NGÔN NGỮ

LẬP TRÌNH C#

Mục Lục

1. Microsoft .NET 10
Tình hình trước khi MS.NET ra đời 10
Nguồn gốc của .NET 12
Microsoft .NET 12
Tổng quan 12
Kiến trúc .NET Framework 13
Common Language Runtime 15
Thư viện .NET Framework 16
Phát triển ứng dụng client 16
Biên dịch và MSIL 17
Ngôn ngữ C# 18
2. Ngôn ngữ C# 20
Tại sao phải sử dụng ngôn ngữ C# 20
C# là ngôn ngữ đơn giản 20
C# là ngôn ngữ hiện đại 21
C# là ngôn ngữ hướng đối tượng 21
C# là ngôn ngữ mạnh mẽ 22
C# là ngôn ngữ ít từ khóa 22
C# là ngôn ngữ module hóa 22
C# sẽ là ngôn ngữ phổ biến 22
Ngôn ngữ C# với ngôn ngữ khác 23
Các bước chuẩn bị cho chương trình 24
Chương trình C# đơn giản 25
Phát triển chương trình minh họa 31
Câu hỏi & bài tập 35
3. Nền tảng ngôn ngữ C# 39
Kiểu dữ liệu 40
Kiểu dữ liệu xây dựng sẵn 41
Chọn kiểu dữ liệu 42
Chuyển đổi kiểu dữ liệu 43
Biến và hằng 44
Gán giá trị xác định cho biến 45
Hằng 46
Kiểu liệt kê 47

Kiểu chuỗi ký tự....50
Định danh....50
Biểu thức....50
Khoảng trắng....51
Câu lệnh....51
Phân nhánh không có điều kiện....52
Phân nhánh có điều kiện....53
Câu lệnh lặp....60
Toán tử....68
Namespace....76
Các chỉ dẫn biên dịch....80
Câu hỏi & bài tập....82
4. Xây dựng lớp - Đối tượng....87
Định nghĩa lớp....88
Thuộc tính truy cập....91
Tham số của phương thức....92
Tạo đối tượng....93
Bộ khởi dựng....93
Khởi tạo biến thành viên....96
Bộ khởi dựng sao chép....98
Từ khóa this....99
Sử dụng các thành viên static....100
Gọi phương thức static....101
Sử dụng bộ khởi dựng static....101
Sử dụng bộ khởi dựng private....102
Sử dụng thuộc tính static....102
Hủy đối tượng....104
Truyền tham số....107
Nạp chồng phương thức....112
Đóng gói dữ liệu với thuộc tính....116
Thuộc tính chỉ đọc....119
Câu hỏi & bài tập....121
5. Kế thừa – Đa hình....125
Đặc biệt hóa và tổng quát hóa....126
Sự kế thừa....129
Thực thi kế thừa....129
Gọi phương thức khởi dựng của lớp cơ sở....131
Gọi phương thức của lớp cơ sở....132

Điều khiển truy xuất....132
Đa hình....133
Kiểu đa hình....133
Phương thức đa hình....133
Từ khóa new và override....137
Lớp trừu tượng....139
Gốc của tất cả các lớp- lớp Object....142
Boxing và Unboxing dữ liệu....144
Boxing dữ liệu ngầm định....144
Unboxing phải thực hiện tường minh....145
Các lớp lồng nhau....147
Câu hỏi & bài tập....149
6. Nạp chồng toán tử....153
Sử dụng từ khóa operator....153
Hỗ trợ ngôn ngữ .NET khác....154
Sử dụng toán tử....154
Toán tử so sánh bằng....156
Toán tử chuyển đổi....157
Câu hỏi & bài tập....163
7. Cấu trúc....165
Định nghĩa một cấu trúc....165
Tạo cấu trúc....168
Cấu trúc là một kiểu giá trị....168
Gọi bộ khởi dựng mặc định....169
Tạo cấu trúc không gọi new....170
Câu hỏi & bài tập....172
8. Thực thi giao diện....176
Thực thi giao diện....177
Thực thi nhiều giao diện....180
Mở rộng giao diện....181
Kết hợp các giao diện....181
Truy cập phương thức giao diện....187
Gán đối tượng cho giao diện....187
Toán tử is....188
Toán tử as....190
Giao diện đối lập với trừu tượng....192
Thực thi phủ quyết giao diện....193
Thực thi giao diện tường minh....197

Lựa chọn thể hiện phương thức giao diện....200
Ẩn thành viên....200
Câu hỏi & bài tập....207
9. Mảng, chỉ mục, và tập hợp....211
Mảng....212
Khai báo mảng....213
Giá trị mặc định....214
Truy cập các thành phần trong mảng....214
Khởi tạo thành phần trong mảng....216
Sử dụng từ khóa params....216
Câu lệnh foreach....218
Mảng đa chiều....220
Mảng đa chiều cùng kích thước....220
Mảng đa chiều có kích thước khác nhau....224
Chuyển đổi mảng....227
Bộ chỉ mục....232
Bộ chỉ mục và phép gán....236
Sử dụng kiểu chỉ số khác....237
Giao diện tập hợp....241
Giao diện IEnumerable....242
Giao diện ICollection....246
Danh sách mảng....247
Thực thi IComparable....251
Thực thi IComparer....254
Hàng đợi....259
Ngăn xếp....262
Kiểu từ điển....265
Hastables....266
Giao diện IDictionary....267
Tập khóa và tập giá trị....269
Giao diện IDictionaryEnumerator....270
Câu hỏi & bài tập....271
10. Xử lý chuỗi....275
Lớp đối tượng string....276
Tạo một chuỗi....276
Tạo một chuỗi dùng phương thức ToString....277
Thao tác trên chuỗi....278
Tìm một chuỗi con....285

Chia chuỗi....286
Thao tác trên chuỗi dùng StringBuilder....288
Các biểu thức quy tắc....290
Sử dụng biểu thức quy tắc qua lớp Regex....291
Sử dụng Regex để tìm tập hợp....294
Sử dụng Regex để gom nhóm....295
Sử dụng CaptureCollection....298
Câu hỏi & bài tập....301
11. Cơ chế ủy quyền và sự kiện....303
Ủy quyền....304
Sử dụng ủy quyền xác nhận phương thức lúc thực thi....304
Ủy quyền tĩnh....314
Dùng ủy quyền như thuộc tính....315
Thiết lập thứ tự thi hành với mảng ủy quyền....316
Multicasting....320
Sự kiện....324
Cơ chế publishing- subscribing....324
Sự kiện và ủy quyền....325
Câu hỏi & bài tập....333
12. Các lớp cơ sở .NET....335
Lớp đối tượng trong .NET Framework....335
Lớp Timer....337
Lớp về thư mục và hệ thống....340
Lớp Math....342
Lớp thao tác tập tin....345
Làm việc với tập tin dữ liệu....351
Câu hỏi & bài tập....362
13. Xử lý ngoại lệ....364
Phát sinh và bắt giữ ngoại lệ....365
Câu lệnh throw....365
Câu lệnh catch....367
Câu lệnh finally....373
Những đối tượng ngoại lệ....375
Tạo riêng các ngoại lệ....378
Phát sinh lại ngoại lệ....381
Câu hỏi & bài tập....385

Tham Khảo

Giáo trình "*Ngôn ngữ Lập trình C#*" được biên dịch và tổng hợp từ:

- ☆*Programming C#*, Jesse Liberty, O'Reilly.
- ☆*C# in 21 Days*, Bradley L.Jones, SAMS.
- ☆*Windows Forms Programming with C#*, Erik Brown, Manning.
- ☆*MSDN Library – April 2002*.

Quy ước

Giáo trình sử dụng một số quy ước như sau:

- Các thuật ngữ được giới thiệu lần đầu tiên sẽ *in nghiêng*.
- Mã nguồn của chương trình minh họa dùng font Verdana -10.
- Các từ khóa của C# dùng font **Verdana-10, đậm** hoặc Verdana-10, bình thường.
- Tên namspace, lớp, đối tượng, phương thức, thuộc tính, sự kiện... dùng font Verdana-10.
- Kết quả của chương trình xuất ra màn hình console dùng font `Courier New-10`.

Chương 1

MICROSOFT .NET

- **Tình hình trước khi MS.NET ra đời**
  - **Nguồn gốc của .NET**
- **Microsoft .NET**
  - **Tổng quan**
  - **Kiến trúc .NET Framework**
  - **Common Language Runtime (CLR)**
  - **Thư viện .NET Framework**
  - **Phát triển ứng dụng client**
- **Biên dịch và MSIL**
- **Ngôn ngữ C#**

Tình hình trước khi MS.NET ra đời

Trong lĩnh vực công nghệ thông tin của thế giới ngày nay, với sự phát triển liên tục và đa dạng nhất là phần mềm, các hệ điều hành, các môi trường phát triển, các ứng dụng liên tục ra đời. Tuy nhiên, đôi khi việc phát triển không đồng nhất và nhất là do lợi ích khác nhau của các công ty phần mềm lớn làm ảnh hưởng đến những người xây dựng phần mềm.

Cách đây vài năm Java được Sun viết ra, đã có sức mạnh đáng kể, nó hướng tới việc chạy trên nhiều hệ điều hành khác nhau, độc lập với bộ xử lý (Intel, Risc,…). Đặc biệt là Java rất thích hợp cho việc viết các ứng dụng trên Internet. Tuy nhiên, Java lại có hạn chế về mặt tốc độ và trên thực tế vẫn chưa thịnh hành. Mặc dù Sun Corporation và IBM có đẩy mạnh Java, nhưng Microsoft đã dùng ASP để làm giảm khả năng ảnh hưởng của Java.

Để lập trình trên Web, lâu nay người ta vẫn dùng CGI-Perl và gần đây nhất là PHP, một ngôn ngữ giống như Perl nhưng tốc độ chạy nhanh hơn. Ta có thể triển khai Perl trên Unix/Linux hay MS Windows. Tuy nhiên có nhiều người không thích dùng do bản thân ngôn ngữ hay các qui ước khác thường và Perl không được phát triển thống nhất, các công cụ được xây dựng cho Perl tuy rất mạnh nhưng do nhiều nhóm phát triển và người ta không đảm bảo rằng tương lai của nó ngày càng tốt đẹp hơn.

Trong giới phát triển ứng dụng trên Windows ta có thể viết ứng dụng bằng Visual C++, Delphi hay Visual Basic, đây là một số công cụ phổ biến và mạnh. Trong đó Visual C++ là một ngôn ngữ rất mạnh và cũng rất khó sử dụng. Visual Basic thì đơn giản dễ học, dễ dùng nhất nên rất thông dụng. Lý do chính là Visual Basic giúp chúng ta có thể viết chương trình trên Windows dễ dàng mà không cần thiết phải biết nhiều về cách thức MS Windows hoạt động, ta chỉ cần biết một số kiến thức căn bản tối thiểu về MS Windows là có thể lập trình được. Do đó theo quan điểm của Visual Basic nên nó liên kết với Windows là điều tự nhiên và dễ hiểu, nhưng hạn chế là Visual Basic không phải ngôn ngữ hướng đối tượng (Object Oriented).

Delphi là hậu duệ của Turbo Pascal của Borland. Nó cũng giống và tương đối dễ dùng như Visual Basic. Delphi là một ngôn ngữ hướng đối tượng. Các điều khiển dùng trên Form của Delphi đều được tự động khởi tạo mã nguồn. Tuy nhiên, chức năng khởi động mã nguồn này của Delphi đôi khi gặp rắc rối khi có sự can thiệp của người dùng vào. Sau này khi công ty Borland bị bán và các chuyên gia xây dựng nên Delphi đã chạy qua bên Microsoft, và Delphi không còn được phát triển tốt nữa, người ta không dám đầu tư triển khai phần mềm vào Delphi. Công ty sau này đã phát triển dòng sản phẩm Jbuilder (dùng Java) không còn quan tâm đến Delphi.

Tuy Visual Basic bền hơn do không cần phải khởi tạo mã nguồn trong Form khi thiết kế nhưng Visual Basic cũng có nhiều khuyết điểm :

- Không hỗ trợ thiết kế hướng đối tượng, nhất là khả năng thừa kế (inheritance).
- Giới hạn về việc chạy nhiều tiểu trình trong một ứng dụng, ví dụ ta không thể dùng Visual Basic để viết một Service kiểu NT.
- Khả năng xử lý lỗi rất yếu, không thích hợp trong môi trường Multi- tier
- Khó dùng chung với ngôn ngữ khác như C++.
- Không có User Interface thích hợp cho Internet.

Do Visual Basic không thích hợp cho viết các ứng Web Server nên Microsoft tạo ra ASP (Active Server Page). Các trang ASP này vừa có tag HTML vừa chứa các đoạn script (VBScript, JavaScript) nằm lẫn lộn nhau. Khi xử lý một trang ASP, nếu là tag HTML thì sẽ được gởi thẳng qua Browser, còn các script thì sẽ được chuyển thành các dòng HTML rồi gởi đi, ngoại trừ các function hay các sub trong ASP thì vị trí các script khác rất quan trọng.

