yêu cầu những đối tượng Employee tự so sánh với nhau public int Compare( object lhs, object rhs)
{
Employee l = (Employee) lhs; Employee r = (Employee) rhs; return l.CompareTo(r, WhichComparision); }
public Employee.EmployeeComparer.ComparisionType WhichComparision { get { return whichComparision; } set { whichComparision = value; }
}
private Employee.EmployeeComparer.ComparisionType whichComparision; } private int empID; private int yearsOfSvc = 1;
}
public class Teser { static void Main() { ArrayList empArray = new ArrayList();
Random r = new Random(); // đưa vào mảng for(int i=0; i < 5; i++) { empArray.Add( new Employee(r.Next(10)+100, r.Next(20))); }
// hiển thị tất cả nội dung của mảng Employee for(int i=0; i < empArray.Count; i++)
{ Console.Write("\n{0} ", empArray[i].ToString());
}
Console.WriteLine("\n");
// sắp xếp và hiển thị mảng
Employee.EmployeeComparer c = Employee.GetComparer();
c.WhichComparision = Employee.EmployeeComparer.ComparisionType.EmpID; empArray.Sort(c);
// hiển thị nội dung của mảng for(int i=0; i < empArray.Count; i++) { Console.Write("\n{0} ", empArray[i].ToString());
}
Console.WriteLine("\n");
```

```
c.WhichComparision = Employee.EmployeeComparer.ComparisionType.Yrs; empArray.Sort(c); // hiển thị nội dung của mảng for(int i=0; i < empArray.Count; i++) { Console.Write("\n{0} ", empArray[i].ToString());
} Console.WriteLine("\n");
}
}
}
```

Kết quả:

```
ID: 100. Years of Svc: 16
ID: 102. Years of Svc: 8
ID: 107. Years of Svc: 17
ID: 105. Years of Svc: 0
ID: 101. Years of Svc: 3
ID: 100. Years of Svc: 16
ID: 101. Years of Svc: 3
ID: 102. Years of Svc: 8
ID: 105. Years of Svc: 0
ID: 107. Years of Svc: 17
ID: 105. Years of Svc: 0
ID: 101. Years of Svc: 3
ID: 102. Years of Svc: 8
ID: 100. Years of Svc: 16
ID: 107. Years of Svc: 17
```

Khối đầu tiên hiển thị kết quả thứ tự vừa nhập vào. Trong đó giá trị của empID, và yearsOfSvc được phát sinh ngẫu nhiên. Khối thứ hai hiển thị kết quả sau khi sắp theo empID,và khối cuối cùng thể hiện kết quả được xếp theo năm phục vụ.

Hàng đợi (Queue) và ngăn xếp (Stack)

Hàng đợi (Queue)

Hàng đợi là kiểu dữ liệu tốt để quản lý những nguồn tài nguyên giới hạn. Ví dụ, chúng ta muốn gởi thông điệp đến một tài nguyên mà chỉ xử lý được duy nhất một thông điệp một lần. Khi đó chúng ta sẽ thiết lập một hàng đợi thông điệp để xử lý các thông điệp theo thứ tự đưa vào.

Lớp Queue thể hiện kiểu dữ liệu như trên, trong bảng 9.4 sau liệt kê những phương thức và thuộc tính thành viên.

Những phương thức và thuộc tính của Queue	
Phương thức- thuộc Mục đích tính	
Synchronized()	Phương thức static trả về một Queue wrapper đượcthread-safe.
Count	Thuộc tính trả về số thành phần trong hàng đợi
IsReadOnly	Thuộc tính xác định hàng đợi là chỉ đọc
IsSynchronized	Thuộc tính xác định hàng đợi được đồng bộ
Thuộc tính trả về đối tượng có thể được sử dụng đểđồng bộ SyncRoot	
truy cập Queue.	

Clear()	Xóa tất cả các thành phần trong hàng đợi
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong mảng.
Sao chép những thành phần của hàng đợi đến mảngmột chiều	
CopyTo()	
đã tồn tại	
Dequeue()	Xóa và trả về thành phần bắt đầu của hàng đợi.
Enqueue()	Thêm một thành phần vào hàng đợi.
GetEnumerator()	Trả về một enumerator cho hàng đợi.
Peek()	Trả về phần tử đầu tiên của hàng đợi và không xóa nó.
ToArray()	Sao chép những thành phần qua một mảng mới

Chúng ta có thể thêm những thành phần vào trong hàng đợi với phương thức Enqueue và sau đó lấy chúng ra khỏi hàng đợi với Dequeue hay bằng sử dụng enumerator. Ví dụ sau minh họa việc sử dụng hàng đợi.

Làm việc với hàng đợi.

```
namespace Programming_CSharp { using System; using System.Collections; public class Tester { public static void Main() {
Queue intQueue = new Queue(); // đưa vào trong mảng for(int i=0; i <5; i++) { intQueue.Enqueue(i*5); }
// hiện thị hàng đợi Console.Write("intQueue values:\t");
PrintValues( intQueue);
// xóa thành phần ra khỏi hàng đợi Console.WriteLine("\nDequeue\t{0}", intQueue.Dequeue());
// hiển thị hàng đợi Console.Write("intQueue values:\t");
PrintValues(intQueue);
// xóa thành phần khỏi hàng đợi Console.WriteLine("\nDequeue\t{0}", intQueue.Dequeue());
// hiển thị hàng đợi Console.Write("intQueue values:\t");
PrintValues(intQueue); // Xem thành phần đầu tiên trong hàng đợi. Console.WriteLine("\nPeek \t{0}", intQueue.Peek());
// hiển thị hàng đợi Console.Write("intQueue values:\t"); PrintValues(intQueue); } public static void PrintValues(IEnumerable myCollection) { IEnumerator myEnumerator = myCollection.GetEnumerator();
while (myEnumerator.MoveNext()) Console.Write("{0} ", myEnumerator.Current); Console.WriteLine();
}
}
}
```

Kết quả: intQueue values: 0 5 10 15 20 Dequeue 0 intQueue values: 5 10 15 20
Dequeue 5
IntQueue values: 10 15 20
Peek 10
IntQueue values: 10 15 20

Trong ví dụ này ArrayList được thay bằng Queue, chúng ta cũng có thể Enqueue những đối tượng do ta định nghĩa. Trong trong chương trình trên đầu tiên ta đưa 5 số nguyên

vào trong hàng đợi theo tứ tự 0 5 10 15 20. Sau khi đưa vào ta lấy ra phần tử đầu tiên là 0 nên hàng đợi còn lại 4 số là 5 10 15 20, lần thứ hai ta lấy ra 5 và chỉ còn 3 phần tử trong mảng 10 15 20. Cuối cùng ta dùng phương thức Peek() là chỉ xem phần tử đầu hàng đợi chứ không xóa chúng ra khỏi hàng đợi nên kết quả cuối cùng hàng đợi vẫn còn 3 số là 10 15 20. Một điểm lưu ý là lớp Queue là một lớp có thể đếm được enumerable nên ta có thể truyền vào phương thức PrintValues với kiểu tham số khai báo IEnumerable. Việc chuyển đổi này là ngầm định. Trong phương thức PrintValues ta gọi phương thức GetEnumerator, nên nhớ rằng đây là phương thức đơn của tất cả những lớp IEnumerable. Kết quả là một đối tượng Enumerator được trả về, do đó chúng ta có thể sử dụng chúng để liệt kê tất cả những đối tượng có trong tập hợp.

Ngăn xếp (stack)

Ngăn xếp là một tập hợp mà thứ tự là vào trước ra sau hay vào sao ra trước (LIFO), tương như một chồng đĩa được xếp trong nhà hàng. Đĩa ở trên cùng tức là đĩa xếp sau thì được lấy ra trước do vậy đĩa nằm dưới đáy tức là đĩa đưa vào đầu tiên sẽ được lấy ra sau cùng.

Hai phương thức chính cho việc thêm và xóa từ stack là Push và Pop, ngoài ra ngăn xếp cũng đưa ra phương thức Peek tương tự như Peek trong hàng đợi. Bảng 9.5 sau minh họa các phương thức và thuộc tính của lớp Stack.

Phương thức và thuộc tính của lớp Stack Phương thức- thuộc	
Mục đích tính Synchronized()	Phương thức static trả về một Stack wrapperđược thread-safe.
Count	Thuộc tính trả về số thành phần trong ngăn xếp
IsReadOnly	Thuộc tính xác định ngăn xếp là chỉ đọc
IsSynchronized Thuộc tính trả về đối tượng có thể được sử dụngđể đồng bộ	Thuộc tính xác định ngăn xếp được đồng bộ
SyncRoot truy cập Stack.	
Clear()	Xóa tất cả các thành phần trong ngăn xếp
Clone()	Tạo ra một bản sao
Contains() Sao chép những thành	Xác định xem một thành phần có trong mảng. phần của ngăn xếp đếnmảng một chiều
CopyTo() đã tồn tại	
Pop()	Xóa và trả về phần tử đầu Stack
Push()	Đưa một đối tượng vào đầu ngăn xếp
GetEnumerator()	Trả về một enumerator cho ngăn xếp.
Peek() Trả về phần tử đầu tiên của ngăn xếp và khôngxóa nó. ToArray() Sao chép những thành phần qua một mảng mới	

Ba lớp ArrayList, Queue, và Stack đều chứa phương thức nạp chồng CopyTo() và ToArray() để sao chép những thành phần của chúng qua một mảng. Trong trường hợp của ngăn xếp phương thức CopyTo() sẽ chép những thành phần của chúng đến mảng một chiều đã hiện hữu, và viết chồng lên nội dung của mảng bắt đầu tại chỉ mục mà ta xác nhận. Phương thức ToArray() trả về một mảng mới với những nội dung của những thành phần trong mảng.

Sử dụng kiểu Stack.