Khi một số chức năng nào được viết tốt người ta dịch thành ActiveX và đưa nó vào Web Server. Tuy nhiên vì lý do bảo mật nên các ISP (Internet Service Provider) làm máy chủ cho Web site thường rất dè đặt khi cài ActiveX lạ trên máy của họ. Ngoài ra việc tháo gỡ các phiên bản của ActiveX này là công việc rất khó, thường xuyên làm cho Administrator nhức đầu. Những người đã từng quản lý các version của DLL trên Windows điều than phiền tại sao phải đăng ký các DLL và nhất là chỉ có thể đăng ký một phiên bản của DLL mà thôi. Và từ "*DLL Hell*" xuất hiện tức là địa ngục DLL…

Sau này để giúp cho việc lập trình ASP nhanh hơn thì công cụ Visual InterDev, một IDE (Integrated Development Environment) ra đời. Visual InterDev tạo ra các Design Time Controls cho việc thiết kế các điều khiển trên web,… Tiếc thay Visual InterDev không bền vững lắm nên sau một thời gian thì các nhà phát triển đã rời bỏ nó.

Tóm lại bản thân của ASP hãy còn một số khuyết điểm quan trọng, nhất là khi chạy trên Internet Information Server với Windows NT 4, ASP không đáng tin cậy lắm.

Tóm lại trong giới lập trình theo Microsoft thì việc lập trình trên desktop cho đến lập trình hệ phân tán hay trên web là không được nhịp nhàng cho lắm. Để chuyển được từ lập trình client hay desktop đến lập trình web là một chặng đường dài.

Nguồn gốc .NET

Đầu năm 1998, sau khi hoàn tất phiên bản Version 4 của Internet Information Server (IIS), các đội ngũ lập trình ở Microsoft nhận thấy họ còn rất nhiều sáng kiến để kiện toàn IIS. Họ bắt đầu xây dựng một kiến trúc mới trên nền tảng ý tưởng đó và đặt tên là Next Generation Windows Services (NGWS).

Sau khi Visual Basic được trình làng vào cuối 1998, dự án kế tiếp mang tên Visual Studio 7 được xác nhập vào NGWS. Đội ngũ COM+/MTS góp vào một universal runtime cho tất cả ngôn ngữ lập trình chung trong Visual Studio, và tham vọng của họ cung cấp cho các ngôn ngữ lập trình của các công ty khác dùng chung luôn. Công việc này được xúc tiến một cách hoàn toàn bí mật mãi cho đến hội nghị Professional Developers' Conference ở Orlado vào tháng 7/2000. Đến tháng 11/2000 thì Microsoft đã phát hành bản Beta 1 của .NET gồm 3 đĩa CD. Tính đến lúc này thì Microsoft đã làm việc với .NET gần 3 năm rồi, do đó bản Beta 1 này tương đối vững chắc.

.NET mang dáng dấp của những sáng kiến đã được áp dụng trước đây như p-code trong UCSD Pascal cho đến Java Virtual Machine. Có điều là Microsoft góp nhặt những sáng kiến của người khác, kết hợp với sáng kiến của chính mình để làm nên một sản phẩm hoàn chỉnh từ bên trong lẫn bên ngoài. Hiện tại Microsoft đã công bố phiên bản release của .NET.

Thật sự Microsoft đã đặt cược vào .NET vì theo thông tin của công ty, đã tập trung 80% sức mạnh của Microsoft để nghiên cứu và triển khai .NET (bao gồm nhân lực và tài chính ?), tất cả các sản phẩm của Microsoft sẽ được chuyển qua .NET.

Microsoft .NET

Tổng quan

Microsoft .NET gồm 2 phần chính : Framework và Integrated Development Environment (IDE). Framework cung cấp những gì cần thiết và căn bản, chữ Framework có nghĩa là khung hay khung cảnh trong đó ta dùng những hạ tầng cơ sở theo một qui ước nhất định để công việc được trôi chảy. IDE thì cung cấp một môi trường giúp chúng ta triển khai dễ dàng, và nhanh chóng các ứng dụng dựa trên nền tảng .NET. Nếu không có IDE chúng ta cũng có thể

dùng một trình soạn thảo ví như Notepad hay bất cứ trình soạn thảo văn bản nào và sử dụng command line để biên dịch và thực thi, tuy nhiên việc này mất nhiều thời gian. Tốt nhất là chúng ta dùng IDE phát triển các ứng dụng, và cũng là cách dễ sử dụng nhất.

Thành phần Framework là quan trọng nhất .NET là cốt lõi và tinh hoa của môi trường, còn IDE chỉ là công cụ để phát triển dựa trên nền tảng đó thôi. Trong .NET toàn bộ các ngôn ngữ C#, Visual C++ hay Visual Basic.NET đều dùng cùng một IDE.

Tóm lại Microsoft .NET là nền tảng cho việc xây dựng và thực thi các ứng dụng phân tán thế hệ kế tiếp. Bao gồm các ứng dụng từ client đến server và các dịch vụ khác. Một số tính năng của Microsoft .NET cho phép những nhà phát triển sử dụng như sau:

- Một mô hình lập trình cho phép nhà phát triển xây dựng các ứng dụng dịch vụ web và ứng dụng client với Extensible Markup Language (XML).
- Tập hợp dịch vụ XML Web, như Microsoft .NET My Services cho phép nhà phát triển đơn giản và tích hợp người dùng kinh nghiệm.
- Cung cấp các server phục vụ bao gồm: Windows 2000, SQL Server, và BizTalk Server, tất cả điều tích hợp, hoạt động, và quản lý các dịch vụ XML Web và các ứng dụng.
- Các phần mềm client như Windows XP và Windows CE giúp người phát triển phân phối sâu và thuyết phục người dùng kinh nghiệm thông qua các dòng thiết bị.
- Nhiều công cụ hỗ trợ như Visual Studio .NET, để phát triển các dịch vụ Web XML, ứng dụng trên nền Windows hay nền web một cách dễ dàng và hiệu quả.

Kiến trúc .NET Framework

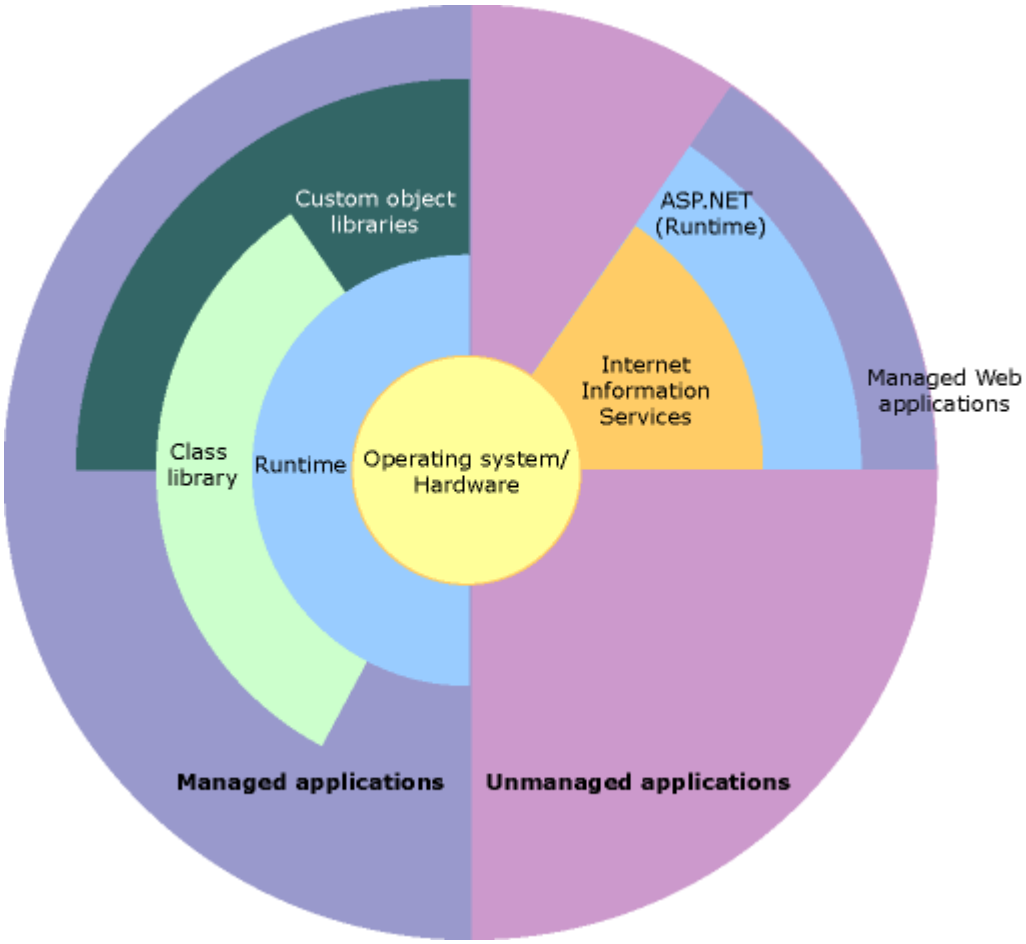
.NET Framework là một platform mới làm đơn giản việc phát triển ứng dụng trong môi trường phân tán của Internet. .NET Framework được thiết kế đầy đủ để đáp ứng theo quan điểm sau:

- Để cung cấp một môi trường lập trình hướng đối tượng vững chắc, trong đó mã nguồn đối tượng được lưu trữ và thực thi một cách cục bộ. Thực thi cục bộ nhưng được phân tán trên Internet, hoặc thực thi từ xa.
- Để cung cấp một môi trường thực thi mã nguồn mà tối thiểu được việc đóng gói phần mềm và sự tranh chấp về phiên bản.
- Để cung cấp một môi trường thực thi mã nguồn mà đảm bảo việc thực thi an toàn mã nguồn, bao gồm cả việc mã nguồn được tạo bởi hãng thứ ba hay bất cứ hãng nào mà tuân thủ theo kiến trúc .NET.
- Để cung cấp một môi trường thực thi mã nguồn mà loại bỏ được những lỗi thực hiện các script hay môi trường thông dịch.
- Để làm cho những người phát triển có kinh nghiệm vững chắc có thể nắm vững nhiều kiểu ứng dụng khác nhau. Như là từ những ứng dụng trên nền Windows đến những ứng dụng dựa trên web.

- Để xây dựng tất cả các thông tin dựa triên tiêu chuẩn công nghiệp để đảm bảo rằng mã nguồn trên .NET có thể tích hợp với bất cứ mã nguồn khác.

.NET Framework có hai thành phần chính: Common Language Runtime (CLR) và thư viện lớp .NET Framework. CLR là nền tảng của .NET Framework. Chúng ta có thể hiểu runtime như là một agent quản lý mã nguồn khi nó được thực thi, cung cấp các dịch vụ cốt lõi như: quản lý bộ nhớ, quản lý tiểu trình, và quản lý từ xa. Ngoài ra nó còn thúc đẩy việc sử dụng kiểu an toàn và các hình thức khác của việc chính xác mã nguồn, đảm bảo cho việc thực hiện được bảo mật và mạnh mẽ. Thật vậy, khái niệm quản lý mã nguồn là nguyên lý nền tảng của runtime. Mã nguồn mà đích tới runtime thì được biết như là mã nguồn được quản lý (managed code). Trong khi đó mã nguồn mà không có đích tới runtime thì được biết như mã nguồn không được quản lý (unmanaged code).

Thư viện lớp, một thành phần chính khác của .NET Framework là một tập hợp hướng đối tượng của các kiểu dữ liệu được dùng lại, nó cho phép chúng ta có thể phát triển những ứng dụng từ những ứng dụng truyền thống command-line hay những ứng dụng có giao diện đồ họa (GUI) đến những ứng dụng mới nhất được cung cấp bởi ASP.NET, như là Web Form và dịch vụ XML Web.



Hình 1.1: Mô tả các thành phần trong .NET Framework.

Common Language Runtime (CLR)

Như đã đề cập thì CLR thực hiện quản lý bộ nhớ, quản lý thực thi tiểu trình, thực thi mã nguồn, xác nhận mã nguồn an toàn, biên bịch và các dịch vụ hệ thống khác. Những đặc tính trên là nền tảng cơ bản cho những mã nguồn được quản lý chạy trên CLR.

Do chú trọng đến bảo mật, những thành phần được quản lý được cấp những mức độ quyền hạn khác nhau, phụ thuộc vào nhiều yếu tố nguyên thủy của chúng như: liên quan đến Internet, hệ thống mạng trong nhà máy, hay một máy tính cục bộ. Điều này có nghĩa rằng, một thành phần được quản lý có thể có hay không có quyền thực hiện một thao tác truy cập tập tin, thao tác truy cập registry, hay các chức năng nhạy cảm khác.

CLR thúc đẩy việc mã nguồn thực hiện việc truy cập được bảo mật. Ví dụ, người sử dụng giới hạn rằng việc thực thi nhúng vào trong một trang web có thể chạy được hoạt hình trên màn hình hay hát một bản nhạc, nhưng không thể truy cập được dữ liệu riêng tư, tập tin hệ thống, hay truy cập mạng. Do đó, đặc tính bảo mật của CLR cho phép những phần mềm đóng gói trên Inernet có nhiều đặc tính mà không ảnh hưởng đến việc bảo mật hệ thống.

CLR còn thúc đẩy cho mã nguồn được thực thi mạnh mẽ hơn bằng việc thực thi mã nguồn chính xác và sự xác nhận mã nguồn. Nền tảng của việc thực hiện này là Common Type System (CTS). CTS đảm bảo rằng những mã nguồn được quản lý thì được tự mô tả (self-describing). Sự khác nhau giữa Microsoft và các trình biên dịch ngôn ngữ của hãng thứ ba là việc tạo ra các mã nguồn được quản lý có thể thích hợp với CTS. Điều này thì mã nguồn được quản lý có thể sử dụng những kiểu được quản lý khác và những thể hiện, trong khi thúc đẩy nghiêm ngặt việc sử dụng kiểu dữ liệu chính xác và an toàn.

Thêm vào đó, môi trường được quản lý của runtime sẽ thực hiện việc tự động xử lý layout của đối tượng và quản lý những tham chiếu đến đối tượng, giải phóng chúng khi chúng không còn được sử dụng nữa. Việc quản lý bộ nhớ tự động này còn giải quyết hai lỗi chung của ứng dụng: thiếu bộ nhớ và tham chiếu bộ nhớ không hợp lệ.

Trong khi runtime được thiết kế cho những phần mềm của tương lai, nó cũng hỗ trợ cho phân mềm ngày nay và trước đây. Khả năng hoạt động qua lại giữa mã nguồn được quản lý và mã nguồn không được quản lý cho phép người phát triển tiếp tục sử dụng những thành phần cần thiết của COM và DLL.

Rutime được thiết kế để cải tiến hiệu suất thực hiện. Mặc dù CLR cung cấp nhiều các tiêu chuẩn dịch vụ runtime, nhưng mã nguồn được quản lý không bao giờ được dịch. Có một đặc tính gọi là Just-in-Time (JIT) biên dịch tất cả những mã nguồn được quản lý vào trong ngôn ngữ máy của hệ thống vào lúc mà nó được thực thi. Khi đó, trình quản lý bộ nhớ xóa bỏ những phân mảnh bộ nhớ nếu có thể được và gia tăng tham chiếu bộ nhớ cục bộ, và kết quả gia tăng hiệu quả thực thi.

Thư viện lớp .NET Framework

Thư viện lớp .NET Framework là một tập hợp những kiểu dữ liệu được dùng lại và được kết hợp chặt chẽ với Common Language Runtime. Thư viện lớp là hướng đối tượng cung cấp những kiểu dữ liệu mà mã nguồn được quản lý của chúng ta có thể dẫn xuất. Điều này không chỉ làm cho những kiểu dữ liệu của .NET Framework dễ sử dụng mà còn làm giảm thời gian liên quan đến việc học đặc tính mới của .NET Framework. Thêm vào đó, các thành phần của các hãng thứ ba có thể tích hợp với những lớp trong .NET Framework.

Cũng như mong đợi của người phát triển với thư viện lớp hướng đối tượng, kiểu dữ liệu . NET Framework cho phép người phát triển thiết lập nhiều mức độ thông dụng của việc lập trình, bao gồm các nhiệm vụ như: quản lý chuỗi, thu thập hay chọn lọc dữ liệu, kết nối với cơ cở dữ liệu, và truy cập tập tin. Ngoài những nhiệm vụ thông dụng trên. Thư viện lớp còn đưa vào những kiểu dữ liệu để hỗ trợ cho những kịch bản phát triển chuyên biệt khác. Ví dụ người phát triển có thể sử dụng .NET Framework để phát triển những kiểu ứng dụng và dịch vụ như sau:

- Ứ ng dụng Console
- Ứ ng dụng giao diện GUI trên Windows (Windows Forms)
- Ứ ng dụng ASP.NET
- Dịch vụ XML Web
- Dịch vụ Windows

Trong đó những lớp Windows Forms cung cấp một tập hợp lớn các kiểu dữ liệu nhằm làm đơn giản việc phát triển các ứng dụng GUI chạy trên Windows. Còn nếu như viết các ứng dụng ASP.NET thì có thể sử dụng các lớp Web Forms trong thư viện .NET Framework.

Phát triển ứng dụng Client

Những ứng dụng client cũng gần với những ứng dụng kiểu truyền thống được lập trình dựa trên Windows. Đây là những kiểu ứng dụng hiển thị những cửa sổ hay những form trên desktop cho phép người dùng thực hiện một thao tác hay nhiệm vụ nào đó. Những ứng dụng client bao gồm những ứng dụng như xử lý văn bản, xử lý bảng tính, những ứng dụng trong lĩnh vực thương mại như công cụ nhập liệu, công cụ tạo báo cáo...Những ứng dụng client này thường sử dụng những cửa sổ, menu, toolbar, button hay các thành phần GUI khác, và chúng thường truy cập các tài nguyên cục bộ như là các tập tin hệ thống, các thiết bị ngoại vi như máy in.

Một loại ứng dụng client khác với ứng dụng truyền thống như trên là ActiveX control (hiện nay nó được thay thế bởi các Windows Form control) được nhúng vào các trang web trên Internet. Các ứng dụng này cũng giống như những ứng dụng client khác là có thể truy cập tài nguyên cục bộ.

Trong quá khứ, những nhà phát triển có thể tạo các ứng dụng sử dụng C/C++ thông qua kết nối với MFC hoặc sử dụng môi trường phát triển ứng dụng nhanh (RAD: Rapid

Application Development). .NET Framework tích hợp diện mạo của những sản phẩm thành một. Môi trường phát triển cố định làm đơn giản mạnh mẽ sự phát triển của ứng dụng client.

Những lớp .NET Framework chứa trong .NET Framework được thiết kế cho việc sử dụng phát triển các GUI. Điều này cho phép người phát triển nhanh chóng và dễ dàng tạo các cửa sổ, button, menu, toolbar, và các thành phần khác trong các ứng dụng được viết phục vụ cho lĩnh vực thương mại. Ví dụ như, .NET cung cấp những thuộc tính đơn giản để hiệu chỉnh các hiệu ứng visual liên quan đến form. Trong vài trường hợp hệ điều hành không hỗ trợ việc thay đổi những thuộc tính này một cách trực tiếp, và trong trường hợp này .NET tự động tạo lại form. Đây là một trong nhiều cách mà .NET tích hợp việc phát triển giao diện làm cho mã nguồn đơn giản và mạnh mẽ hơn.

Không giống như ActiveX control, Windows Form control có sự truy cập giới hạn đến máy của người sử dụng. Điều này có nghĩa rằng mà nguồn thực thi nhị phân có thể truy cập một vài tài nguyên trong máy của người sử dụng (như các thành phần đồ họa hay một số tập tin được giới hạn) mà không thể truy cập đến những tài nguyên khác. Nguyên nhân là sự bảo mật truy cập của mã nguồn. Lúc này các ứng dụng được cài đặt trên máy người dùng có thể an toàn để đưa lên Internet

Biên dịch và MSIL

Trong .NET Framework, chương trình không được biên dịch vào các tập tin thực thi mà thay vào đó chúng được biên dịch vào những tập tin trung gian gọi là Microsoft Intermediate Language (MSIL). Những tập tin MSIL được tạo ra từ C# cũng tương tự như các tập tin MSIL được tạo ra từ những ngôn ngữ khác của .NET, platform ở đây không cần biết ngôn ngữ của mã nguồn. Điều quan trọng chính yếu của CLR là chung (common), cùng một runtime hỗ trợ phát triển trong C# cũng như trong VB.NET.

Mã nguồn C# được biên dịch vào MSIL khi chúng ta build project. Mã MSIL này được lưu vào trong một tập tin trên đĩa. Khi chúng ta chạy chương trình, thì MSIL được biên dịch một lần nữa, sử dụng trình biên dịch Just-In-Time (JIT). Kết quả là mã máy được thực thi bởi bộ xử lý của máy.

Trình biên dịch JIT tiêu chuẩn thì thực hiện theo yêu cầu. Khi một phương thức được gọi, trình biên dịch JIT phân tích MSIL và tạo ra sản phẩm mã máy có hiệu quả cao, mã này có thể chạy rất nhanh. Trình biên dịch JIT đủ thông minh để nhận ra khi một mã đã được biên dịch, do vậy khi ứng dụng chạy thì việc biên dịch chỉ xảy ra khi cần thiết, tức là chỉ biên dịch mã MSIL chưa biên dịch ra mã máy. Khi đó một ứng dụng .NET thực hiện, chúng có xu hướng là chạy nhanh và nhanh hơn nữa, cũng như là những mã nguồn được biên dịch rồi thì được dùng lại.

Do tất cả các ngôn ngữ .NET Framework cùng tạo ra sản phẩm MSIL giống nhau, nên kết quả là một đối tượng được tạo ra từ ngôn ngữ này có thể được truy cập hay được dẫn xuất từ

một đối tượng của ngôn ngữ khác trong .NET. Ví dụ, người phát triển có thể tạo một lớp cơ sở trong VB.NET và sau đó dẫn xuất nó trong C# một cách dễ dàng.

Ngôn ngữ C#

Ngôn ngữ C# khá đơn giản, chỉ khoảng 80 từ khóa và hơn mười mấy kiểu dữ liệu được xây dựng sẵn. Tuy nhiên, ngôn ngữ C# có ý nghĩa cao khi nó thực thi những khái niệm lập trình hiện đại. C# bao gồm tất cả những hỗ trợ cho cấu trúc, thành phần component, lập trình hướng đối tượng. Những tính chất đó hiện diện trong một ngôn ngữ lập trình hiện đại. Và ngôn ngữ C# hội đủ những điều kiện như vậy, hơn nữa nó được xây dựng trên nền tảng của hai ngôn ngữ mạnh nhất là C++ và Java.

Ngôn ngữ C# được phát triển bởi đội ngũ kỹ sư của Microsoft, trong đó người dẫn đầu là Anders Hejlsberg và Scott Wiltamuth. Cả hai người này điều là những người nổi tiếng, trong đó Anders Hejlsberg được biết đến là tác giả của Turbo Pascal, một ngôn ngữ lập trình PC phổ biến. Và ông đứng đầu nhóm thiết kế Borland Delphi, một trong những thành công đầu tiên của việc xây dựng môi trường phát triển tích hợp (IDE) cho lập trình client/server.

Phần cốt lõi hay còn gọi là trái tim của bất cứ ngôn ngữ lập trình hướng đối tượng là sự hỗ trợ của nó cho việc định nghĩa và làm việc với những lớp. Những lớp thì định nghĩa những kiểu dữ liệu mới, cho phép người phát triển mở rộng ngôn ngữ để tạo mô hình tốt hơn để giải quyết vấn đề. Ngôn ngữ C# chứa những từ khóa cho việc khai báo những kiểu lớp đối tượng mới và những phương thức hay thuộc tính của lớp, và cho việc thực thi đóng gói, kế thừa, và đa hình, ba thuộc tính cơ bản của bất cứ ngôn ngữ lập trình hướng đối tượng.

Trong ngôn ngữ C# mọi thứ liên quan đến khai báo lớp điều được tìm thấy trong phần khai báo của nó. Định nghĩa một lớp trong ngôn ngữ C# không đòi hỏi phải chia ra tập tin header và tập tin nguồn giống như trong ngôn ngữ C++. Hơn thế nữa, ngôn ngữ C# hỗ trợ kiểu XML, cho phép chèn các tag XML để phát sinh tự động các document cho lớp.

C# cũng hỗ trợ giao diện interface, nó được xem như một cam kết với một lớp cho những dịch vụ mà giao diện quy định. Trong ngôn ngữ C#, một lớp chỉ có thể kế thừa từ duy nhất một lớp cha, tức là không cho đa kế thừa như trong ngôn ngữ C++, tuy nhiên một lớp có thể thực thi nhiều giao diện. Khi một lớp thực thi một giao diện thì nó sẽ hứa là nó sẽ cung cấp chức năng thực thi giao diện.

Trong ngôn ngữ C#, những cấu trúc cũng được hỗ trợ, nhưng khái niệm về ngữ nghĩa của nó thay đổi khác với C++. Trong C#, một cấu trúc được giới hạn, là kiểu dữ liệu nhỏ gọn, và khi tạo thể hiện thì nó yêu cầu ít hơn về hệ điều hành và bộ nhớ so với một lớp. Một cấu trúc thì không thể kế thừa từ một lớp hay được kế thừa nhưng một cấu trúc có thể thực thi một giao diện.

Ngôn ngữ C# cung cấp những đặc tính hướng thành phần (component-oriented), như là những thuộc tính, những sự kiện. Lập trình hướng thành phần được hỗ trợ bởi CLR cho phép lưu trữ metadata với mã nguồn cho một lớp. Metadata mô tả cho một lớp, bao gồm những

phương thức và những thuộc tính của nó, cũng như những sự bảo mật cần thiết và những thuộc tính khác. Mã nguồn chứa đựng những logic cần thiết để thực hiện những chức năng của nó.. Do vậy, một lớp được biên dịch như là một khối self-contained, nên môi trường hosting biết được cách đọc metadata của một lớp và mã nguồn cần thiết mà không cần những thông tin khác để sử dụng nó.

Một lưu ý cuối cùng về ngôn ngữ C# là ngôn ngữ này cũng hỗ trợ việc truy cập bộ nhớ trực tiếp sử dụng kiểu con trỏ của C++ và từ khóa cho dấu ngoặc [] trong toán tử. Các mã nguồn này là không an toàn (unsafe). Và bộ giải phóng bộ nhớ tự động của CLR sẽ không thực hiện việc giải phóng những đối tượng được tham chiếu bằng sử dụng con trỏ cho đến khi chúng được giải phóng.