```
namespace Programming_CSharp
{ using System; using System.Collections; // lớp đơn giản để lưu trữ public class Tester
{ static void Main() {
Stack intStack = new Stack(); // đưa vào ngăn xếp for (int i=0; i < 8; i++) {
intStack.Push(i*5); }
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xóa phần tử đầu tiên
Console.WriteLine("\nPop\t{0}", intStack.Pop()); // hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xóa tiếp phần tử khác Console.WriteLine("\nPop\t{0}", intStack.Pop());
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xem thành phần đầu tiên stack Console.WriteLine("\nPeek \t{0}", intStack.Peek());
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// khai báo mảng với 12 phần tử Array targetArray = Array.CreateInstance(typeof(int),
12); for(int i=0; i <=8; i++) { targetArray.SetValue(100*i, i); }
}
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xóa phần tử đầu tiên Console.WriteLine("\nPop\t{0}", intStack.Pop());
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xóa tiếp phần tử khác Console.WriteLine("\nPop\t{0}", intStack.Pop());
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// xem thành phần đầu tiên stack Console.WriteLine("\nPeek \t{0}", intStack.Peek());
// hiển thị stack
Console.Write("intStack values:\t");
PrintValues( intStack );
// khai báo mảng với 12 phần tử Array targetArray = Array.CreateInstance(typeof(int),
12); for(int i=0; i <=8; i++) { targetArray.SetValue(100*i, i); }
// hiển thị giá trị của mảng
Console.WriteLine("\nTarget array: ");
PrintValues( targetArray );
// chép toàn bộ stack vào mảng tại vị trí 6 intStack.CopyTo( targetArray, 6); // hiển thị
giá trị của mảng sau copy Console.WriteLine("\nTarget array after copy:");
PrintValues( targetArray );
// chép toàn bộ stack vào mảng mới
Object[] myArray = intStack.ToArray();
```

```
// hiển thị giá trị của mảng mới Console.WriteLine("\nThe new array: "); PrintValues( myArray ); } public static void PrintValues(IEnumerable myCollection) { IEnumerator myEnumerator = myCollection.GetEnumerator();
while (myEnumerator.MoveNext()) Console.Write("{0} ", myEnumerator.Current); Console.WriteLine();
}
}
}
```

Kết quả:

```
intStack values: 35 30 25 20 15 10 5 0
Pop 35 intStack values: 30 25 20 15 10 5 0 Pop 30 intStack values: 25 20 15 10 5 0 Peek 25 intStack values: 25 20 15 10 5 0
Target array:
0 100 200 300 400 500 600 700 800 0 0 0 Target array after copy:
0 100 200 300 400 500 25 20 15 10 5 0
The new array:
25 20 15 10 5 0
```

Kết quả cho thấy rằng các mục được đưa vào trong ngăn xếp và được lấy ra theo thứ tự LIFO. Trong ví dụ sử dụng lớp Array như là lớp cơ sở cho tất cả các lớp mảng. Tạo ra một mảng với 12 phần tử nguyên bằng cách gọi phương thức tĩnh CreateInstance(). Phương thức này có hai tham số một là kiểu dữ liệu trong trường hợp này là số nguyên int và tham số thứ hai thể hiện kích thước của mảng. Mảng sau đó được đưa vào bằng phương thức SetValue() phương thức này cũng lấy hai tham số là đối tượng được thêm vào và vị trí được thêm vào. Như kết quả cho ta thấy phương thức CopyTo() đã viết chồng lên giá trị của mảng từ vị trí thứ 6. Lưu ý rằng phương thức ToArray() được thiết kế để trả về một mảng đối tượng do vậy mảng myArray được khai báo tương ứng.

```
Object[] myArray = myIntStack.ToArray();
```

6. Kiểu từ điển

Để tìm thấy giá trị trong từ điển chúng ta hãy tưởng tượng rằng chúng ta muốn giữ một danh sách các thủ phủ của bang. Một hướng tiếp cận là đặt chúng vào trong một mảng theo thứ tự anphabe như sau:

```
string[] stateCapitals = new string[50];
```

Mảng stateCapital sẽ nắm giữ 50 thủ phủ bang. Mỗi thủ phủ được truy cập như một khoảng dời (offset) trong mảng.

Để truy cập thủ phủ của bang Arkansas, chúng ta cần phải biết bang Arkansas nằm ở vị trí thứ tư trong thứ tự alphabe danh sách các bang, nên ta có thể truy cập như sau:

```
string capitalOfArkansas = stateCapitals[3];
```

Tuy nhiên, thật không thuận tiện khi khi truy cập các thủ phủ của các bang thông qua cấu trúc mảng như vậy. Giả sử nếu chúng ta muốn truy cập thủ phủ của bang Massachusett, không phải dễ dàng xác định rằng thứ tự của bang là thứ 21 theo alphabe. Một cách thuận tiện hơn là lưu trữ thủ phủ theo tên của bang. Một từ điển cho phép chúng ta lưu trữ một giá trị (trong trường hợp này là thủ phủ) và với một khóa truy cập (là tên của bang). Kiểu dữ liệu từ điển trong .NET Framework có thể kết hợp bất cứ kiểu khóa nào như kiểu chuỗi, số nguyên, đối tượng...với bất cứ kiểu giá trị nào (chuỗi, số nguyên, kiểu đối tượng).

Thuộc tính quan trọng của một từ điển tốt là dễ thêm giá trị mới vào, và nhanh chóng truy cập đến giá trị. Một vài từ điển thì nhanh hơn về thời gian thêm một giá trị mới vào, một số khác thì tối ưu cho việc truy cập. Một trong minh họa cho kiểu từ điển là kiểu dữ liệu hashtable hay còn gọi là bảng băm.

Hashtables

Hashtable là một kiểu từ điển được tối ưu cho việc truy cập được nhanh. Một số các phương thức chính và các thuộc tính của kiểu dữ liệu Hashtable được trình bày trong bảng dưới.

Phương thức và thuộc tính của lớp Hashtable

Phương thức- thuộc tính	Mục đích
Phương thức static trả về một Hashtable wrapperđược thread-Synchronized() safe.	
Count	Thuộc tính trả về số thành phần trong hashtable
IsReadOnly	Thuộc tính xác định hashtable là chỉ đọc
IsSynchronized	Thuộc tính xác định hashtable được đồng bộ
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng đểđồng bộ truy cập Hastable.
Keys	Thuộc tính trả về một ICollection chứa những khóa trong hashtable.
Values	Thuộc tính trả về một ICollection chứa những giátrị trong hashtable.
Add()	Thêm một thành phần mới với khóa và giá trị xácđịnh.
Clear()	Xóa tất cả đối tượng trong hashtable.
Item()	Chỉ mục cho hastable
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong hashtable.
ContainsKey()	Xác định xem hashtable có chứa một khóa xácđịnh
CopyTo()	Sao chép những thành phần của hashtable đếnmảng một chiều đã tồn tại
GetEnumerator()	Trả về một enumerator cho hashtable.
GetObjectData()	Thực thi ISerializable và trả về dữ liệu cần thiết đểlưu trữ.
OnDeserialization	Thực thi ISerializable và phát sinh sự kiệndeserialization khi hoàn thành.
Remove()	Xóa một thành phần với khóa xác định.

Trong một Hashtable, mỗi giá trị được lưu trữ trong một vùng. Mỗi vùng được đánh số tương tự như là từng offset trong mảng. Do khóa có thể không phải là số nguyên, nên phải chuyển các khóa thành các khóa số để ánh xạ đến vùng giá trị được đánh số. Mỗi khóa phải cung cấp phương thức GetHashCode() để nhận giá trị mã hóa thành số của nó. Chúng ta nhớ rằng mọi thứ trong C# đều được dẫn xuất từ lớp object. Lớp object cung cấp một phương thức ảo là GetHashCode(), cho phép các lớp dẫn xuất tự do kế thừa hay viết lại. Việc thực thi thông thường của phương thức GetHashCode() đối với chuỗi thì đơn giản bằng cách cộng các giá trị Unicode của từng ký tự lại rồi sau đó sử dụng toán tử chia lấy dư để nhận lại một giá trị từ 0 đến số vùng được phân của hashtable. Ta không cần phải viết lại phương thức này với kiểu chuỗi.

Khi chúng ta thêm giá trị vào Hashtable thì Hashtable sẽ gọi phương thức GetHashCode() cho mỗi giá trị mà chúng ta cung cấp. Phương thức này trả về một số nguyên, xác định vùng mà giá trị được lưu trữ trong hashtable.Dĩ nhiên là có thể nhiều giá trị nhận chung một khóa tức là cùng một vùng trong hashtable, điều này gọi là sự xung đột. Có một vài cách để giải quyết sự xung đột này. Trong đó cách chung nhất và được hỗ trợ bởi CLR là cho mỗi vùng duy trì một danh sách có thứ tự các giá trị. Khi chúng ta truy cập một giá trị trong hashtable, chúng ta cung cấp một khóa. Một lần nữa Hashtable gọi phương thức GetHashCode() trên khóa và sử dụng giá trị trả về để tìm vùng tương ứng. Nếu chỉ có một giá trị thì nó sẽ trả về, nếu có nhiều hơn hai giá trị thì việc tìm kiếm nhị phân sẽ được thực hiện trên những nội dung của vùng đó. Bởi vì có một vài giá trị nên việc tìm kiếm này thực hiện thông thường là rất nhanh.
Khóa trong Hashtable có thể là kiểu dữ liệu nguyên thủy hay là các thể hiện của các kiểu dữ liệu do người dùng định nghĩa (các lớp cho người lập trình tạo ra). Những đối tượng được sử dụng làm khóa trong hashtable phải thực thi GetHashCode() và Equals(). Trong hầu hết trường hợp, chúng ta có thể sử dụng kế thừa từ Object.

Giao diện IDictionary

Hashtable là một từ điển ví nó thực thi giao diện IDictionary. IDictionary cung cấp một thuộc tính public là Item. Thuộc tính Item truy cập một giá trị thông qua một khóa xác định. Trong ngôn ngữ C# thuộc tính Item được khai báo như sau:
object this[object key] { get; set;}
Thuộc tính Item được thực thi trong ngôn ngữ C# với toán tử chỉ mục ([]). Do vậy chúng ta có thể truy cập những item trong bất cứ đối tượng từ điển bằng cú pháp giống như truy cập mảng.
Ví dụ sau minh họa việc thêm một item vào trong bảng Hashtable và sau đó truy cập lại chúng thông qua thuộc tính Item. thuộc tính Item tương như như toán tử offset.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
{ using System; using System.Collections; public class Tester { static void Main() {
// tạo và khởi tạo hashtable Hashtable hashTable = new Hashtable(); hashTable.Add("00440123","Ngoc Thao"); hashTable.Add("00123001","My Tien"); hashTable.Add("00330124","Thanh Tung"); // truy cập qua thuộc tính Item
Console.WriteLine("myHashtable[\"00440123\"]: {0}", hashTable["00440123"]);
}
}
}
---------------------------------------------------------------------------
Kết quả: hashTable[“00440123”]: Ngoc Thao
---------------------------------------------------------------------------
```

Ví dụ trên bắt đầu bằng việc tạo một bảng Hashtable mới, sử dụng các giá trị mặc định của dung lượng, phương thức tạo mã băm và phương tức so sánh. Tiếp sau là việc thêm 3 bộ giá trị vào theo thứ tự khóa/giá trị. Sau khi các item đã được thêm vào chúng ta có thể lấy giá trị thông qua khóa với cách thức dùng toán tử offset.