Chương 2

NGÔN NGỮ C#

- **Tại sao phải sử dụng ngôn ngữ C#**
  - **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**
  - **C# là ngôn ngữ ít từ khóa**
  - **C# là ngôn ngữ module hóa**
  - **C# sẽ là ngôn ngữ phổ biến**
- **Ngôn ngữ C# và những ngôn ngữ khác**
- **Các bước chuẩn bị cho chương trình**
- **Chương trình C# đơn giản**
- **Phát triển chương trình minh họa**
- **Câu hỏi & bài tập**

Tại sao phải sử dụng ngôn ngữ 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

C# là ngôn ngữ đơn giản

C# loại bỏ một vài sự phức tạp và rối rắm của những ngôn ngữ như Java và c++, bao gồm việc loại bỏ những macro, những template, đa kế thừa, và lớp cơ sở ảo (virtual base class). Chúng là những nguyên nhân gây ra sự nhầm lẫn hay dẫn đến những vấn đề cho các người phát triển C++. Nếu chúng ta là người học ngôn ngữ này đầu tiên thì chắc chắn là ta sẽ không trải qua những thời gian để học nó! Nhưng khi đó ta sẽ không biết được hiệu quả của ngôn ngữ C# khi loại bỏ những vấn đề trên.

Ngôn ngữ C# đơn giản vì nó dựa trên nền tảng C và C++. Nếu chúng ta thân thiện với C và C++ hoặc thậm chí là Java, chúng ta sẽ thấy C# khá giống về diện mạo, cú pháp, biểu thức, toán tử và những chức năng khác được lấy trực tiếp từ ngôn ngữ C và C++, nhưng nó đã được cải tiến để làm cho ngôn ngữ đơn giản hơn. Một vài trong các sự cải tiến là loại bỏ các dư thừa, hay là thêm vào những cú pháp thay đổi. Ví dụ như, trong C++ có ba toán tử làm việc với các thành viên là ::, . , và ->. Để biết khi nào dùng ba toán tử này cũng phức tạp và dễ nhầm lẫn. Trong C#, chúng được thay thế với một toán tử duy nhất gọi là . (dot). Đối với người mới học thì điều này và những việc cải tiến khác làm bớt nhầm lẫn và đơn giản hơn.

✍*Ghi chú*: Nếu chúng ta đã sử dụng Java và tin rằng nó đơn giản, thì chúng ta cũng sẽ tìm thấy rằng C# cũng đơn giản. Hầu hết mọi người đều không tin rằng Java là ngôn ngữ đơn giản. Tuy nhiên, C# thì dễ hơn là Java và C++.

C# là ngôn ngữ hiện đại

Điều gì làm cho một ngôn ngữ hiện đại? Những đặc tính như là xử lý ngoại lệ, thu gom bộ nhớ tự động, những kiểu dữ liệu mở rộng, và bảo mật mã nguồn là những đặc tính được mong đợi trong một ngôn ngữ hiện đại. C# chứa tất cả những đặc tính trên. Nếu là người mới học lập trình có thể chúng ta sẽ cảm thấy những đặc tính trên phức tạp và khó hiểu. Tuy nhiên, cũng đừng lo lắng chúng ta sẽ dần dần được tìm hiểu những đặc tính qua các chương trong cuốn sách này.

✍*Ghi chú*: Con trỏ được tích hợp vào ngôn ngữ C++. Chúng cũng là nguyên nhân gây ra những rắc rối của ngôn ngữ này. C# loại bỏ những phức tạp và rắc rối phát sinh bởi con trỏ. Trong C#, bộ thu gom bộ nhớ tự động và kiểu dữ liệu an toàn được tích hợp vào ngôn ngữ, sẽ loại bỏ những vấn đề rắc rối của C++.

C# là ngôn ngữ hướng đối tượng

Những đặc điểm chính của ngôn ngữ hướng đối tượng (Object-oriented language) là sự đóng gói (encapsulation), sự kế thừa (inheritance), và đa hình (polymorphism). C# hỗ trợ tất

cả những đặc tính trên. Phần hướng đối tượng của C# sẽ được trình bày chi tiết trong một chương riêng ở phần sau.

C# là ngôn ngữ mạnh mẽ và cũng mềm dẻo

Như đã đề cập trước, với ngôn ngữ C# chúng ta chỉ bị giới hạn ở chính bởi bản thân hay là trí tưởng tượng của chúng ta. Ngôn ngữ này không đặt những ràng buộc lên những việc có thể làm. C# được sử dụng cho nhiều các dự án khác nhau như là tạo ra ứng dụng xử lý văn bản, ứng dụng đồ họa, bản tính, hay thậm chí những trình biên dịch cho các ngôn ngữ khác.

C# là ngôn ngữ ít từ khóa

C# là ngôn ngữ sử dụng giới hạn những từ khóa. Phần lớn các từ khóa được sử dụng để mô tả thông tin. Chúng ta có thể nghĩ rằng một ngôn ngữ có nhiều từ khóa thì sẽ mạnh hơn. Điều này không phải sự thật, ít nhất là trong trường hợp ngôn ngữ C#, chúng ta có thể tìm thấy rằng ngôn ngữ này có thể được sử dụng để làm bất cứ nhiệm vụ nào. Bảng sau liệt kê các từ khóa của ngôn ngữ C#.

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

Bảng 1.2: Từ khóa của ngôn ngữ C#.

C# là ngôn ngữ hướng module

Mã nguồn C# có thể được viết trong những phần được gọi là những lớp, những lớp này chứa các phương thức thành viên của nó. Những lớp và những phương thức có thể được sử dụng lại trong ứng dụng hay các chương trình khác. Bằng cách truyền các mẫu thông tin đến những lớp hay phương thức chúng ta có thể tạo ra những mã nguồn dùng lại có hiệu quả.

C# sẽ là một ngôn ngữ phổ biến

C# là một trong những ngôn ngữ lập trình mới nhất. Vào thời điểm cuốn sách này được viết, nó không được biết như là một ngôn ngữ phổ biến. Nhưng ngôn ngữ này có một số lý do để trở thành một ngôn ngữ phổ biến. Một trong những lý do chính là Microsoft và sự cam kết của .NET

Microsoft muốn ngôn ngữ C# trở nên phổ biến. Mặc dù một công ty không thể làm một sản phẩm trở nên phổ biến, nhưng nó có thể hỗ trợ. Cách đây không lâu, Microsoft đã gặp sự thất bại về hệ điều hành Microsoft Bob. Mặc dù Microsoft muốn Bob trở nên phổ biến nhưng thất bại. C# thay thế tốt hơn để đem đến thành công sơ với Bob. Thật sự là không biết khi nào mọi người trong công ty Microsoft sử dụng Bob trong công việc hằng ngày của họ. Tuy nhên, với C# thì khác, nó được sử dụng bởi Microsoft. Nhiều sản phẩm của công ty này đã chuyển đổi và viết lại bằng C#. Bằng cách sử dụng ngôn ngữ này Microsoft đã xác nhận khả năng của C# cần thiết cho những người lập trình.

Micorosoft .NET là một lý do khác để đem đến sự thành công của C#. .NET là một sự thay đổi trong cách tạo và thực thi những ứng dụng.

Ngoài hai lý do trên ngôn ngữ C# cũng sẽ trở nên phổ biến do những đặc tính của ngôn ngữ này được đề cập trong mục trước như: đơn giản, hướng đối tượng, mạnh mẽ...

Ngôn ngữ C# và những ngôn ngữ khác

Chúng ta đã từng nghe đến những ngôn ngữ khác như Visual Basic, C++ và Java. Có lẽ chúng ta cũng tự hỏi sự khác nhau giữa ngôn ngữ C# và nhưng ngôn ngữ đó. Và cũng tự hỏi tại sao lại chọn ngôn ngữ này để học mà không chọn một trong những ngôn ngữ kia. Có rất nhiều lý do và chúng ta hãy xem một số sự so sánh giữa ngôn ngữ C# với những ngôn ngữ khác giúp chúng ta phần nào trả lời được những thắc mắc.

Microsoft nói rằng C# mang đến sức mạnh của ngôn ngữ C++ với sự dễ dàng của ngôn ngữ Visual Basic. Có thể nó không dễ như Visual Basic, nhưng với phiên bản Visual Basic.NET (Version 7) thì ngang nhau. Bởi vì chúng được viết lại từ một nền tảng. Chúng ta có thể viết nhiều chương trình với ít mã nguồn hơn nếu dùng C#.

Mặc dù C# loại bỏ một vài các đặc tính của C++, nhưng bù lại nó tránh được những lỗi mà thường gặp trong ngôn ngữ C++. Điều này có thể tiết kiệm được hàng giờ hay thậm chí hàng ngày trong việc hoàn tất một chương trình. Chúng ta sẽ hiểu nhiều về điều này trong các chương của giáo trình.

Một điều quan trọng khác với C++ là mã nguồn C# không đòi hỏi phải có tập tin header. Tất cả mã nguồn được viết trong khai báo một lớp.

Như đã nói ở bên trên. .NET runtime trong C# thực hiện việc thu gom bộ nhớ tự động. Do điều này nên việc sử dụng con trỏ trong C# ít quan trọng hơn trong C++. Những con trỏ cũng có thể được sử dụng trong C#, khi đó những đoạn mã nguồn này sẽ được đánh dấu là không an toàn (unsafe code).

C# cũng từ bỏ ý tưởng đa kế thừa như trong C++. Và sự khác nhau khác là C# đưa thêm thuộc tính vào trong một lớp giống như trong Visual Basic. Và những thành viên của lớp được gọi duy nhất bằng toán tử "." khác với C++ có nhiều cách gọi trong các tình huống khác nhau.

Một ngôn ngữ khác rất mạnh và phổ biến là Java, giống như C++ và C# được phát triển dựa trên C. Nếu chúng ta quyết định sẽ học Java sau này, chúng ta sẽ tìm được nhiều cái mà học từ C# có thể được áp dụng.

Điểm giống nhau C# và Java là cả hai cùng biên dịch ra mã trung gian: C# biên dịch ra MSIL còn Java biên dịch ra bytecode. Sau đó chúng được thực hiện bằng cách thông dịch hoặc biên dịch just-in-time trong từng máy ảo tương ứng. Tuy nhiên, trong ngôn ngữ C# nhiều hỗ trợ được đưa ra để biên dịch mã ngôn ngữ trung gian sang mã máy. C# chứa nhiều kiểu dữ liệu cơ bản hơn Java và cũng cho phép nhiều sự mở rộng với kiểu dữ liệu giá trị. Ví dụ, ngôn ngữ C# hỗ trợ kiểu liệt kệ (enumerator), kiểu này được giới hạn đến một tập hằng được định nghĩa trước, và kiểu dữ liệu cấu trúc đây là kiểu dữ liệu giá trị do người dùng định nghĩa. Chúng ta sẽ được tìm hiểu kỹ hơn về kiểu dữ liệu tham chiếu và kiểu dữ liệu giá trị sẽ được trình bày trong phần sau

Tương tự như Java, C# cũng từ bỏ tính đa kế thừa trong một lớp, tuy nhiên mô hình kế thừa đơn này được mở rộng bởi tính đa kế thừa nhiều giao diện.

Các bước chuẩn bị cho chương trình

Thông thường, trong việc phát triển phần mềm, người phát triển phải tuân thủ theo quy trình phát triển phần mềm một cách nghiêm ngặt và quy trình này đã được chuẩn hóa. Tuy nhiên trong phạm vi của chúng ta là tìm hiểu một ngôn ngữ mới và viết những chương trình nhỏ thì không đòi hỏi khắt khe việc thực hiện theo quy trình. Nhưng để giải quyết được những vấn đề thì chúng ta cũng cần phải thực hiện đúng theo các bước sau. Đầu tiên là phải xác định vấn đề cần giải quyết. Nếu không biết rõ vấn đề thì ta không thể tìm được phương pháp giải quyết. Sau khi xác định được vấn đề, thì chúng ta có thể nghĩ ra các kế hoạch để thực hiện. Sau khi có một kế hoạch, thì có thể thực thi kế hoạch này. Sau khi kế hoạch được thực thi, chúng ta phải kiểm tra lại kết quả để xem vấn đề được giải quyết xong chưa. Logic này thường được áp dụng trong nhiều lĩnh vực khác nhau, trong đó có lập trình.

Khi tạo một chương trình trong C# hay bất cứ ngôn ngữ nào, chúng ta nên theo những bước tuần tự sau:

- Xác định mục tiêu của chương trình.
- Xác định những phương pháp giải quyết vấn đề.
- Tạo một chương trình để giải quyết vấn đề.
- Thực thi chương trình để xem kết quả.

Ví dụ mục tiêu để viết chương trình xử lý văn bản đơn giản, mục tiêu chính là xây dựng chương trình cho phép soạn thảo và lưu trữ những chuỗi ký tự hay văn bản. Nếu không có mục tiêu thì không thể viết được chương trình hiệu quả.

Bước thứ hai là quyết định đến phương pháp để viết chương trình. Bước này xác định những thông tin nào cần thiết được sử dụng trong chương trình, các hình thức nào được sử dụng. Từ những thông tin này chúng ta rút ra được phương pháp để giải quyết vấn đề.

Bước thứ ba là bước cài đặt, ở bước này có thể dùng các ngôn ngữ khác nhau để cài đặt, tuy nhiên, ngôn ngữ phù hợp để giải quyết vấn đề một cách tốt nhất sẽ được chọn. Trong phạm vi của sách này chúng ta mặc định là dùng C#, đơn giản là chúng ta đang tìm hiểu nó! Và bước cuối cùng là phần thực thi chương trình để xem kết quả.