Tập khóa và tập giá trị

Các kiểu từ cung cấp thêm hai thuộc tính là thuộc tính Keys, và thuộc tính Values. Trong đó Keys truy cập đối tượng ICollection với tất cả những khóa trong Hashtable, và Values truy cập đối tượng ICollection với tất cả giá trị. Ví dụ minh họa như sau.
Tập khóa và tập giá trị.

```
---------------------------------------------------------------------------
namespace Progrmming_CSharp { using System; using System.Collections; public class Tester { static void Main() {
// tạo và khởi tạo hashtable Hashtable hashTable = new Hashtable(); hashTable.Add("00440123","Ngoc Thao"); hashTable.Add("00123001","My Tien"); hashTable.Add("00330124","Thanh Tung");
// nhận tập khóa
ICollection keys = hashTable.Keys;
// nhập tập giá trị ICollection values = hashTable.Values; // xuất tập khóa foreach( string s in keys) { Console.WriteLine("{0}", s);
} // xuất tập giá trị foreach( string s in values) { Console.WriteLine("{0}", s);
}
}
}
}
---------------------------------------------------------------------------
Kết quả:
00123001
00440123
00330124
My Tien
Ngoc Thao
Thanh Tung
---------------------------------------------------------------------------
```

Mặc dù thứ tự của keys không được đảm bảo theo thứ tự nhưng chúng đảm bảo rằng cùng với thứ tự đưa ra của giá trị. Như chúng ta thấy trên khóa 00123001 tương ứng với My Tien,...

Giao diện IDictionaryEnumerator

Những đối tượng IDictionary cũng hỗ trợ vòng lặp foreach bằng việc thực thi phương thức GetEnumerator(), phương thức này trả về một IDictionaryEnumerator. IDictionaryEnumerator được sử dụng để liệt kê bất cứ đối tượng IDictionary nào. Nó cung cấp thuộc tính để truy cập cả khóa và giá trị cho mỗi thành phần trong từ điển. Ta có ví dụ minh họa như sau:

Sử dụng giao diện IDictionaryEnumerator.

```
---------------------------------------------------------------------------
namespace Progrmming_CSharp { using System; using System.Collections; public class Tester { static void Main() {
// tạo và khởi tạo hashtable Hashtable hashTable = new Hashtable(); hashTable.Add("00440123","Ngoc Thao"); hashTable.Add("00123001","My Tien"); hashTable.Add("00330124","Thanh Tung"); Console.WriteLine("hashTable");
Console.WriteLine("Count: {0}",hashTable.Count); Console.WriteLine("Keys and Values:"); Print( hashTable ); } public static void Print(Hashtable table) { IDictionaryEnumerator enumerator = table.GetEnumerator(); while ( enumerator.MoveNext() ) {
Console.WriteLine("\t{0}:\t{1}", enumerator.Key, enumerator.Value); }
Console.WriteLine();
}
```

```
}
}
```
----------Kết quả: hashTableg
Count: 3
Keys and Values:
00123001: My Tien
00440123: Ngoc Thao
00330124: Thanh Tung

Bài tập:

Viết chương trình tạo mảng có 100 số nguyên dương, các số nguyên dương này là bội số của 3

Bài tập nâng cao:

Viết chương trình tạo ra 2 mảng số nguyên là A và B, có 100 phần tử, mỗi phần tử là 1 số nguyên dương có 1 chữ số, tiến hành cộng 2 mảng A và B như cách cộng số nguyên thông thường.

Những trọng tâm cần chú ý trong bài:

- Hiểu các kiến thức về mảng và danh sách mảng;
- Hiểu các kiến thức về bộ chỉ mục và tập hợp.
- Biết các kiến thức về bộ từ điển dựng sẵn trong C#.
- Giải quyết được một số bài tập trên mảng, chỉ mục và tập hợp;
- Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:
    - Hiểu các kiến thức về mảng và danh sách mảng;
    - Hiểu các kiến thức về bộ chỉ mục và tập hợp.
    - Biết các kiến thức về bộ từ điển dựng sẵn trong C#.

+ Về kỹ năng: Giải quyết được một số bài tập trên mảng, chỉ mục và tập hợp.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Giải quyết được một số bài tập trên mảng, chỉ mục và tập hợp.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

BÀI 7: XỬ LÝ CHUỖI
Mã bài: MĐ 11 - 09

Giới thiệu:
Chuỗi là thành phần không thể thiếu trong dữ liệu, việc xử lý chuỗi rất khó khăn, chúng ta cần nắm rõ các hàm liên quan để vận dụng vào bài toán

Mục tiêu:
+ Hiểu được đặc tính các lớp dựng sẵn string trong C#;
+ Sử dụng thạo các lớp có sẵn để làm các bài tập;
+ Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập.

Nội dung chính:

1. Lớp đối tượng string

Ngôn ngữ C# hỗ trợ khá đầy đủ các chức năng của kiểu chuỗi mà chúng ta có thể thấy được ở các ngôn ngữ lập trình cấp cao khác. Điều quan trọng hơn là ngôn ngữ C# xem những chuỗi như là những đối tượng và được đóng gói tất cả các thao tác, sắp xếp, và các phương thức tìm kiếm thường được áp dụng cho chuỗi ký tự.
Những thao tác chuỗi phức tạp và so khớp mẫu được hỗ trợ bởi việc sử dụng các biểu thức quy tắc (regular expression). Ngôn ngữ C# kết hợp sức mạnh và sự phức tạp của cú pháp biểu thức quy tắc, (thông thường chỉ được tìm thấy trong các ngôn ngữ thao tác chuỗi như Awk, Perl), với một thiết kế hướng đối tượng đầy đủ.
Trong chương 10 này chúng ta sẽ học cách làm việc với kiểu dữ liệu string của ngôn ngữ C#, kiểu string này chính là một alias của lớp System.String của .NET Framework. Chúng ta cũng sẽ thấy được cách rút trích ra chuỗi con, thao tác và nối các chuỗi, xây dựng một chuỗi mới với lớp StringBuilder. Thêm vào đó, chúng ta sẽ được học cách sử dụng lớp Regex để so khớp các chuỗi dựa trên biểu thức quy tắc phức tạp.
C# xem những chuỗi như là những kiểu dữ liệu cơ bản tức là các lớp này rất linh hoạt, mạnh mẽ, và nhất là dễ sử dụng. Mỗi đối tượng chuỗi là một dãy cố định các ký tự Unicode. Nói cách khác, các phương thức được dùng để làm thay đổi một chuỗi thực sự trả về một bản sao đã thay đổi, chuỗi nguyên thủy không thay đổi. Khi chúng ta khai báo một chuỗi C# bằng cách dùng từ khóa string, là chúng ta đã khai báo một đối tượng của lớp System.String, đây là một trong những kiểu dữ liệu được xây dựng sẵn được cung cấp bởi thư viện lớp .NET (.NET Framework Class Library). Do đó một kiểu dữ liệu chuỗi C# là kiểu dữ liệu System.String, và trong suốt chương này dùng hai tên hoán đổi lẫn nhau.
Khai báo của lớp System.String như sau:
public sealed class String : IComparable, ICloneble,
IConvertible
Khai báo này cho thấy lớp String đã được đóng dấu là không cho phép kế thừa, do đó chúng ta không thể dẫn xuất từ lớp này được. Lớp này cũng thực thi ba giao diện hệ thống là IComparable, ICloneable, và IConvertible – giao diện này cho phép lớp System.String chuyển đổi với những lớp khác trong hệ thống .NET.
Như chúng ta đã xem trong chương 9, giao diện IComparable được thực thi bởi các kiểu dữ liệu đã được sắp xếp. Ví dụ như chuỗi thì theo cách sắp xếp Alphabe. Bất cứ chuỗi nào đưa ra cũng có thể được so sánh với chuỗi khác để chỉ ra rằng chuỗi nào có thứ tự trước. Những lớp IComparable thực thi phương thức CompareTo().
Những đối tượng ICloneable có thể tạo ra những thể hiện khác với cùng giá trị như là thể hiện nguyên thuỷ. Do đó ta có thể tạo ra một chuỗi mới từ chuỗi ban đầu và giá trị

của chuỗi mới bằng với chuỗi ban đầu. Những lớp ICloneable thực thi phương thức Clone().
Những lớp IConvertible cung cấp phương thức để dễ dàng chuyển đổi qua các kiểu dữ liệu cơ bản khác như là ToInt32(), ToDouble(), ToDecimal(),...

1.1. Tạo một chuỗi

Cách phổ biến nhất để tạo ra một chuỗi là gán cho một chuỗi trích dẫn tức là chuỗi nằm trong dấu ngoặc kép, kiểu chuỗi này cũng được biết như là một chuỗi hằng, khai báo như sau:
string newString = "Day la chuoi hang";
Những chuỗi trích dẫn có thể được thêm các ký tự escape, như là “\n” hay “\t”, ký tự này bắt đầu với dấu chéo ngược (“\”), các ký tự này được dùng để chỉ ra rằng tại vị trí đó xuống dòng hay tab được xuất hiện. Bởi vì dấu gạch chéo ngược này cũng được dùng trong vài cú pháp dòng lệnh, như là địa chỉ URLs hay đường dẫn thư mục, do đó trong chuỗi trích dẫn dấu chéo ngược này phải được đặt trước dấu chéo ngược khác, tức là dùng hai dấu chéo ngược trong trường hợp này.
Chuỗi cũng có thể được tạo bằng cách sử dụng chuỗi cố định hay nguyên văn (verbatim), tức là các ký tự trong chuỗi được giữ nguyên không thay đổi. Chuỗi này được bắt đầu với biểu tượng @. Biểu tượng này bảo với hàm khởi dựng của lớp String rằng chuỗi theo sau là nguyên văn, thậm chí nó chứa nhiều dòng hoặc bao gồm những ký tự escape. Trong chuỗi nguyên văn, ký tự chéo ngược và những ký tự sau nó đơn giản là những ký tự được thêm vào chuỗi. Do vậy, ta có 2 định nghĩa chuỗi sau là tương đương với nhau:
string literal1 = "\\\\MyDocs\\CSharp\\ProgrammingC#.cs"; string verbatim1 = @"\\MyDocs\CSharp\ProgrammingC#.cs";
Trong chuỗi thứ nhất, là một chuỗi bình thường được sử dụng, do đó dấu ký tự chéo là ký tự escape, nên nó phải được đặt trước một ký tự chéo ngược thứ hai. Trong khai báo thứ hai chuỗi nguyên văn được sử dụng, nên không cần phải thêm ký tự chéo ngược.
Một ví dụ thứ hai minh họa việc dùng chuỗi nguyên văn:
string literal2 = "Dong mot \n dong hai";
string verbatim2 = @"Dong mot dong hai";
Nói chung ta ta có thể sử dụng qua lại giữa hai cách định nghĩa trên. Việc lựa chọn phụ thuộc vào sự thuận tiện trong từng trường hợp hay phong cách riêng của mỗi người.

1.2. Tạo một chuỗi dùng phương thức ToString

Một cách rất phổ biến khác để tạo một chuỗi là gọi phương thức ToString() của một đối tượng và gán kết quả đến một biến chuỗi. Tất cả các kiểu dữ liệu cơ bản phủ quyết phương thức này rất đơn giản là chuyển đổi giá trị (thông thường là giá trị số) đến một chuỗi thể hiện của giá trị. Trong ví dụ theo sau, phương thức ToString() của kiểu dữ liệu int được gọi để lưu trữ giá trị của nó trong một chuỗi:
int myInt = "9"; string intString = myInt.ToString();
Phương thức myInt.ToString() trả về một đối tượng String và đối tượng này được gán cho intString.
Lớp String của .NET cung cấp rất nhiều bộ khởi dựng hỗ trợ rất nhiều kỹ thuật khác nhau để gán những giá trị chuỗi đến kiểu dữ liệu chuỗi. Một vài bộ khởi dựng có thể cho phép chúng ta tạo một chuỗi bằng cách truyền vào một mảng ký tự hoặc một con trỏ ký tự. Truyền một mảng chuỗi như là tham số đến bộ khởi dựng của String là tạo ra một thể hiện CLR- compliant (một thể hiện đúng theo yêu cầu của CLR). Còn việc truyền một con trỏ chuỗi như một tham số của bộ khởi dựng String là việc tạo một thể hiện không an toàn (unsafe).

1.3. Thao tác trên chuỗi

Lớp string cung cấp rất nhiều số lượng các phương thức để so sánh, tìm kiếm và thao tác trên chuỗi, các phương thức này được trình bày trong bảng 10.1:

Phương thức và thuộc tính của lớp String

System.String	
Phương thức/ Trường	Ý nghĩa
Empty	Trường public static thể hiện một chuỗi rỗng.
Compare()	Phương thức public static để so sánh hai chuỗi.
CompareOrdinal()	Phương thức public static để so sánh hai chuỗi khôngquan tâm đến thứ tự.
Concat()	Phương thức public static để tạo chuỗi mới từ một haynhiều chuỗi.
Copy()	Phương thức public static tạo ra một chuỗi mới bằngsao từ chuỗi khác.
Equal()	Phương thức public static kiểm tra xem hai chuỗi cócùng giá trị hay không.
Format()	Phương thức public static định dạng một chuỗi dùngký tự lệnh định dạng xác định.
Intern()	Phương thức public static trả về tham chiếu đến thểhiện của chuỗi.
IsInterned()	Phương thức public static trả về tham chiếu của chuỗi
Join()	Phương thức public static kết nối các chuỗi xác địnhgiữa mỗi thành phần của mảng chuỗi.
Chars()	Indexer của chuỗi.
Length()	Chiều dài của chuỗi.
Clone()	Trả về chuỗi.
CompareTo()	So sánh hai chuỗi.
CopyTo()	Sao chép một số các ký tự xác định đến một mảng kýtự Unicode.
EndsWidth()	Chỉ ra vị trí của chuỗi xác định phù hợp với chuỗi đưara.
Insert()	Trả về chuỗi mới đã được chèn một chuỗi xác định.
LastIndexOf()	Chỉ ra vị trí xuất hiện cuối cùng của một chuỗi xácđịnh trong chuỗi.
PadLeft()	Canh lề phải những ký tự trong chuỗi, chèn vào bêntrái khoảng trắng hay các ký tự xác định.
PadRight()	Canh lề trái những ký tự trong chuỗi, chèn vào bênphải khoảng trắng hay các ký tự xác định.
Remove()	Xóa đi một số ký tự xác định.
Split()	Trả về chuỗi được phân định bởi những ký tự xác địnhtrong chuỗi.
StartWidth()	Xem chuỗi có bắt đầu bằng một số ký tự xác định haykhông.
SubString()	Lấy một chuỗi con.
ToCharArray()	Sao chép những ký tự từ một chuỗi đến mảng ký tự.
ToLower()	Trả về bản sao của chuỗi ở kiểu chữ thường.

ToUpper() Trả về bản sao của chuỗi ở kiểu chữ hoa.
Xóa bỏ tất cả sự xuất hiện của tập hợp ký tự xác địnhtừ vị trí đầu
Trim()
tiên đến vị trí cuối cùng trong chuỗi.
TrimEnd() Xóa như nhưng ở vị trí cuối.
TrimStart() Xóa như Trim nhưng ở vị trí đầu.
Trong ví dụ sau đây chúng ta minh họa việc sử dụng một số các phương thức của chuỗi như Compare(), Concat() (và dùng toán tử +), Copy() (và dùng toán tử =), Insert(), EndsWith(), và chỉ mục IndexOf.
Làm việc với chuỗi.
---
namespace Programming_CSharp { using System; public class StringTester { static void Main() { // khởi tạo một số chuỗi để thao tác string s1 = "abcd"; string s2 = "ABCD"; string s3 = @"Trung Tam Dao Tao CNTT Thanh pho Ho Chi Minh
Viet Nam"; int result;
// So sánh hai chuỗi với nhau có phân biệt chữ thường và
chữ hoa result = string.Compare(s1 ,s2);
Console.WriteLine("So sanh hai chuoi S1: {0} và S2: {1} ket qua: {2} \n", s1 ,s2 ,result);
// Sử dụng tiếp phương thức Compare() nhưng trường hợp này không biệt // chữ thường hay chữ hoa
// Tham số thứ ba là true sẽ bỏ qua kiểm tra ký tự thường-
hoa result = string. Compare(s1, s2, true); Console.WriteLine("Khong phan biet chu thuong va hoa\n");
Console.WriteLine("S1: {0} , S2: {1}, ket qua : {2}\n", s1, s2, result); // phương thức nối các chuỗi string s4 = string.Concat(s1, s2);
Console.WriteLine("Chuoi S4 noi tu chuoi S1 va S2: {0}", s4); // sử dụng nạp chồng toán tử + string s5 = s1 + s2;
Console.WriteLine("Chuoi S5 duoc noi tu chuoi S1 va S2:
{0}", s5);
// Sử dụng phương thức copy chuỗi string
s6 = string.Copy(s5); Console.WriteLine("S6 duoc sao chep tu S5: {0}", s6); // Sử dụng nạp chồng toán tử = string s7 = s6; Console.WriteLine(“S7 = S6: {0}”, s7);
// Sử dụng ba cách so sánh hai chuỗi
// Cách 1 sử dụng một chuỗi để so sánh với chuỗi còn lại
Console.WriteLine("S6.Equals(S7) ?: {0}", s6.Equals(s7));
// Cách 2 dùng hàm của lớp string so sánh hai chuỗi
Console.WriteLine("Equals(S6, s7) ?: {0}", string.Equals(s6, s7)); // Cách 3 dùng toán tử so sánh
Console.WriteLine("S6 == S7 ?: {0}", s6 == s7);
// Sử dụng hai thuộc tính hay dùng là chỉ mục và chiều dài
của chuỗi
Console.WriteLine("\nChuoi S7 co chieu dai la : {0}",
s7.Length);
Console.WriteLine("Ky tu thu 3 cua chuoi S7 la : {0}", s7[2]
); // Kiểm tra xem một chuỗi có kết thúc với một nhóm ký
// tự xác định hay không
Console.WriteLine("S3: {0}\n ket thuc voi chu CNTT ? :

```
{1}\n", s3, s3.EndsWith("CNTT"));
Console.WriteLine("S3: {0}\n ket thuc voi chu Nam ? :
{1}\n", s3, s3.EndsWith("Nam"));
// Trả về chỉ mục của một chuỗi con
Console.WriteLine("\nTim vi tri xuat hien dau tien cua chu CNTT ");

Console.WriteLine("trong chuoi S3 là {0}\n)";
s3.IndexOf("CNTT")); // Chèn từ nhân lực vào trước CNTT trong chuỗi S3 string s8 =
s3.Insert(18, "nhan luc"); Console.WriteLine(" S8 : {0}\n", s8);
// Ngoài ra ta có thể kết hợp như sau,
string s9 = s3.Insert( s3.IndexOf( "CNTT" ) , "nhan luc ");
Console.WriteLine(" S9 : {0}\n", s9);
} // end Main
} // end class
} // end namespace
```

Kết quả:

```
So sanh hai chuoi S1: abcd và S2: ABCD ket qua: -1
Khong phan biet chu thuong va hoa
S1: abcd , S2: ABCD, ket qua : 0
Chuoi S4 noi tu chuoi S1 va S2: abcdABCD
Chuoi S5 duoc noi tu chuoi S1 + S2: abcdABCD S6 duoc sao chep tu S5: abcdABCD
S7 = S6: abcdABCD S6.Equals(S7) ?: True Equals(S6, s7) ?: True S6 == S7 ?: True
Chuoi S7 co chieu dai la : 8
Ky tu thu 3 cua chuoi S7 la : c
S3: Trung Tam Dao Tao CNTT
Thanh pho Ho Chi Minh Viet Nam ket thuc voi chu CNTT ? : False
S3: Trung Tam Dao Tao CNTT
Thanh pho Ho Chi Minh Viet Nam ket thuc voi chu Minh ? : True Tim vi tri xuat hien
dau tien cua chu CNTT trong chuoi S3 là 18
S8 : Trung Tam Dao Tao nhan luc CNTT Thanh pho Ho Chi Minh Viet Nam
S9 : Trung Tam Dao Tao nhan luc CNTT Thanh pho Ho Chi Minh Viet Nam
```

Như chúng ta đã xem đoạn chương trình minh họa trên, chương trình bắt đầu với ba khai báo chuỗi:

```
string s1 = "abcd"; string s2 = "ABCD";
string s3 = @"Trung Tam Dao Tao CNTT Thanh pho Ho Chi Minh
Viet Nam";
```

Hai chuỗi đầu s1 và s2 được khai báo chuỗi ký tự bình thường, còn chuỗi thứ ba được khai báo là chuỗi nguyên văn (verbatim string) bằng cách sử dụng ký hiệu @ trước chuỗi. Chương trình bắt đầu bằng việc so sánh hai chuỗi s1 và s2. Phương thức Compare() là phương thức tĩnh của lớp string, và phương thức này đã được nạp chồng. Phiên bản đầu tiên của phương thức nạp chồng này là lấy hai chuỗi và so sánh chúng với nhau:

```
// So sánh hai chuỗi với nhau có phân biệt chữ thường và
chữ hoa result = string.Compare( s1 ,s2);
Console.WriteLine("So sanh hai chuoi s1: {0} và s2: {1} ket qua: {2} \n", s1 ,s2 ,result);
```