Chương trình C# đơn giản

Để bắt đầu cho việc tìm hiểu ngôn ngữ C# và tạo tiền đề cho các chương sau, chương đầu tiên trình bày một chương trình C# đơn giản nhất.

Ví dụ 2.1 : Chương trình C# đầu tiên.

```
---------------------------------------------------------------------------
class ChaoMung
{
    static void Main( )
    {
        // Xuat ra man hinh
        System.Console.WriteLine("Chao Mung");
    }
}
---------------------------------------------------------------------------
```

Kết quả:

```
Chao Mung
---------------------------------------------------------------------------
```

Sau khi viết xong chúng ta lưu dưới dạng tập tin có phần mở rộng *.cs (C sharp). Sau đó biên dịch và chạy chương trình. Kết quả là một chuỗi "`Chao Mung`" sẽ xuất hiện trong màn hình console.

Các mục sau sẽ giới thiệu xoay quanh ví dụ 2.1, còn phần chi tiết từng loại sẽ được trình bày trong các chương kế tiếp.

Lớp, đối tượng và kiểu dữ liệu (type)

Điều cốt lõi của lập trình hướng đối tượng là tạo ra các kiểu mới. Kiểu là một thứ được xem như trừu tượng. Nó có thể là một bảng dữ liệu, một tiểu trình, hay một nút lệnh trong một cửa sổ. Tóm lại kiểu được định nghĩa như một dạng vừa có thuộc tính chung (properties) và các hành vi ứng xử (behavior) của nó.

Nếu trong một ứng dụng trên Windows chúng ta tạo ra ba nút lệnh OK, Cancel, Help, thì thực chất là chúng ta đang dùng ba thể hiện của một kiểu nút lệnh trong Windows và các nút này cùng chia xẻ các thuộc tính và hành vi chung với nhau. Ví dụ, các nút có các thuộc tính như kích thước, vị trí, nhãn tên (label), tuy nhiên mỗi thuộc tính của một thể hiện không nhất thiết phải giống nhau, và thường thì chúng khác nhau, như nút OK có nhãn là "OK", Cancel có nhãn là "Cancel"...Ngoài ra các nút này có các hành vi ứng xử chung như khả năng vẽ, kích hoạt, đáp ứng các thông điệp nhấn,…Tùy theo từng chức năng đặc biệt riêng của từng loại thì nội dung ứng xử khác nhau, nhưng tất cả chúng được xem như là cùng một kiểu.

Cũng như nhiều ngôn ngữ lập trình hướng đối tượng khác, kiểu trong C# được định nghĩa là một lớp (class), và các thể hiện riêng của từng lớp được gọi là đối tượng (object). Trong các chương kế tiếp sẽ trình bày các kiểu khác nhau ngoài kiểu lớp như kiểu liệt kê, cấu trúc và kiểu ủy quyền (delegates).

Quay lại chương trình ChaoMung trên, chương trình này chỉ có một kiểu đơn giản là lớp ChaoMung. Để định nghĩa một kiểu lớp trong C# chúng ta phải dùng từ khoá class, tiếp sau là tên lớp trong ví dụ trên tên lớp là ChaoMung. Sau đó định nghĩa các thuộc tính và hành động cho lớp. Thuộc tính và hành động phải nằm trong dấu { }.

✍*Ghi chú*: Khai báo lớp trong C# không có dấu ; sau ngoặc } cuối cùng của lớp. Và khác với lớp trong C/C++ là chia thành 2 phần header và phần định nghĩa. Trong C# , định nghĩa một lớp được gói gọn trong dấu { } sau tên lớp và trong cùng một tập tin.

Phương thức

Hai thành phần chính cấu thành một lớp là thuộc tính hay tính chất và phương thức hay còn gọi là hành động ứng xử của đối tượng. Trong C# hành vi được định nghĩa như một phương thức thành viên của lớp.

Phương thức chính là các hàm được định nghĩa trong lớp. Do đó, ta còn có thể gọi các phương thức thành viên là các hàm thành viên trong một lớp. Các phương thức này chỉ ra rằng các hành động mà lớp có thể làm được cùng với cách thức làm hành động đó. Thông thường, tên của phương thức thường được đặt theo tên hành động, ví dụ như DrawLine() hay GetString().

Tuy nhiên trong ví dụ 2.1 vừa trình bày, chúng ta có hàm thành viên là Main() hàm này là hàm đặc biệt, không mô tả hành động nào của lớp hết, nó được xác định là hàm đầu vào của lớp (entry point) và được CRL gọi đầu tiên khi thực thi.

✍*Ghi chú*: Trong C#, hàm Main() được viết ký tự hoa đầu, và có thể trả về giá trị void hay int

Khi chương trình thực thi, CLR gọi hàm Main() đầu tiên, hàm Main() là đầu vào của chương trình, và mỗi chương trình phải có một hàm Main(). Đôi khi chương trình có nhiều hàm Main() nhưng lúc này ta phải xác định các chỉ dẫn biên dịch để CLR biết đâu là hàm Main() đầu vào duy nhất trong chương trình.

Việc khai báo phương thức được xem như là một sự giao ước giữa người tạo ra lớp và người sử dụng lớp này. Người xây dựng các lớp cũng có thể là người dùng lớp đó, nhưng không hoàn toàn như vậy. Vì có thể các lớp này được xây dựng thành các thư viện chuẩn và cung cấp cho các nhóm phát triển khác…Do vậy việc tuân thủ theo các qui tắc lá rất cần thiết.

Để khai báo một phương thức, phải xác định kiểu giá trị trả về, tên của phương thức, và cuối cùng là các tham số cần thiết cho phương thức thực hiện.

Chú thích

Một chương trình được viết tốt thì cần phải có chú thích các đoạn mã được viết. Các đoạn chú thích này sẽ không được biên dịch và cũng không tham gia vào chương trình. Mục đích chính là làm cho đoạn mã nguồn rõ ràng và dễ hiểu.

Trong ví dụ 2.1 có một dòng chú thích :

```
// Xuat ra man hinh.
```

Một chuỗi chú thích trên một dòng thì bắt đầu bằng ký tự "//". Khi trình biên dịch gặp hai ký tự này thì sẽ bỏ qua dòng đó.

Ngoài ra C# còn cho phép kiểu chú thích cho một hay nhiều dòng, và ta phải khai báo "/*" ở phần đầu chú thích và kết thúc chú thích là ký tự "*/".

Ví dụ 2.2 : Minh họa dùng chú thích trên nhiều dòng.

```
---------------------------------------------------------------------------
class ChaoMung
{
        static void Main()
        {
            /* Xuat ra man hinh chuoi 'chao mung'
             Su dung ham WriteLine cua lop System.Console
            */
            System.Console.WriteLine("Chao Mung");
        }
}
---------------------------------------------------------------------------
```

Kết quả:

```
Chao Mung
---------------------------------------------------------------------------
```

Ngoài hai kiểu chú thích trên giống trong C/C++ thì C# còn hỗ trợ thêm kiểu thứ ba cũng là kiểu cuối cùng, kiểu này chứa các định dạng XML nhằm xuất ra tập tin XML khi biên dịch để tạo sưu liệu cho mã nguồn. Chúng ta sẽ bàn kiểu này trong các chương trình ở các phần tiếp.

Ứng dụng Console

Ví dụ đơn giản trên được gọi là ứng dụng console, ứng dụng này giao tiếp với người dùng thông quan bàn phím và không có giao diện người dùng (UI), giống như các ứng dụng thường thấy trong Windows. Trong các chương xây dựng các ứng dụng nâng cao trên Windows hay Web thì ta mới dùng các các giao diện đồ họa. Còn để tìm hiểu về ngôn ngữ C# thuần tuý thì cách tốt nhất là ta viết các ứng dụng console.

Trong hai ứng dụng đơn giản trên ta đã dùng phương thức WriteLine() của lớp Console. Phương thức này sẽ xuất ra màn hình dòng lệnh hay màn hình DOS chuỗi tham số đưa vào, cụ thể là chuỗi "`Chao Mung`".

Namespace

Như chúng ta đã biết .NET cung cấp một thư viện các lớp đồ sộ và thư viện này có tên là FCL (Framework Class Library). Trong đó Console chỉ là một lớp nhỏ trong hàng ngàn lớp trong thư viện. Mỗi lớp có một tên riêng, vì vậy FCL có hàng ngàn tên như ArrayList, Dictionary, FileSelector,…

Điều này làm nảy sinh vấn đề, người lập trình không thể nào nhớ hết được tên của các lớp trong .NET Framework. Tệ hơn nữa là sau này có thể ta tạo lại một lớp trùng với lớp đã có chẳng hạn. Ví dụ trong quá trình phát triển một ứng dụng ta cần xây dựng một lớp từ điển và lấy tên là Dictionary, và điều này dẫn đến sự tranh chấp khi biên dịch vì C# chỉ cho phép một tên duy nhất.

Chắc chắn rằng khi đó chúng ta phải đổi tên của lớp từ điển mà ta vừa tạo thành một cái tên khác chẳng hạn như myDictionary. Khi đó sẽ làm cho việc phát triển các ứng dụng trở nên phức tạp, cồng kềnh. Đến một sự phát triển nhất định nào đó thì chính là cơn ác mộng cho nhà phát triển.

Giải pháp để giải quyết vấn đề này là việc tạo ra một namespace**,** namsespace sẽ hạn chế phạm vi của một tên, làm cho tên này chỉ có ý nghĩa trong vùng đã định nghĩa.

Giả sử có một người nói Tùng là một kỹ sư, từ kỹ sư phải đi kèm với một lĩnh vực nhất định nào đó, vì nếu không thì chúng ta sẽ không biết được là anh ta là kỹ sư cầu đường, cơ khí hay phần mềm. Khi đó một lập trình viên C# sẽ bảo rằng Tùng là CauDuong.KySu phân biệt với CoKhi.KySu hay PhanMem.KySu. Namespace trong trường hợp này là CauDuong, CoKhi, PhanMem sẽ hạn chế phạm vi của những từ theo sau. Nó tạo ra một vùng không gian để tên sau đó có nghĩa.

Tương tự như vậy ta cứ tạo các namespace để phân thành các vùng cho các lớp trùng tên không tranh chấp với nhau.

Tương tự như vậy, .NET Framework có xây dựng một lớp Dictionary bên trong namespace System.Collections, và tương ứng ta có thể tạo một lớp Dictionary khác nằm trong namespace ProgramCSharp.DataStructures, điều này hoàn toàn không dẫn đến sự tranh chấp với nhau.

Trong ví dụ minh họa 1.2 đối tượng Console bị hạn chế bởi namespace bằng việc sử dụng mã lệnh:

```
System.Console.WriteLine();
```

Toán tử '.'

Trong vídụ 2.2 trên dấu '.' được sử dụng để truy cập đến phương thức hay dữ liệu trong một lớp (trong trường hợp này phương thức là WriteLine()), và ngăn cách giữa tên lớp đến một namespace xác nhận (namspace System và lớp là Console). Việc thực hiện này theo hướng từ trên xuống, trong đó mức đầu tiên namespace là System, tiếp theo là lớp Console, và cuối cùng là truy cập đến các phương thức hay thuộc tính của lớp.

Trong nhiều trường hợp namespace có thể được chia thành các namespace con gọi là subnamespace. Ví dụ trong namespace System có chứa một số các subnamesapce như Configuration, Collections, Data, và còn rất nhiều nữa, hơn nữa trong namespace Collection còn chia thành nhiều namesapce con nữa.

Namespace giúp chúng ta tổ chức và ngăn cách những kiểu. Khi chúng ta viết một chương trình C# phức tạp, chúng ta có thể phải tạo một kiến trúc namespace riêng cho mình, và không giới hạn chiều sâu của cây phân cấp namespace. Mục đích của namespace là giúp chúng ta chia để quản lý những kiến trúc đối tượng phức tạp.

Từ khóa using

Để làm cho chương trình gọn hơn, và không cần phải viết từng namespace cho từng đối tượng, C# cung cấp từ khóa là using, sau từ khóa này là một namespace hay subnamespace với mô tả đầy đủ trong cấu trúc phân cấp của nó.

Ta có thể dùng dòng lệnh :

```
using System;
```

ở đầu chương trình và khi đó trong chương trình nếu chúng ta có dùng đối tượng Console thì không cần phải viết đầy đủ : System.Console. mà chỉ cần viết Console. thôi.

Ví dụ 2.3: Dùng khóa using

```
---------------------------------------------------------------------------
using System;
class ChaoMung
{
    static void Main()
    {
        //Xuat ra man hinh chuoi thong bao
        Console.WriteLine("Chao Mung");
    }
}
---------------------------------------------------------------------------
```

Kết quả:

```
Chao Mung
```

Lưu ý rằng phải đặt câu using System trước định nghĩa lớp ChaoMung.
Mặc dù chúng ta chỉ định rằng chúng ta sử dụng namespace System, và không giống như các ngôn ngữ khác, không thể chỉ định rằng chúng ta sử dụng đối tượng System.Console.