Ở đây việc so sánh có phân biệt chữ thường và chữ hoa, phương thức trả về các giá trị khác nhau phụ thuộc vào kết quả so sánh:
Một số âm nếu chuỗi đầu tiên nhỏ hơn chuỗi thứ hai
Giá trị 0 nếu hai chuỗi bằng nhau
Một số dương nếu chuỗi thứ nhất lớn hơn chuỗi thứ hai.
Trong trường hợp so sánh trên thì đưa ra kết quả là chuỗi s1 nhỏ hơn chuỗi s2. Trong Unicode cũng như ASCII thì thứ tự của ký tự thường nhỏ hơn thứ tự của ký tự hoa:
So sanh hai chuoi S1: abcd và S2: ABCD ket qua: -1
Cách so sánh thứ hai dùng phiên bản nạp chồng Compare() lấy ba tham số. Tham số Boolean quyết định bỏ qua hay không bỏ qua việc so sánh phân biệt chữ thường và chữ hoa. Tham số này có thể bỏ qua. Nếu giá trị của tham số là true thì việc so sánh sẽ bỏ qua sự phân biệt chữ thường và chữ hoa. Việc so sánh sau sẽ không quan tâm đến kiểu loại chữ:

```
// Tham số thứ ba là true sẽ bỏ qua kiểm tra ký tự thường – hoa result = string.
Compare(s1, s2, true);
Console.WriteLine("Khong phan biet chu thuong va hoa\n");
Console.WriteLine("S1: {0} , S2: {1}, ket qua : {2}\n", s1, s2, result);
```

Lúc này thì việc so sánh hoàn toàn giống nhau và kết quả trả về là giá trị 0:

```
Khong phan biet chu thuong va hoa
S1: abcd , S2: ABCD, ket qua : 0
```

Ví dụ minh họa trên tiếp tục với việc nối các chuỗi lại với nhau. Ở đây sử dụng hai cách để nối liền hai chuỗi. Chúng ta có thể sử dụng phương thức Concat() đây là phương thức public static của string:

```
string s4 = string.Concat(s1, s2);
```

Hay cách khác đơn giản hơn là việc sử dụng toán tử nối hai chuỗi (+):

```
string s5 = s1 + s2;
```

Trong cả hai trường hợp thì kết quả nối hai chuỗi hoàn toàn thành công và như sau:

```
Chuoi S4 noi tu chuoi S1 va S2: abcdABCD
Chuoi S5 duoc noi tu chuoi S1 + S2: abcdABCD
```

Tương tự như vậy, việc tạo một chuỗi mới có thể được thiết lập bằng hai cách. Đầu tiên là chúng ta có thể sử dụng phương thức static Copy() như sau:

```
string s6 = string.Copy(s5);
```

Hoặc thuận tiện hơn chúng ta có thể sử dụng phương thức nạp chồng toán tử (=) thông qua việc sao chép ngầm định:

```
string s7 = s6;
```

Kết quả của hai cách tạo trên đều hoàn toàn như nhau:

```
S6 duoc sao chep tu S5: abcdABCD S7 = S6: abcdABCD
```

Lớp String của .NET cung cấp ba cách để kiểm tra bằng nhau giữa hai chuỗi. Đầu tiên là chúng ta có thể sử dụng phương thức nạp chồng Equals() để kiểm tra trực tiếp rằng S6 có bằng S7 hay không:

```
Console.WriteLine("S6.Equals(S7) ?: {0}", S6.Equals(S7));
```

Kỹ thuật so sánh thứ hai là truyền cả hai chuỗi vào phương thức Equals() của string:

```
Console.WriteLine("Equals(S6, s7) ?: {0}", string.Equals(S6, S7));
```

Và phương pháp cuối cùng là sử dụng nạp chồng toán tử so sánh (=) của String:

```
Console.WriteLine("S6 == S7 ?: {0}", s6 == s7);
```

Trong cả ba trường hợp thì kết quả trả về là một giá trị Boolean, ta có kết quả như sau:

```
S6.Equals(S7) ?: True Equals(S6, s7) ?: True S6 == S7 ?:
```

True
Việc so sánh bằng nhau giữa hai chuỗi là việc rất tự nhiên và thường được sử dụng. Tuy nhiên, trong một số ngôn ngữ, như VB.NET, không hỗ trợ nạp chồng toán tử. Do đó để chắc chắn chúng ta nên sử dụng phương thức Equals() là tốt nhất.
Các đoạn chương trình tiếp theo của ví dụ 10.1 sử dụng toán tử chỉ mục ([]) để tìm ra ký tự xác định trong một chuỗi. Và dùng thuộc tính Length để lấy về chiều dài của toàn bộ một chuỗi:
Console.WriteLine("\nChuoi S7 co chieu dai la : {0}", s7.Length);
Console.WriteLine("Ky tu thu 3 cua chuoi S7 la : {0}", s7[2]);
Kết quả là:
Chuoi S7 co chieu dai la : 8
Ky tu thu 3 cua chuoi S7 la : c
Phương thức EndsWith() hỏi xem một chuỗi có chứa một chuỗi con ở vị trí cuối cùng hay không. Do vậy, chúng ta có thể hỏi rằng chuỗi S3 có kết thúc bằng chuỗi "CNTT" hay chuỗi "Nam":
// Kiểm tra xem một chuỗi có kết thúc với một nhóm ký tự xác định hay không
Console.WriteLine("S3: {0}\n ket thuc voi chu CNTT ? :
{1}\n", s3, s3.EndsWith("CNTT"));
Console.WriteLine("S3: {0}\n ket thuc voi chu Nam ? :
{1}\n", s3, s3.EndsWith("Nam"));
Kết quả trả về là lần kiểm tra đầu tiên là sai do chuỗi S3 không kết thúc chữ "CNTT", và lần kiểm tra thứ hai là đúng:
S3: Trung Tam Dao Tao CNTT
Thanh pho Ho Chi Minh Viet Nam ket thuc voi chu CNTT ? : False
S3: Trung Tam Dao Tao CNTT
Thanh pho Ho Chi Minh Viet Nam ket thuc voi chu Minh ? : True
Phương thức IndexOf() chỉ ra vị trí của một con bên trong một chuỗi (nếu có). Và phương thức Insert() chèn một chuỗi con mới vào một bản sao chép của chuỗi ban đầu. Đoạn lệnh tiếp theo của ví dụ minh họa thực hiện việc xác định vị trí xuất hiện đầu tiên của chuỗi "CNTT" trong chuỗi S3:
Console.WriteLine("\nTim vi tri xuat hien dau tien cua chu CNTT ");
Console.WriteLine("trong chuoi S3 là {0}\n", s3.IndexOf("CNTT"));
Và kết quả tìm được là 18:
Tim vi tri xuat hien dau tien cua chu CNTT trong chuoi S3 là 18
Chúng ta có thể chèn vào chuỗi từ "nhan luc" và theo sau chuỗi này là một khoảng trắng vào trong chuỗi ban đầu. Khi thực hiện thì phương thức trả về bản sao của chuỗi đã được chèn vào chuỗi con mới và được gán lại vào chuỗi S8:
string s8 = s3.Insert(18, "nhan luc "); Console.WriteLine(" S8 : {0}\n", s8);
Kết quả đưa ra là:
S8 : Trung Tam Dao Tao nhan luc CNTT Thanh pho Ho Chi Minh Viet Nam
Cuối cùng, chúng ta có thể kết hợp một số các phép toán để thực hiện việc chèn như sau:
string s9 = s3.Insert(s3.IndexOf("CNTT") , "nhan luc ");
Console.WriteLine(" S9 : {0}\n", s9);
Kết quả cuối cùng cũng tương tự như cách chèn bên trên:
S9 : Trung Tam Dao Tao nhan luc CNTT Thanh pho Ho Chi Minh Viet Nam
1.4. Tìm một chuỗi con

Trong kiểu dữ liệu String có cung cấp phương thức Substring() để trích một chuỗi con từ chuỗi ban đầu. Cả hai phiên bản đều dùng một chỉ mục để xác định vị trí bắt đầu trích ra. Và một trong hai phiên bản dùng chỉ mục thứ hai làm vị trí kết thúc của chuỗi. Trong ví dụ sau minh họa việc sử dụng phương thức Substring() của chuỗi.
Sử dụng phương thức Substring().

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; using System.Text; public class StringTester { static void Main() { // Khai báo các chuỗi để sử dụng string s1 = "Mot hai ba bon"; int ix; // lấy chỉ số của khoảng trắng cuối cùng ix = s1.LastIndexOf(" "); // lấy từ cuối cùng string s2 = s1.Substring( ix+1); // thiết lập lại chuỗi s1 từ vị trí 0 đến vị trí ix // do đó s1 có giá trị mới là mot hai ba s1 = s1.Substring(0, ix); // tìm chỉ số của khoảng trắng cuối cùng (sau hai) ix = s1.LastIndexOf(" "); // thiết lập s3 là chuỗi con bắt đầu từ vị trí ix
// do đó s3 = "ba"
string s3 = s1.Substring(ix+1); // thiết lập lại s1 bắt đầu từ vị trí 0 đến cuối vị trí ix // s1 = "mot hai" s1 = s1.Substring(0, ix); // ix chỉ đến khoảng trắng giữa "mot" va "hai" ix = s1.LastIndexOf(" "); // tạo ra s4 là chuỗi con bắt đầu từ sau vị trí ix, do // vậy có giá trị là "hai" string s4 = s1.Substring( ix+1); // thiết lập lại giá trị của s1 s1 = s1.Substring(0, ix); // lấy chỉ số của khoảng trắng cuối cùng, lúc này ix là -1 ix = s1.LastIndexOf(" ");
// tạo ra chuỗi s5 bắt đầu từ chỉ số khoảng trắng, nhưng
không có khoảng // trắng và ix là -1 nên chuỗi bắt đầu từ 0 string s5 = s1.Substring(ix +1); Console.WriteLine("s2 : {0}\n s3 : {1}", s2, s3); Console.WriteLine("s4 : {0}\n s5 : {1}\n", s4, s5); Console.WriteLine("s1: {0}\n", s1);
}// end Main
}// end class
}// end namespace
---------------------------------------------------------------------------
Kết quả: s2 : bon s3 : ba s4 : hai s5 : mot s1 : mot
---------------------------------------------------------------------------
```

Ví dụ minh họa trên không phải là giải pháp tốt để giải quyết vấn đề trích lấy các ký tự trong một chuỗi. Nhưng nó là cách gần đúng tốt nhất và minh họa hữu dụng cho kỹ thuật này.

1.5. Chia chuỗi

Một giải pháp giải quyết hiệu quả hơn để minh họa cho ví dụ 10.2 là có thể sử dụng phương thức Split() của lớp string. Chức năng chính là phân tích một chuỗi ra thành các chuỗi con.
Để sử dụng Split(), chúng ta truyền vào một mảng các ký tự phân cách, các ký tự này được dùng để chia các từ trong chuỗi. Và phương thức sẽ trả về một mảng những chuỗi con.
Sử dụng phương thức Split().

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; using System.Text;
public class StringTester { static void Main() { // tạo các chuỗi để làm việc string s1 = "Mot, hai, ba Trung Tam Dao Tao CNTT"; // tạo ra hằng ký tự khoảng trắng và dấu phẩy const char Space = ‘ ‘; const char Comma = ‘,’; // tạo ra mảng phân cách char[] delimiters = new char[] {
```

```
Space, Comma }; string output = " "; int ctr = 1; // thực hiện việc chia một chuỗi dùng
vòng lặp // đưa kết quả vào mảng các chuỗi foreach ( string subString in
s1.Split(delimiters) ) { output += ctr++; output += ": ";
output += subString; output += "\n"; }// end foreach Console.WriteLine( output );
}// end Main
}// end class
} // end namespace
```

Kết quả:

```
1: Mot 2: 3: hai 4:
5: ba
6: Trung
7: Tam
8: Dao
9: Tao
10: CNTT
```

Đoạn chương trình bắt đầu bằng việc tạo một chuỗi để minh họa việc phân tích:

```
string s1 = "Mot, hai, ba Trung Tam Dao Tao CNTT";
```

Hai ký tự khoảng trắng và dấu phẩy được dùng làm các ký tự phân cách. Sau đó phương thức Split() được gọi trong chuỗi này, và truyền kết quả vào mỗi vòng lặp:

```
foreach ( string subString in s1.Split(delimiters) )
```

Chuỗi output chứa các chuỗi kết quả được khởi tạo là chuỗi rỗng. Ở đây chúng ta tạo ra chuỗi output bằng bốn bước. Đầu tiên là nối giá trị của biến đếm ctr, tiếp theo là thêm dấu hai chấm, rồi đưa chuỗi được chia ra từ chuỗi ban đầu, và cuối cùng là thêm ký tự qua dòng mới.

Và bốn bước trên cứ được lặp đến khi nào chuỗi không còn chia ra được.

Có một vấn đề cần nói là kiểu dữ liệu string không được thiết kế cho việc thêm vào một chuỗi định dạng sẵn để tạo ra một chuỗi mới trong mỗi vòng lặp trên, nên chúng ta mới phải thêm vào từng ký tự như vậy. Một lớp StringBuilder được tạo ra để phục vụ cho nhu cầu thao tác chuỗi tốt hơn.

1.6. Thao tác trên chuỗi dùng StringBuilder

Lớp StringBuilder được sử dụng để tạo ra và bổ sung các chuỗi. Hay có thể nói lớp này chính là phần đóng gói của một bộ khởi dựng cho một String. Một số thành viên quan trọng StringBuilder được tóm tắt trong bảng 10.2 như sau:

Phương thức của lớp StringBuilder
System.StringBuilder

Phương thức	Ý nghĩa
Capacity()	Truy cập hay gán một số ký tự mà StringBuilder nắm giữ.
Chars()	Chỉ mục.
Length()	Thiết lập hay truy cập chiều dài của chuỗi
MaxCapacity()	Truy cập dung lượng lớn nhất của StringBuilder
Append()	Nối một kiểu đối tượng vào cuối của StringBuilder
AppendFormat()	Thay thế định dạng xác định bằng giá trị được định dạngcủa một đối tượng.
EnsureCapacity()	Đảm bảo rằng StringBuilder hiện thời có khả năng tối

thiểulớn như một giá trị xác định.
Insert() Chèn một đối tượng vào một vị trí xác định
Thay thế tất cả thể hiện của một ký tự xác định với nhữngký
Replace() tự mới.
Không giống như String, StringBuilder thì dễ thay đổi. Khi chúng ta bổ sung một đối tượng StringBuilder thì chúng ta đã làm thay đổi trên giá trị thật của chuỗi, chứ không phải trên bản sao. Ví dụ minh họa sau thay thế đối tượng String bằng một đối tượng StringBuilder.
Sử dụng chuỗi StringBuilder.
---
namespace Programming_CSharp { using System; using System.Text; public class StringTester { static void Main() { // khởi tạo chuỗi để sử dụng string s1 = "Mot, hai, ba Trung Tam Dao Tao CNTT"; // tạo ra hằng ký tự khoảng trắng và dấu phẩy const char Space = ‘ ‘; const char Comma = ‘,’; // tạo ra mảng phân cách char[] delimiters = new char[]
{
Space, Comma
}; // sử dụng StringBuilder để tạo chuỗi output StringBuilder output = new StringBuilder(); int ctr = 1; // chia chuỗi và dùng vòng lặp để đưa kết quả vào // mảng các chuỗi foreach (string subString in s1.Split(delimiters)) { // AppendFormat nối một chuỗi được định dạng output.AppendFormat("{0}: {1}\n", ctr++, subString); }// end foreach Console.WriteLine(output);
}
}
}

Chúng ta chỉ thay phần cuối của đoạn chương trình. Rõ ràng việc sử dụng StringBuilder thuận tiện hơn là việc sử dụng các toán tử bổ sung trong chuỗi. Ở đây chúng ta sử dụng phương thức AppendFormat() của StringBuilder để nối thêm một chuỗi được định dạng để tạo ra một chuỗi mới. Điều này quá dễ dàng và khá là hiệu quả. Kết quả chương trình thực hiện cũng tượng tự như ví dụ minh họa dùng String:
1: Mot
2:
3: hai 4:
5: ba
6: Trung
7: Tam
8: Dao
9: Tao
10: CNTT

2. Các biểu thức quy tắc

Kết quả của việc áp dụng một biểu thức qui tắc đến một chuỗi là trả về một chuỗi con hoặc là trả về một chuỗi mới có thể được bổ sung từ một vài phần của chuỗi nguyên thủy ban đầu. Chúng ta nên nhớ rằng string là không thể thay đổi được và do đó cũng không thể thay đổi bởi biểu thức qui tắc.
Bằng cách áp dụng chính xác biểu thức qui tắc cho chuỗi sau: Mot, hai, ba, Trung Tam Dao Tao CNTT

chúng ta có thể trả về bất cứ hay tất cả danh sách các chuỗi con (Mot, hai,...) và có thể tạo ra các phiên bản chuỗi được bổ sung của những chuỗi con (như : TrUng TAM,...).
Biểu thức qui tắc này được quyết định bởi cú pháp các ký tự qui tắc của chính bản thân nó. Một biểu thức qui tắc bao gồm hai kiểu ký tự:
Ký tự bình thường (literal): những ký tự này mà chúng ta sử dụng để so khớp với chuỗi ký tự đích.
Metacharacter: là các biểu tượng đặc biệt, có hành động như là các lệnh trong bộ phân tích (parser) của biểu thức.
Bộ phân tích là một cơ chế có trách nhiệm hiểu được các biểu thức qui tắc. Ví dụ nếu như chúng ta tạo một biểu thức qui tắc như sau:
^(From|To|Subject|Date):
Biểu thức này sẽ so khớp với bất cứ chuỗi con nào với những từ như “From”, “To”, “Subject”, và “Date” miễn là những từ này bắt đầu bằng ký tự dòng mới (^) và kết thúc với dấu hai chấm (:).
Ký hiệu dấu mũ (^) trong trường hợp này chỉ ra cho bộ phân tích biểu thức qui tắc rằng chuỗi mà chúng ta muốn tìm kiếm phải bắt đầu từ dòng mới. Trong biểu thức này các ký tự như “(”,”)”, và “|” là các metacharacter dùng để nhóm các chuỗi ký tự bình thường như “From”, “To”,”Subject”, và “Date” và chỉ ra rằng bất cứ sự lựa chọn nào trong số đó đều được so khớp đúng. Ngoài ra ký tự “^” cũng là ký tự metacharacter chỉ ra bắt đầu dòng mới.
Tóm lại với chuỗi biểu thức qui tắc như:
^(From|To|Subject|Date):
ta có thể phát biểu theo ngôn ngữ tự nhiên như sau: “Phù hợp với bất cứ chuỗi nào bắt đầu bằng một dòng mới được theo sau bởi một trong bốn chữ From, To, Subject, Date và theo sau là ký tự dấu hai chấm”.
Việc trình bày đầy đủ về biểu thức quy tắc vượt quá phạm vi của cuốn sách này, do sự đa dạng và khá phức tạp của nó. Tuy nhiên, trong phạm vi trình bày của chương 10 này, chúng ta sẽ được tìm hiểu một số các thao tác phổ biến và hữu dụng của biểu thức quy tắc.

Sử dụng biểu thức quy tắc qua lớp Regex

MS.NET cung cấp một hướng tiếp cận hướng đối tượng (object- oriented approad) cho biểu thức quy tắc để so khớp, tìm kiếm và thay thế chuỗi. Biểu thức quy tắc của ngôn ngữ C# là được xây dựng từ lớp regexp của ngôn ngữ Perl5.
Namspace System.Text.RegularExpressions của thư viện BCL (Base Class Library) chứa đựng tất cả các đối tượng liên quan đến biểu thức quy tắc trong môi trường .NET. Và lớp quan trọng nhất mà biểu thức quy tắc hỗ trợ là Regex. Ta có thể tạo thể hiện của lớp Regex và sử dụng một số phương thức tĩnh trong ví dụ minh họa.
Sử dụng lớp Regex.