Ví dụ 2.4: Không hợp lệ trong C#.

```
using System.Console;
class ChaoMung
{
    static void Main()
    {
        //Xuat ra man hinh chuoi thong bao
        WriteLine("Chao Mung");
    }
}
```

Đoạn chương trình trên khi biên dịch sẽ được thông báo một lỗi như sau:
error CS0138: A using namespace directive can only be applied to namespace;
'System.Console' is a class not a namespace.
Cách biểu diễn namespace có thể làm giảm nhiều thao tác gõ bàn phím, nhưng nó có thể sẽ không đem lại lợi ích nào bởi vì nó có thể làm xáo trộn những namespace có tên không khác nhau. Giải pháp chung là chúng ta sử dụng từ khóa **using** với các namespace đã được xây dựng sẵn, các namespace do chúng ta tạo ra, những namespace này chúng ta đã nắm chắc sưu liệu về nó. Còn đối với namespace do các hãng thứ ba cung cấp thì chúng ta không nên dùng từ khóa **using**.

Phân biệt chữ thường và chữ hoa

Cũng giống như C/C++, C# là ngôn ngữ phân biệt chữ thường với chữ hoa, điều này có nghĩa rằng hai câu lệnh writeLine thì khác với WriteLine và cũng khác với WRITELINE.
Đáng tiếc là C# không giống như VB, môi trường phát triển C# sẽ không tự sửa các lỗi này, nếu chúng ta viết hai chữ với cách khác nhau thì chúng ta có thể đưa vào chương trình gỡ rối tìm ra các lỗi này.

Để trách việc lãnh phí thời gian và công sức, người ta phát triển một số qui ước cho cách đặt tên biến, hằng, hàm, và nhiều định danh khác nữa. Qui ước trong giáo trình này dùng cú pháp lạc đà (camel notation) cho tên biến và cú pháp Pascal cho hàm, hằng, và thuộc tính.
Ví dụ :

Biến myDictionary theo cách đặt tên cú pháp lạc đà.
Hàm DrawLine, thuộc tính ColorBackground theo cách đặt tên cú pháp Pascal.

Từ khóa static

Hàm Main() trong ví dụ minh họa trên có nhiều hơn một cách thiết kế. Trong minh họa này hàm Main() được khai báo với kiểu trả về là void, tức là hàm này không trả về bất cứ giá trị nào cả. Đôi khi cần kiểm tra chương trình có thực hiện đúng hay không, người lập trình có thể khai báo hàm Main() trả về một giá trị nào đó để xác định kết quả thực hiện của chương trình. Trong khai báo của ví dụ trên có dùng từ khóa **static**:

```
static void Main()
{
    .......
}
```

Từ khóa này chỉ ra rằng hàm Main() có thể được gọi mà không cần phải tạo đối tượng ChaoMung. Những vấn đề liên quan đến khai báo lớp, phương thức, hay thuộc tính sẽ được trình bày chi tiết trong các chương tiếp theo.

Phát triển chương trình minh họa

Có tối thiểu là hai cách để soạn thảo, biên dịch và thực thi chương trình trong cuốn sách này:

- Sử dụng môi trường phát triển tích hợp (IDE) Visual Studio .NET
- Sử dụng chương trình soạn thảo văn bản bất kỳ như Notepad rồi dùng biên dịch dòng lệnh.

Mặc dù chúng ta có thể phát triển phần mềm bên ngoài Visual Studio .NET, IDE cung cấp nhiều các tiện ích hỗ trợ cho người phát triển như: hỗ trợ phần soạn thảo mã nguồn như canh lề, màu sắc, tích hợp các tập tin trợ giúp, các đặc tính intellisense,...Nhưng điều quan trọng nhất là IDE phải có công cụ debug mạnh và một số công cụ trợ giúp phát triển ứng dụng khác.

Trong cuốn sách này giả sử rằng người đọc đang sử dụng Visual Studio .NET. Phần trình này sẽ tập trung vào ngôn ngữ và platform hơn là công cụ phát triển. Chúng ta có thể sao chép tất cả những mã nguồn ví dụ vào trong một chương trình soạn thảo văn bản như Notepad hay Emacs, lưu chúng dưới dạng tập tin văn bản, và biên dịch chúng bằng trình biên dịch dòng lệnh C#, chương trình này được phân phối cùng .NET Framework SDK. Trong những chương cuối về xây dựng các ứng dụng trên Windows và Web, chúng ta sẽ sử dụng công cụ Visual Studio .NET để tạo ra các Windows Form và Web Form, tuy nhiên chúng ta cũng có thể viết bằng tay trong Notepad nếu chúng ta quyết định sử dụng cách làm bằng tay thay vì dùng công cụ thiết kế.

Sử dụng Notepad soạn thảo

Đầu tiên chúng ta sẽ mở chương trình Notepad rồi soạn thảo chương trình minh họa trên, lưu ý là ta có thể sử dụng bất cứ trình soạn thảo văn bản nào chứ không nhất thiết là Notepad. Sau khi soạn thảo xong thì lưu tập tin xuống đĩa và tập tin này có phần mở rộng là *.cs, trong ví dụ này là chaomung.cs. Bước tiếp theo là biên dịch tập tin nguồn vừa tạo ra. Để biên dịch ta dùng trình biên dịch dòng lệnh C# (csc.exe) chương trình này được chép vào máy trong quá trình cài .NET Framework. Để biết csc.exe nằm chính xác vị trí nào trong đĩa ta có thể dùng chức năng tìm kiếm của Windows.

Để thực hiện biên dịch chúng ta mở một cửa sổ dòng lệnh rồi đánh vào lệnh theo mẫu sau:

```
csc.exe [/out: <file thực thi>]  <file nguồn>
```

Ví dụ: csc.exe /out:d:\chaomung.exe d:\chaomung.cs

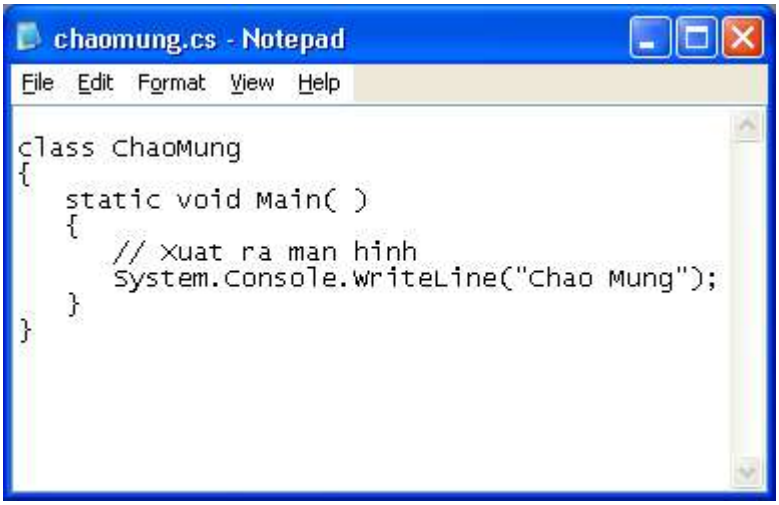
Thường thì khi biên dịch ta chỉ cần hai phần là tên của trình biên dịch và tên tập tin nguồn mà thôi. Trong mẫu trên có dùng một trong nhiều tùy chọn khi biên dịch là /out, theo sau là tên của chương trình thực thi hay chính là kết quả biên dịch tập tin nguồn.

Các tham số tùy chọn có rất nhiều nếu muốn tìm hiểu chúng ta có thể dùng lệnh:

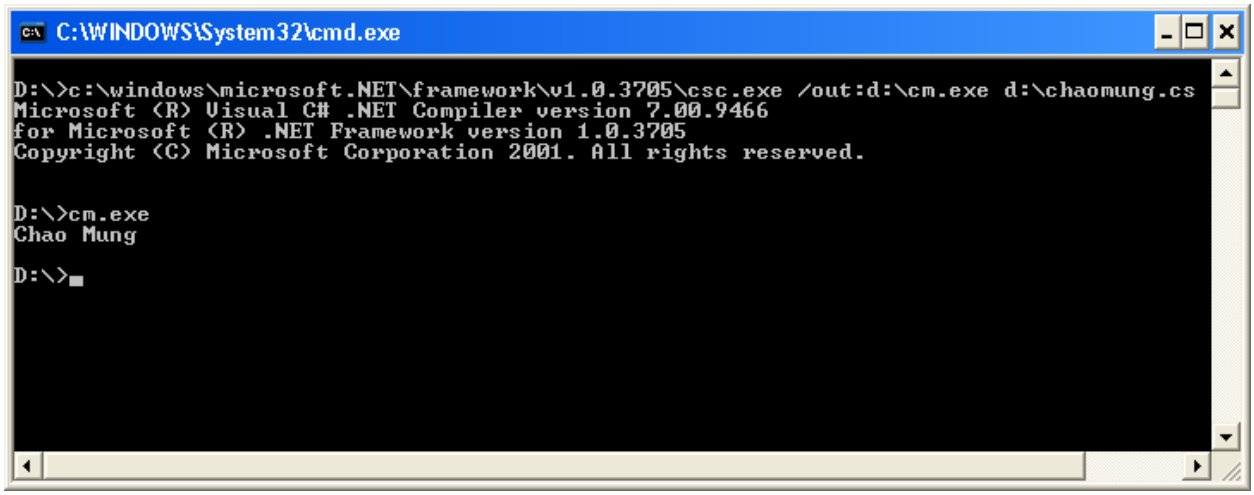
```
csc.exe /?
```

Lệnh này xuất ra màn hình toàn bộ các tùy chọn biên dịch và các hướng dẫn sử dụng.

Hai hình sau minh họa quá trình nhập mã nguồn chương trình C# bằng một trình soạn thảo văn bản đơn giản như Notepad trong Windows. Và sau đó biên dịch tập tin mã nguồn vừa tạo ra bằng chương trình csc.exe một trình biên dịch dòng lệnh của C#. Kết quả là một tập tin thực thi được tạo ra và ta sẽ chạy chương trình này.



Hình 2.2: Mã nguồn được soạn thảo trong Notepad.



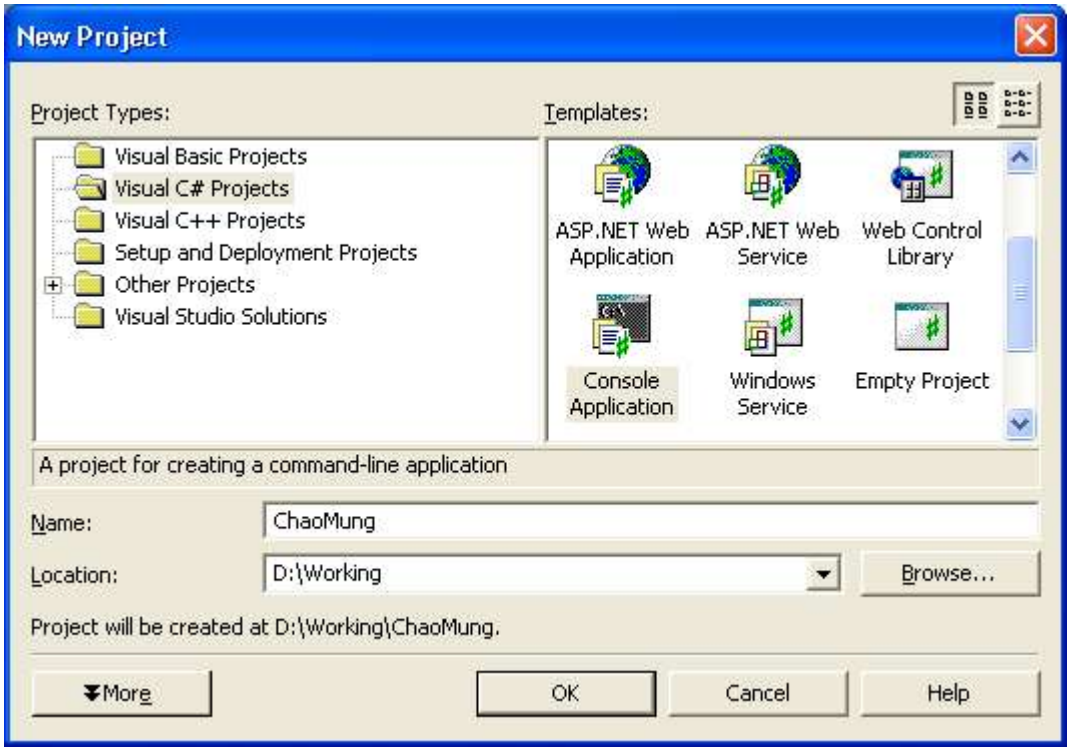
Hình 2.3: Biên dịch và thực thi chương trình.

Sử dụng Visual Studio .NET để tạo chương trình

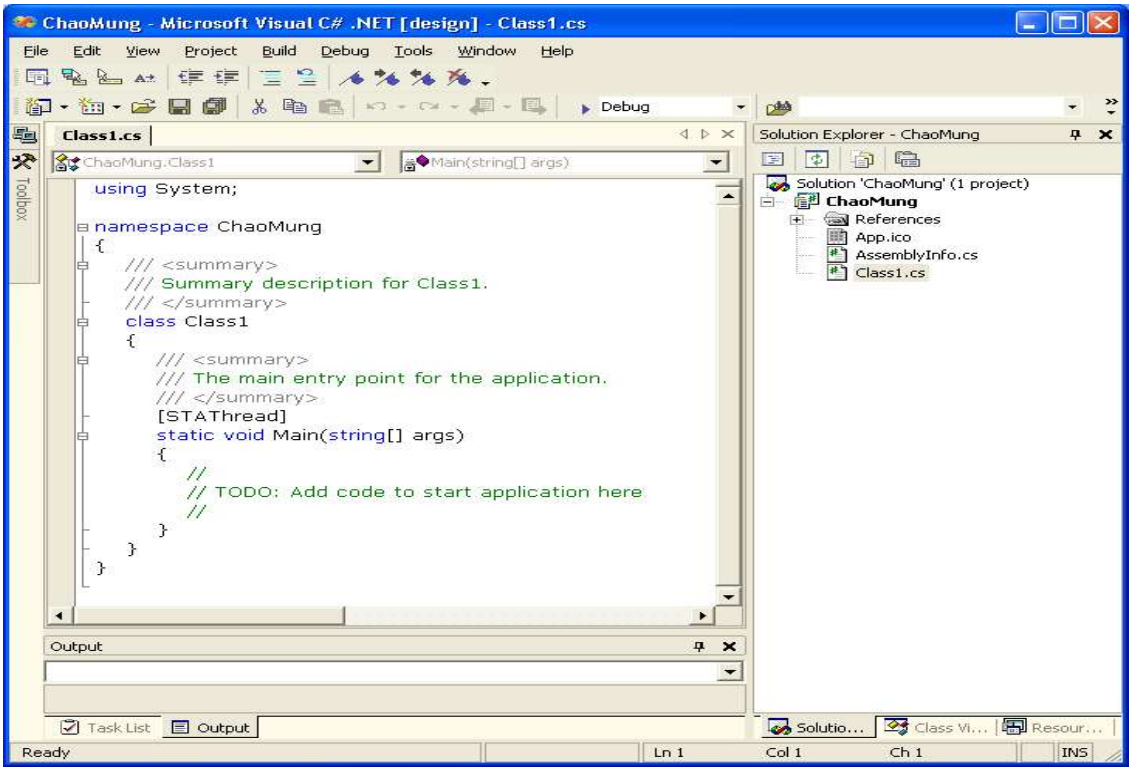
Để tạo chương trình chào mừng trong IDE, lựa chọn mục Visual Studio .NET trong menu Start hoặc icon của nó trên desktop, sau khi khởi động xong chương trình, chọn tiếp chức năng File→ New → Project trong menu. Chức năng này sẽ gọi cửa sổ New Project (hình 2.4 bên dưới). Nếu như chương trình Visual Studio .NET được chạy lần đầu tiên, khi đó cửa sổ New Project sẽ xuất hiện tự động mà không cần phải kích hoạt.

Để tạo ứng dụng, ta lựa chọn mục Visual C# Projects trong cửa sổ Project Type bên trái. Lúc này chúng ta có thể nhập tên cho ứng dụng và lựa chọn thư mục nơi lưu trữ các tập tin này. Cuối cùng, kích vào OK khi mọi chuyện khởi tạo đã chấm dứt và một cửa sổ mới sẽ xuất hiện (hình 2.4 bên dưới), chúng ta có thể nhập mã nguồn vào đây.

Lưu ý rằng Visual Studio .NET tạo ra một namespace dựa trên tên của project mà ta vừa cung cấp (ChaoMung), và thêm vào chỉ dẫn sử dụng namespace System bằng lệnh **using**, bởi hầu như mọi chương trình mà chúng ta viết đều cần sử dụng các kiểu dữ liệu chứa trong namespace System.



Hình 2.4: Tạo ứng dụng C# console trong Visual Studio .NET.



Hình 2.5: Phần soạn thảo mã nguồn cho project.

Visual Studio .NET tạo một lớp tên là Class1, lớp này chúng ta có thể tùy ý đổi tên của chúng. Khi đổi tên của lớp, tốt nhất là đổi tên luôn tập tin chứa lớp đó (Class1.cs). Giả sử

trong ví dụ trên chúng ta đổi tên của lớp thành ChaoMung, và đổi tên tập tin Class1.cs (đổi tên tập tin trong cửa sổ Solution Explorer).

Cuối cùng, Visual Studio .NET tạo một khung sườn chương trình, và kết thúc với chú thích TODO là vị trí bắt đầu của chúng ta. Để tạo chương trình chào mừng trong minh họa trên, ta bỏ tham số string[] args của hàm Main() và xóa tất cả các chú thích bên trong của hàm. Sau đó nhập vào dòng lệnh sau bên trong thân của hàm Main()

```
// Xuat ra man hinh
System.Console.WriteLine("Chao Mung");
```

Sau tất cả công việc đó, tiếp theo là phần biên dịch chương trình từ Visual Studio .NET. Thông thường để thực hiện một công việc nào đó ta có thể chọn kích hoạt chức năng trong menu, hay các button trên thanh toolbar, và cách nhanh nhất là sử dụng các phím nóng hay các phím kết hợp để gọi nhanh một chức năng.

Trong ví dụ, để biên dịch chương trình nhấn *Ctrl-Shift-B* hoặc chọn chức năng:
Build → Build Solution. Một cách khác nữa là dùng nút lệnh trên thanh toolbar:

Để chạy chương trình vừa được tạo ra mà không sử dụng chế độ debug chúng ta có thể nhấn *Ctrl-F5* hay chọn Debug → Start Without Debugging hoặc nút lệnh trên thanh toolbar của Visual Studio . NET

✍*Ghi chú*: Tốt hơn hết là chúng ta nên bỏ ra nhiều thời gian để tìm hiểu hay khám phá môi trường phát triển Visual Studio .NET. Đây cũng là cách thức tốt mà những người phát triển ứng dụng và chúng ta nên thực hiện. Việc tìm hiểu Visual Studio .NET và thông thạo nó sẽ giúp cho chúng ta rất nhiều trong quá trình xây dựng và phát triển ứng dụng sau này.

Câu hỏi và trả lời

Câu hỏi 1: Một chương trình C# có thể chạy trên bất cứ máy nào?

Trả lời 1: Không phải tất cả. Một chương trình C# chỉ chạy trên máy có Common Language Runtime (CLR) được cài đặt. Nếu chúng ta copy một chương trình exe của C# qua một máy không có CLR thì chúng ta sẽ nhận được một lỗi. Trong những phiên bản của Windows không có CLR chúng ta sẽ được bảo rằng thiếu tập tin DLL.

Câu hỏi 2: Nếu muốn đưa chương trình mà ta viết cho một người bạn thì tập tin nào mà chúng ta cần đưa?

*Trả lời 2: Thông thường cách tốt nhất là đưa chương trình đã biên dịch. Điều này có nghĩa rằng sau khi mã nguồn được biên dịch, chúng ta sẽ có một chương trình thực thi (tập tin có phần mở rộng *.exe). Như vậy, nếu chúng ta muốn đưa chương trình Chaomung cho tất cả những người bạn của chúng ta thì chỉ cần đưa tập tin Chaomung.exe. Không cần thiết phải đưa tập tin nguồn Chaomung.cs. Và những người bạn của chúng ta không cần thiết phải có trình biên dịch C#. Họ chỉ cần có C# runtime trên máy tính (như CLR của Microsoft) là có thể chạy được chương trình của chúng ta.*

Câu hỏi 3: Sau khi tạo ra được tập tin thực thi .exe. Có cần thiết giữ lại tập tin nguồn không?

Trả lời 3: Nếu chúng ta từ bỏ tập tin mã nguồn thì sau này sẽ rất khó khăn cho việc mở rộng hay thay đổi chương trình, do đó cần thiết phải giữ lại các tập tin nguồn. Hầu hết các IDE tạo ra các các tập tin nguồn (.cs) và các tập tin thực thi. Cũng như giữ các tập tin nguồn chúng ta cũng cần thiết phải giữ các tập tin khác như là các tài nguyên bên ngoài các icon, image, form.. Chúng ta sẽ lưu giữ những tập tin này trong trường hợp chúng ta cần thay đổi hay tạo lại tập tin thực thi.

Câu hỏi 4: Nếu trình biên dịch C# đưa ra một trình soạn thảo, có phải nhất thiết phải sử dụng nó?

Trả lời 4: Không hoàn toàn như vậy. Chúng ta có thể sử dụng bất cứ trình soạn thảo văn bản nào và lưu mã nguồn dưới dạng tập tin văn bản. Nếu trình biên dịch đưa ra một trình soạn thảo thì chúng ta nên sử dụng nó. Nếu chúng ta có môt trình soạn thảo khác tốt hơn chúng ta có thể sử dụng nó. Một số các tiện ích soạn thảo mã nguồn có thể giúp cho ta dễ dàng tìm các lỗi cú pháp, giúp tạo một số mã nguồn tự động đơn giản...Nói chung là tùy theo chúng ta nhưng theo tôi thì Visual Studio .NET cũng khá tốt để sử dụng

Câu hỏi 5: Có thể không quan tâm đến những cảnh báo khi biên dịch mã nguồn

Trả lời 5: Một vài cảnh báo không ảnh hưởng đến chương trình khi chạy, nhưng một số khác có thể ảnh hưởng đến chương trình chạy. Nếu trình biên dịch đưa ra cảnh báo, tức là tín hiệu cho một thứ gì đó không đúng. Hầu hết các trình biên dịch cho phép chúng ta thiết lập mức độ cảnh báo. Bằng cách thiết lập mức độ cảnh báo chúng ta có thể chỉ quan tâm đến những cảnh báo nguy hiểm, hay nhận hết tất cả những cảnh báo. Nói chung cách tốt nhất là chúng ta nên xem tất cả những cảnh báo để sửa chữa chúng, một chương trình tạm gọi là đạt yêu cầu khi không có lỗi biên dịch và cũng không có cảnh báo (nhưng chưa chắc đã chạy đúng kết quả!).

Câu hỏi thêm

Câu hỏi 1: Hãy đưa ra 3 lý do tại sao ngôn ngữ C# là một ngôn ngữ lập trình tốt?

Câu hỏi 2: IL và CLR viết tắt cho từ nào và ý nghĩa của nó?

Câu hỏi 3: Đưa ra các bước cơ bản trong chu trình xây dựng chương trình?

Câu hỏi 4: Trong biên dịch dòng lệnh thì lệnh nào được sử dụng để biên dịch mã nguồn .cs và lệnh này gọi chương trình nào?

Câu hỏi 5: Phần mở rộng nào mà chúng ta nên sử dụng cho tập tin mã nguồn C#?

Câu hỏi 6: Một tập tin .txt chứa mã nguồn C# có phải là một tập tin mã nguồn C# hợp lệ hay không? Có thể biên dịch được hay không?

Câu hỏi 7: Ngôn ngữ máy là gì? Khi biên dịch mã nguồn C# ra tập tin .exe thì tập tin này là ngôn ngữ gì?

Câu hỏi 8: Nếu thực thi một chương trình đã biên dịch và nó không thực hiện đúng như mong đợi của chúng ta, thì điều gì chúng ta cần phải làm?

Câu hỏi 9: Một lỗi tương tự như bên dưới thường xuất hiện khi nào?

mycode.cs(15,5): error CS1010: NewLine in constan

Câu hỏi 10: Tại sao phải khai báo static cho hàm Main của lớp?

Câu hỏi 11: Một mã nguồn C# có phải chứa trong các lớp hay là có thể tồn tại bên ngoài lớp như C/C++?

Câu hỏi 12: So sánh sự khác nhau cơ bản giữa C# và C/C++, C# với Java, hay bất cứ ngôn ngữ cấp cao nào mà bạn đã biết?

Câu hỏi 13: Con trỏ có còn được sử dụng trong C# hay không? Nếu có thì nó được quản lý như thế nào?

Câu hỏi 14: Khái niệm và ý nghĩa của namespace trong C#? Điều gì xảy ra nếu như ngôn ngữ lập trình không hỗ trợ namespace?

Bài tập

Bài tập 1: Dùng trình soạn thảo văn bản mở chương trình exe mà ta đã biên dịch từ các chương trình nguồn trước và xem sự khác nhau giữa hai tập tin này, lưu ý sao khi đóng tập tin này ta không chọn lưu tập tin.

Bài tập 2: Nhập vào chương trình sau và biên dịch nó. Cho biết chương trình thực hiện điều gì?

```
---------------------------------------------------------------------------
using System;
class variables
{
    public static void Main()
    {
        int radius = 4;
        const double PI = 3.14159;
        double circum, area;
        area = PI * radius* radius;
        circum = 2 * PI * radius;
        // in kết quả
        Console.WriteLine("Ban kinh = {0}, PI = {1}", radius, PI);
        Console.WriteLine("Dien tich {0}", area);
        Console.WriteLine("Chu vi {0}", circum);
    }
}
---------------------------------------------------------------------------
```

Bài tập 3: Nhập vào chương trình sau và biên dịch. Cho biết chương trình thực hiện điều gì?

```
---------------------------------------------------------------------------
class AClass
```

```
{
    static void Main()
    {
        int x, y;
        for( x = 0; x < 10; x++, System.Console.Write("\n"));
        for( y = 0 ; y < 10; y++, System.Console.WriteLine("{0}",y));
    }
}
```

Bài tập 4: Chương trình sau có chứa lỗi. Nhập vào và sửa những lỗi đó

Bài tập 5: Sửa lỗi và biên dịch chương trình sau

```
class Test
{
    pubic static void Main()
    {

        Console.WriteLine("Xin chao");
        Consoile.WriteLine("Tam biet");
    }
}
```

Bài tập 6: Sửa lỗi và biên dịch chương trình sau

```
class Test
{
    pubic void Main()
    {
        Console.WriteLine('Xin chao');
        Consoile.WriteLine('Tam biet');
    }
}
```

Bài tập 7: Viết chương trình xuất ra bài thơ:

Rằm Tháng Giêng

Rằm xuân lồng lộng trăng soi,
Sông xuân nước lẫn màu trời thêm xuân.
Giữa dòng bàn bạc việc quân
Khuya về bát ngát trăng ngân đầy thuyền.