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; using System.Text; using System.Text.RegularExpressions; public class Tester { static void Main() {
// khởi tạo chuỗi sử dụng
string s1 = "Mot, hai, ba, Trung Tam Dao Tao CNTT"; // tạo chuỗi biểu thức quy tắc
Regex theRegex = new Regex(" |, "); StringBuilder sBuilder = new StringBuilder(); int id = 1; // sử dụng vòng lặp để lấy các chuỗi con foreach ( string subString in theRegex.Split(s1)) {
```

// nối chuỗi vừa tìm được trong biểu thức quy tắc // vào chuỗi StringBuilder theo định dạng sẵn.
sBuilder.AppendFormat("{0}: {1} \n", id++, subString); } Console.WriteLine("{0}", sBuilder);
}// end Main
}// end class
}// end namespace

Kết quả:
1: Mot
2: hai
3: ba
4: Trung
5: Tam
6: Dao
7: Tao
8: CNTT

Ví dụ minh họa bắt đầu bằng việc tạo một chuỗi s1, nội dung của chuỗi này tương tự như chuỗi trong minh họa trên.
string s1 = "Mot, hai, ba, Trung Tam Dao Tao CNTT";
Tếp theo một biểu thức quy tắc được tạo ra, biểu thức này được dùng để tìm kiếm một chuỗi:
Regex theRegex = new Regex(" |, ");
Ở đây một bộ khởi tạo nạp chồng của Regex lấy một chuỗi biểu thức quy tắc như là tham số của nó. Điều này gây ra sự khó hiểu. Trong ngữ cảnh của một chương trình C#, cái nào là biểu thức quy tắc: chuỗi được đưa vào bộ khởi dựng hay là đối tượng Regex? Thật sự thì chuỗi ký tự được truyền vào chính là biểu thức quy tắc theo ý nghĩa truyền thống của thuật ngữ này. Tuy nhiên, theo quan điểm hướng đối tượng của ngôn ngữ C#, đối mục hay tham số của bộ khởi tạo chỉ đơn thuần là chuỗi ký tự, và chính Regex mới là đối tượng biểu thức quy tắc!
Phần còn lại của chương trình thực hiện giống như ví dụ minh họa trước. Ngoại trừ việc gọi phương thức Split() của đối tượng Regex chứ không phải của chuỗi s1. Regex.Split() hành động cũng tương tự như cách String.Split(). Kết quả trả về là mảng các chuỗi, đây chính là các chuỗi con so khớp tìm được theo mẫu đưa ra trong theRegex.
Phương thức Regex.Split() là phương thức được nạp chồng. Phiên bản đơn giản được gọi trong thể hiện của Regex được dùng như trong ví dụ trên. Ngoài ra còn có một phiên bản tĩnh của phương thức này. Phiên bản này lấy một chuỗi để thực hiện việc tìm kiếm và một mẫu để so khớp. Tiếp sau là minh họa sau sử dụng phương thức tĩnh
Sử dụng phương thức tĩnh Regex.Split().
---
namespace Programming_CSharp
{ using System; using System.Text; using System.Text.RegularExpressions; public class Tester { static void Main() { // tạo chuỗi tìm kiếm string s1 = "Mot, hai, ba Trung Tam Dao Tao CNTT"; StringBuilder sBuilder = new StringBuilder(); int id = 1; // ở đây không tạo thể hiện của Regex do sử dụng phương // thức tĩnh của lớp Regex. foreach(string subStr in Regex.Split(s1, " |, ")) { sBuilder.AppendFormat("{0}: {1}\n", id++, subStr); } Console.WriteLine("{0}", sBuilder);

```
}
}
}
```

Kết quả của ví dụ minh họa trên hoàn toàn tương tự như minh họa 10.5. Tuy nhiên trong chương trình thì chúng ta không tạo thể hiện của đối tượng Regex. Thay vào đó chúng ta sử dụng trực tiếp phương thức tĩnh của Regex là Split(). Phương thức này lấy vào hai tham số, tham số đầu tiên là chuỗi đích cần thực hiện so khớp và tham số thứ hai là chuỗi biểu thức quy tắc dùng để so khớp. **Sử dụng Regex để tìm kiếm tập hợp**
Hai lớp được thêm vào trong namespace .NET cho phép chúng ta thực hiện việc tìm kiếm một chuỗi một cách lập đi lặp lại cho đến hết chuỗi, và kết quả trả về là một tập hợp. Tập hợp được trả về có kiểu là MatchCollection, bao gồm không có hay nhiều đối tượng Match. Hai thuộc tính quan trọng của những đối tượng Match là chiều dài và giá trị của nó, chúng có thể được đọc như trong ví dụ minh họa dưới đây.
Sử dụng MatchCollection và Match.

```
namespace Programming_CSharp { using System; using System.Text.RegularExpressions; class Tester { static void Main() { string string1 = "Ngon ngu lap trinh C Sharp"; // tìm bất cứ chuỗi con nào không có khoảng trắng
// bên trong và kết thúc là khoảng trắng
Regex theReg = new Regex(@"(\S+)\s");
// tạo tập hợp và nhận kết quả so khớp MatchCollection theMatches = theReg.Matches(string1); // lặp để lấy kết quả từ tập hợp foreach ( Match theMatch in theMatches) { Console.WriteLine("Chieu dai: {0}", theMatch.Length); // nếu tồn tại chuỗi thì xuất ra if ( theMatch.Length != 0) { Console.WriteLine("Chuoi: {0}", theMatch.ToString());
}// end if
}// end foreach
}// end Main
}// end class
}// end namespace
```

Kết quả:
```
Chieu dai: 5
Chuoi: Ngon
Chieu dai: 4
Chuoi: ngu
Chieu dai: 4
Chuoi: lap
Chieu dai: 6
Chuoi: trinh
Chieu dai: 2
Chuoi: C
```

Ví dụ bắt đầu bằng việc tạo một chuỗi tìm kiếm đơn giản:
```
string string1 = "Ngon ngu lap trinh C Sharp";
```
và một biểu thức quy tắc để thực hiện việc tìm kiếm trên chuỗi string1:
```
Regex theReg = new Regex(@"(\S+)\s");
```

Chuỗi \S tìm ký tự không phải ký tự trắng và dấu cộng chỉ ra rằng có thể có một hay nhiều ký tự. Chuỗi \s (chữ thường) chỉ ra là khoảng trắng. Kết hợp lại là tìm một chuỗi không có khoảng trắng bên trong nhưng theo sau cùng là một khoảng trắng. Chúng ta lưu ý khai báo chuỗi biểu thức quy tắc dạng chuỗi nguyên văn để dễ dàng dùng các ký tự escape như (\).
Kết quả được trình bày là năm từ đầu tiên được tìm thấy. Từ cuối cùng không được tìm thấy bởi vì nó không được theo sau bởi khoảng trắng. Nếu chúng ta chèn một khoảng trắng sau chữ "Sharp" và trước dấu ngoặc đóng, thì chương trình sẽ tìm được thêm chữ "Sharp".
Thuộc tính Length là chiều dài của chuỗi con tìm kiếm được. Chúng ta sẽ tìm hiểu sâu hơn về thuộc tính này trong phần sử dụng lớp CaptureCollection ở cuối chương.

Sử dụng Regex để gom nhóm

Đôi khi lập trình chúng ta cần gom nhóm một số các biểu thức tương tự với nhau theo một quy định nào đó. Ví dụ như chúng ta cần tìm kiếm địa chỉ IP và nhóm chúng lại vào trong nhóm IPAddresses được tìm thấy bất cứ đâu trong một chuỗi.
Lớp Group cho phép chúng ta tạo những nhóm và tìm kiếm dựa trên biểu thức quy tắc, và thể hiện kết quả từ một nhóm biểu thức đơn.
Một biểu thức nhóm định rõ một nhóm và cung cấp một biểu thức quy tắc, bất cứ chuỗi con nào được so khớp bởi biểu thức quy tắc thì sẽ được thêm vào trong nhóm.
Để tạo một nhóm chúng ta có thể viết như sau:
@"(?<ip>(\d|\ .)+)\s"
Lớp Match dẫn xuất từ nhóm Group, và có một tập hợp gọi là Groups chứa tất cả các nhóm mà Match tìm thấy.
Việc tạo và sử dụng tập hợp Groups và lớp Group được minh họa trong ví dụ như sau:
Sử dụng lớp Group.

```
---------------------------------------------------------------------------
namespace Programming_CSharp { using System; using System.Text.RegularExpressions; class Tester { public static void Main() { string string1 = "10:20:30 127.0.0.0 Dolphin.net"; // nhóm thời gian bằng một hay nhiều con số hay dấu :
// và theo sau bởi khoảng trắng.
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" + // địa chỉ IP là một hay nhiều con số hay dấu chấm theo
// sau bởi khoảng trắng
@"(?<ip>(\d|\.)+)\s" + // địa chỉ web là một hay nhiều ký tự
@"(?<site>\S+)");
// lấy một tập hợp các chuỗi được so khớp
MatchCollection theMatches = theReg.Matches( string1 ); // sử dụng vòng lặp để lấy các chuỗi trong tập hợp foreach (Match theMatch in theMatches) { if (theMatch.Length != 0) { Console.WriteLine("\ntheMatch: {0}", theMatch.ToString());
// hiển thị thời gian Console.WriteLine("Time: {0}", theMatch.Groups["time"]);
// hiển thị địa chỉ IP Console.WriteLine("IP: {0}", theMatch.Groups["ip"]);
// hiển thị địa chỉ web site Console.WriteLine("Site: {0}", theMatch.Groups["site"]);
}// end if
}// end foreach
}// end Main
}// end class
```

}// end namespace ----------
Ví dụ minh họa trên bắt đầu bằng việc tạo một chuỗi đơn giản để tìm kiếm như sau:
string string1 = "10:20:30 127.0.0.0 Dolphin.net";
Chuỗi này có thể được tìm thấy trong nội dung của các tập tin log ghi nhận các thông tin ở web server hay từ các kết quả tìm kiếm được trong cơ sở dữ liệu. Trong ví dụ đơn giản này có ba cột, một cột đầu tiên ghi nhận thời gian, cột thứ hai ghi nhận địa chỉ IP, và cột thứ ba ghi nhận địa chỉ web. Mỗi cột được ngăn cách bởi khoảng trắng. Dĩ nhiên là trong các ứng dụng thực tế ta phải giải quyết những vấn đề phức tạp hơn nữa, chúng ta có thể cần phải thực hiện việc tìm kiếm phức tạp hơn và sử dụng nhiều ký tự ngăn cách hơn nữa.
Trong ví dụ này, chúng ta mong muốn là tạo ra một đối tượng Regex để tìm kiếm chuỗi con yêu cầu và phân chúng vào trong ba nhóm: time, địa chỉ IP, và địa chỉ web. Biểu thức quy tắc ở đây cũng khá đơn giản, do đó cũng dễ hiểu.
Ở đây chúng ta quan tâm đến những ký tự tạo nhóm như:
<?<time>
Dấu ngoặc đơn dùng để tạo nhóm. Mọi thứ giữa dấu ngoặc mở trước dấu ? và dấu ngoặc đóng (trong trường hợp này sau dấu +) được xác định là một nhóm. Chuỗi ?<time> định ra tên của nhóm và liên quan đến tất cả các chuỗi ký tự được so khớp theo biểu thức quy tắc (\d|\:)+)\s. Biểu thức này có thể được diễn giải như: “một hay nhiều con số hay những dấu :theo sau bởi một khoảng trắng”.
Tương tự như vậy, chuỗi ?<ip> định tên của nhóm ip, và ?<site> là tên của nhóm site.
Tiếp theo là một tập hợp được định nghĩa để nhận tất cả các chuỗi con được so khớp như sau:
MatchCollection theMatches = theReg.Matches(string1);
Vòng lặp foreach được dùng để lấy ra các chuỗi con được tìm thấy trong tập hợp.
Nếu chiều dài Length của Match là lớn hơn 0, tức là tìm thấy thì chúng ta sẽ xuất ra chuỗi được tìm thấy:
Console.WriteLine("\ntheMatch: {0}", theMatch.ToString());
Và kết quả của ví dụ là:
theMatch: 10:20:30 127.0.0.0 Dolphin.net
Sau đó chương trình lấy nhóm time từ tập hợp nhóm của Match và xuất ra màn hình bằng các lệnh như sau:
Console.WriteLine("time: {0}", theMatch.Groups["time"]);
Kết quả là :
Time: 10:20:30
Tương tự như vậy với nhóm ip và site:
Console.WriteLine("IP: {0}", theMatch.Groups["ip"]);
// hiển thị địa chỉ web site Console.WriteLine("site: {0}", theMatch.Groups["site"]);
Ta nhận được kết quả:
IP: 127.0.0.0
Site: Dolphin.net
Trong ví dụ 10.8 trên thì tập hợp Match chỉ so khớp duy nhất một lần. Tuy nhiên, nó có thể so khớp nhiều hơn nữa trong một chuỗi. Để làm được điều này, chúng ta có thể bổ sung chuỗi tìm kiếm được lấy từ trong một log file như sau:
String string1 = "10:20:30 127.0.0.0 Dolphin.net " + "10:20:31 127.0.0.0 Mun.net " + "10:20:32 127.0.0.0 Msn.net ";
Chuỗi này sẽ tạo ra ba chuỗi con so khớp được tìm thấy trong MatchCollection. Và kết quả ta có thể thấy được là:

theMatch: 10:20:30 127.0.0.0 Dolphin.net
Time: 10:20:30 IP: 127.0.0.0 site: Dolphin.net theMatch: 10:20:31 127.0.0.0 Mun.net
Time: 10:20:31
IP: 127.0.0.0 Site: Mun.net theMatch: 10:20:32 127.0.0.0 Msn.net time: 10:20:32
IP: 127.0.0.0
Site: Msn.net
Trong ví dụ này phần bổ sung, thì theMatches chứa ba đối tượng Match. Mỗi lần lặp thì các chuỗi con được tìm thấy (ba lần) và chúng ta có thể xuất ra chuỗi cũng như từng nhóm riêng bên trong của chuỗi con được tìm thấy.

Sử dụng lớp CaptureCollection

Mỗi khi một đối tượng Regex tìm thấy một chuỗi con, thì môt thể hiện Capture được tạo ra và được thêm vào trong một tập hợp CaptureCollection. Mỗi một đối tượng Capture thể hiện một chuỗi con riêng. Mỗi nhóm có một tập hợp các Capture được tìm thấy trong chuỗi con có liên hệ với nhóm.
Thuộc tính quan trọng của đối tượng Capture là thuộc tính Length, đây chính là chiều dài của chuỗi con được nắm giữ. Khi chúng ta hỏi Match chiều dài của nó, thì chúng ta sẽ nhận được Capture.Length do Match được dẫn xuất từ Group và đến lượt Group lại được dẫn xuất từ Capture.
Mô hình kế thừa trong biểu thức quy tắc của .NET cho phép Match thừa hưởng những giao diện phương thức và thuộc tính của những lớp cha của nó. Theo ý nghĩa này, thì một Group là một Capture (Group is-a Capture), là một đối tượng Capture đóng gói các ý tưởng về các nhóm biểu thức. Đến lượt Match, nó cũng là một Group (Match is-a Group), nó đóng gói tất cả các nhóm biểu thức con được so khớp trong biểu thức quy tắc (Xem chi tiết hơn trong chương 5: Kế thừa và đa hình).
Thông thường, chúng ta sẽ tìm thấy chỉ một Capture trong tập hợp CaptureCollection; nhưng điều này không phải vậy. Chúng ta thử tìm hiểu vấn đề như sau, ở đây chúng ta sẽ gặp trường hợp là phân tích một chuỗi trong đó có nhóm tên của công ty được xuất hiện hai lần. Để nhóm chúng lại trong chuỗi tìm thấy chúng ta tạo nhóm ?<company> xuất hiện ở hai nơi trong mẫu biểu thức quy tắc như sau:

```
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s"
+@"(?<company>\S+)\s" +
@"(?<ip>(\d|\.)+)\s" +
@"(?<company>\S+)\s");
```

Biểu thức quy tắc này nhóm bất cứ chuỗi nào hợp với mẫu so khớp time, và cũng như bất cứ chuỗi nào theo nhóm ip. Giả sử chúng ta dùng chuỗi sau để làm chuỗi tìm kiếm:

```
string string1 = "10:20:30 IBM 127.0.0.0 HP";
```

Chuỗi này chứa tên của hai công ty ở hai vị trí khác nhau, và kết quả thực hiện chương trình là như sau:

```
theMatch: 10:20:30 IBM 127.0.0.0 HP Time: 10:20:30
IP: 127.0.0.0
Company: HP
```

Điều gì xảy ra? Tại sao nhóm Company chỉ thể hiện giá trị HP. Còn chuỗi đầu tiên ở đâu hay là không được tìm thấy? Câu trả lời chính xác là mục thứ hai đã viết chồng mục đầu. Tuy nhiên, Group vẫn lưu giữ cả hai giá trị. Và ta dùng tập hợp Capture để lấy các giá trị này.
Tìm hiểu tập hợp CaptureCollection.

```
namespace Programming_CSharp { using System; using System.Text.RegularExpressions; class Test { public static void Main()
{
// tạo một chuỗi để phân tích // lưu ý là tên công ty được xuất // hiện cả hai nơi string string1 = "10:20:30 IBM 127.0.0.0 HP"; // biểu thức quy tắc với việc nhóm hai lần tên công ty Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
@"(?<company>\S+)\s" +
@"(?<ip>(\d|\ .)+)\s" +
@"(?<company>\S+)\s");
// đưa vào tập hợp các chuỗi được tìm thấy MatchCollection theMatches = theReg.Matches( string1 ); // dùng vòng lặp để lấy kết quả foreach ( Match theMatch in theMatches) { if ( theMatch.Length !=0 ) {
Console.WriteLine("theMatch: {0}", theMatch.ToString());
Console.WriteLine("Tme: {0}", theMatch.Groups["time"]);
Console.WriteLine("IP{0}", theMatch.Groups["ip"]);
Console.WriteLine("Company: {0}", theMatch.Groups["company"]);
// lặp qua tập hợp Capture để lấy nhóm company
foreach ( Capture cap in theMatch.Groups["Company"].Captures) { Console.WriteLine("Capture: {0}", cap.ToString());
}// end foreach
}// end if
}// end foreach
}// end Main
}// end class
}// end namespace
```

Kết quả: theMatch: 10:20:30 IBM 127.0.0.0 HP
Time: 10:20:30
IP: 127.0.0.0
Company: HP Capture: IBM Capture: HP

Trong đoạn vòng lặp cuối cùng:

```
foreach ( Capture cap in theMatch.Groups["Company"].Captures) {
Console.WriteLine("Capture: {0}", cap.ToString());
}// end foreach
```

Đoạn lặp này lặp qua tập hợp Capture của nhóm Company. Chúng ta thử tìm hiểu cách phân tích như sau. Trình biên dịch bắt đầu tìm một tập hợp cái mà chúng sẽ thực hiện việc lặp. theMatch là một đối tượng có một tập hợp tên là Groups. Tập hợp Groups có một chỉ mục đưa ra một chuỗi và trả về một đối tượng Group. Do vậy, dòng lệnh sau trả về một đối tượng đơn Group:

```
theMatch.Groups["company"];
```

Đối tượng Group có một tập hợp tên là Captures, và dòng lệnh tiếp sau trả về một tập hợp
Captures cho Group lưu giữ tại Groups[“company”] bên trong đối tượng theMatch:

```
theMatch.Groups["company"].Captures
```

Vòng lặp foreach lặp qua tập hợp Captures, và lấy từng thành phần ra và gán cho biến cục bộ cap, biến này có kiểu là Capture. Chúng ta có thể xem từ kết quả là có hai thành phần được lưu giữ là : IBM và HP. Chuỗi thứ hai viết chồng lên chuỗi thứ nhất trong nhóm, do vậy chỉ hiển thị giá trị thứ hai là HP. Tuy nhiên, bằng việc sử dụng tập hợp Captures chúng ta có thể thu được cả hai giá trị được lưu giữ.

Bài tập:

Viết chương trình in ra màn hình họ tên của mình, sau đó đếm số ký tự có trong tên của mình.

Bài tập nâng cao:

Viết chương trình cho phép nhập vào họ tên của một người, sau đó tiến hành đảo chuỗi. Ví dụ: Nguyễn Văn A -> A Văn Nguyễn.

Những trọng tâm cần chú ý trong bài:

- Hiểu được đặc tính các lớp dựng sẵn string trong C#;

- Sử dụng thạo các lớp có sẵn để làm các bài tập;

- Nghiêm túc, tỉ mỉ trong học lý thuyết và làm bài tập

Yêu cầu về đánh giá kết quả học tập:

Nội dung:

+ Về kiến thức:

    - Hiểu được đặc tính các lớp dựng sẵn string trong C#;

    - Sử dụng thạo các lớp có sẵn để làm các bài tập;

+ Về kỹ năng: Tạo và thực thi được ứng dụng xử lý chuỗi đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

Phương pháp:

+ Về kiến thức: Được đánh giá bằng hình thức kiểm tra viết, trắc nghiệm, vấn đáp

+ Về kỹ năng: Tạo và thực thi được ứng dụng đơn giản trên C#.

+ Năng lực tự chủ và trách nhiệm: Tỉ mỉ, cẩn thận, chính xác, ngăn nắp trong công việc.

TÀI LIỆU THAM KHẢO

+ C# 2008 - Lập Trình Cơ Bản và nâng cao. Nhà xuất bản lao động và xã hội;
+ MSDN Library;
+ Jesse Liberty, *Programming C#,* Nhà xuất bản: O'Reilly;
+ Bradley L.Jones ,C# in 21 Days, Nhà xuất bản: SAMS.