Hồ Chí Minh.

Chương 3

NỀN TẢNG NGÔN NGỮ C#

- **Kiểu dữ liệu**
    - **Kiểu dữ liệu xây dựng sẵn**
    - **Chọn kiểu dữ liệu**
    - **Chuyển đổi các kiểu dữ liệu**
- **Biến và hằng**
    - **Gán giá trị xác định cho biến**
    - **Hằng**
    - **Kiểu liệt kê**
    - **Kiểu chuỗi ký tự**
    - **Định danh**
- **Biểu thức**
- **Khoảng trắng**
- **Câu lệnh**
    - **Phân nhánh không có điều kiện**
    - **Phân nhánh có điều kiện**
    - **Câu lệnh lặp**
- **Toán tử**
- **Namespace**
- **Các chỉ dẫn biên dịch**
- **Câu hỏi & bài tập**

Trong chương trước chúng ta đã tìm hiểu một chương trình C# đơn giản nhất. Chương trình đó chưa đủ để diễn tả một chương trình viết bằng ngôn ngữ C#, có quá nhiều phần và chi tiết đã bỏ qua. Do vậy trong chương này chúng ta sẽ đi sâu vào tìm hiểu cấu trúc và cú pháp của ngôn ngữ C#.

Chương này sẽ thảo luận về hệ thống kiểu dữ liệu, phân biệt giữa kiểu dữ liệu xây dựng sẵn (như int, bool, string…) với kiểu dữ liệu do người dùng định nghĩa (lớp hay cấu trúc do người lập trình tạo ra...). Một số cơ bản khác về lập trình như tạo và sử dụng biến dữ liệu hay hằng cũng được đề cập cùng với cấu trúc liệt kê, chuỗi, định danh, biểu thức và cậu lệnh.

Trong phần hai của chương hướng dẫn và minh họa việc sử dụng lệnh phân nhánh **if, switch**, **while**, **do**...**while**, **for**, và **foreach**. Và các toán tử như phép gán, phép toán logic, phép toán quan hệ, và toán học...

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.

✍*Ghi chú*: Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu 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 3.1 sau sẽ mô tả một số các kiểu dữ liệu được 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 đến 32767.
ushort	2	Uint16	Số nguyên không dấu 0 – 65.535
int	4	Int32	Số nguyên có dấu –2.147.483.647 và 2.147.483.647
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 đến 9.223.372.036.854.775.807
ulong	8	Uint64	Số nguyên không dấu từ 0 đến 0xffffffffffffffff

Bảng 3.1 : Mô tả các kiểu dữ liệu xây dựng sẵn.

✍*Ghi chú:* 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 hũy 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 hũy 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.

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

Bảng 3.2 : Các kiểu ký tự đặc biệt.

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
```

Biến và hằng

Một biến là một vùng lưu trữ với một kiểu dữ liệu. Trong ví dụ trước cả x, và y điều là biến. Biến có thể được gán giá trị và cũng có thể thay đổi giá trị khi thực hiện các lệnh trong chương trình.

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 3.1 sau minh họa sử dụng biến.

Ví dụ 3.1: 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 3.2:

Ví dụ 3.2: 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 3.2 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 3.3 minh họa một chương trình đúng.

Ví dụ 3.3: 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. Ví dụ:

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

Trong khai báo trên, 32 là một hằng số và DoSoi là một biểu tượng hằng có kiểu nguyên. Ví dụ 3.4 minh họa việc sử dụng những biểu tượng hằng.

Ví dụ 3.4: 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 3.4 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 3.4 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 3.4, 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,
}
```

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

Ta viết lại ví dụ minh họa 3-4 như sau.

Ví dụ 3.5: 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"
```

Kiểu chuỗi sẽ được đề cập sâu trong chương 10.

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.

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;
```

Khoảng trắng (whitespace)

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, củng 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.

Câu lệnh (statement)

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.

Ví dụ 3.6: 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 3.7 bên dưới cách sử dụng câu lệnh **if**...**else**.

Ví dụ 3.7: 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 3.7 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 3.8 sau sẽ minh họa cho việc thực hiện các yêu cầu trên.

*Ví dụ 3.8: 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
        gioiTinh = true; // nam

        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 3.9 dưới đây.

Ví dụ 3.9: 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 3.9 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 từng câu lệnh **case** mà không quan tâm đến giá trị biến chonMau.

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 mauVang case mauLuc	Ban chon mau luc
mauLuc	case mauLuc	Ban chon mau luc
mauLam	case mauLam case mauCham case mauTim case mauLuc	Ban chon mau lam Ban chon mau cham Ban chon mau tim Ban chon mau luc
mauCham	case mauCham case mauTim case mauLuc	Ban chon mau cham Ban chon mau tim Ban chon mau luc
mauTim	case mauTim case mauLuc	Ban chon mau tim Ban chon mau luc

Bảng 3.3: Mô tả các trường hợp thực hiện câu lệnh switch.

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!
--------------------------------------------------------------------------------
```

✍*Ghi chú*: 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.

Ví dụ 3.10: 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 3.11 minh họa việc sử dụng vòng lặp **while**.

Ví dụ 3.11: 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 3.11 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*".

Lưu ý rằng 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.

Ví dụ 3.12: Minh hoạ việc sử dụng vòng lặp do..while.

```
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:

```
i: 11
```

Do khởi tạo biến i giá trị là 11, nên điều kiện của **while** là sai, tuy nhiên vòng lặp **do**...**while** vẫn được thực hiện một lần.

- *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:

Ví dụ 3.13: Sử dụng vòng lặp for.

```
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);
            }
        }
        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.

Ví dụ 3.14: 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**.

Ví dụ 3.15 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 3.16 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.

*Ví dụ 3.16: 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.

Toán tử

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

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 (%). Ví dụ, câu lệnh sau `8%3` thì kết quả trả về là 2 (đây là phần dư còn lại của phép chia nguyên).

Thật sự phép toán chia lấy dư rất hữu dụng cho người lập trình . Khi chúng ta thực hiện một phép chia dư n cho một số khác, nếu số này là bội số của n thì kết quả của phép chia dư là 0. Ví dụ `20 % 5 = 0` vì 20 là một bội số của 5. Điều này cho phép chúng ta ứng dụng trong

vòng lặp, khi muốn thực hiện một công việc nào đó cách khoảng n lần, ta chỉ cần kiểm tra phép chia dư n, nếu kết quả bằng 0 thì thực hiện công việc. Cách sử dụng này đã áp dụng trong ví dụ minh họa sử dụng vòng lặp **for** bên trên. Ví dụ 3.17 sau minh họa sử dụng các phép toán chia trên các số nguyên, thực...

Ví dụ 3.17: 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:

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

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

```
Luong = Luong * 2;
Luong = Luong – 100.000;
…
```

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

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

Bảng 3.4: Mô tả các phép toán tự gán.

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

```
Luong  += 200.000;
Luong  *= 2;
Luong  -=  100.000;
```

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

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

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

```
bienDem++;
```

- *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 3.18 sau

Ví dụ 3.18: 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.

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

Bảng 3.4: Các toán tử so sánh (giả sử value1 = 100, và value2 = 50).

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

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

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

Bảng 3.5: Các toán tử logic (giả sử x = 5, y = 7).

Toán tử and sẽ kiểm tra cả hai điều kiện. Trong bảng 3.5 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 3.6: Liệt kê thứ tự độ ưu tiên các phép toán trong C#.

STT	Loại toán tử	Toán tử	Thứ tự
1	Phép toán cơ bản	(x) x.y f(x) a[x] x++ x—new typeof sizeof 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 logic AND	&	Trái
9	Phép toán logic XOR	^	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

Bảng 3.6: Thứ tự ưu tiên các toán tử.

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 3.19 sau.

Ví dụ 3.19: 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.

Namespace

Chương 2 đã thảo luận việc sử dụng đặc tính *namespace* trong ngôn ngữ C#, nhằm tránh sự xung đột giữa việc sử dụng các thư viện khác nhau từ các nhà cung cấp. Ngoài ra, namespace được xem như là tập hợp các lớp đối tượng, và cung cấp duy nhất các định danh cho các kiểu dữ liệu và được đặt trong một cấu trúc phân cấp. 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** để 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 3.20 minh họa việc tạo một namespace.

Ví dụ 3.20: 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 3.21.

Ví dụ 3.21: 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 3.21 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.

Ví dụ 3.22: 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 3.22 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.

Các chỉ dẫn 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.

Định nghĩa định danh

Câu lệnh tiền xử lý sau:

```
#define DEBUG
```

Lệnh trên định nghĩa một định danh tiền xử lý có tên là DEBUG. Mặc dù những chỉ thị tiền xử lý khác có thể được đặt bất cứ ở đâu trong chương trình, nhưng với chỉ thị định nghĩa định danh thì phải đặt trước tất cả các lệnh khác, bao gồm cả câu lệnh using.

Để kiểm tra một định danh đã được định nghĩa thì ta dùng cú pháp #if <định danh>. Do đó ta có thể viết như sau:

```
#define DEBUG
//...Các đoạn mã nguồn bình thường, không bị tác động bởi trình tiền xử lý
...
#if DEBUG
    // Các đoạn mã nguồn trong khối if debug được biên dịch
#else
    // Các đoạn mã nguồn không định nghĩa debug và không được biên dịch
#endif
//...Các đoạn mã nguồn bình thường, không bị tác động bởi trình tiền xử lý
```

Khi chương trình tiền xử lý thực hiện, chúng sẽ tìm thấy câu lệnh #define DEBUG và lưu lại định danh DEBUG này. Tiếp theo trình tiền xử lý này sẽ bỏ qua tất cả các đoạn mã bình thường khác của C# và tìm các khối #if, #else, và #endif.

Câu lệnh #if sẽ kiểm tra định danh DEBUG, do định danh này đã được định nghĩa, nên đoạn mã nguồn giữa khối #if đến #else sẽ được biên dịch vào chương trình. Còn đoạn mã nguồn giữa #else và #endif sẽ không được biên dịch. Tức là đoạn mã nguồn này sẽ không được thực hiện hay xuất hiện bên trong mã hợp ngữ của chương trình.

Trường hợp câu lệnh #if sai tức là không có định nghĩa một định danh DEBUG trong chương trình, khi đó đoạn mã nguồn ở giữa khối #if và #else sẽ không được đưa vào chương trình để biên dịch mà ngược lại đoạn mã nguồn ở giữa khối #else và #endif sẽ được biên dịch.

✍*Lưu ý*: Tất cả các đoạn mã nguồn bên ngoài #if và #endif thì không bị tác động bởi trình tiền xử lý và tất cả các mã này đều được đưa vào để biên dịch.

Không định nghĩa định danh

Sử dụng chỉ thị tiền xử lý #undef để xác định trạng thái của một định danh là không được định nghĩa. Như chúng ta đã biết trình tiền xử lý sẽ thực hiện từ trên xuống dưới, do vậy một định danh đã được khai báo bên trên với chỉ thị #define sẽ có hiệu quả đến khi một gọi câu lệnh #undef định danh đó hay đến cuối chương trình:

```
#define DEBUG
#if DEBUG
    // Đoạn code này được biên dịch
#endif
....
#undef DEBUG
....
#if DEBUG
    // Đoạn code này không được biên dịch
    #endif
.....
```

#if đầu tiên đúng do DEBUG được định nghĩa, còn #if thứ hai sai không được biên dịch vì DEBUG đã được định nghĩa lại là #undef.

Ngoài ra còn có chỉ thị #elif và #else cung cấp các chỉ dẫn phức tạp hơn. Chỉ dẫn #elif cho phép sử dụng logic "else-if". Ta có thể diễn giải một chỉ dẫn như sau: "*Nếu DEBUG thì làm công việc 1, ngược lại nếu TEST thì làm công việc 2, nếu sai tất cả thì làm trường hợp 3*":

```
....
#if DEBUG
      // Đoạn code này được biên dịch nếu DEBUG được định nghĩa
#elif TEST
      //Đoạn code này được biên dịch nếu DEBUG không được định nghĩa
      // và TEST được định nghĩa
#else
      //Đoạn code này được biên dịch nếu cả DEBUG và
       //TEST không được định nghĩa.
#endif
....
```

Trong ví dụ trên thì chỉ thị tiền xử lý #if đầu tiên sẽ kiểm tra định danh DEBUG, nếu định danh DEBUG đã được định nghĩa thì đoạn mã nguồn ở giữa #if và #elif sẽ được biên dịch, và tất cả các phần còn lại cho đến chỉ thị #endif đều không được biên dịch. Nếu DEBUG không được định nghĩa thì #elif sẽ kiểm tra định danh TEST, đoạn mã ở giữa #elif và #else sẽ được

thực thi khi TEST được định nghĩa. Cuối cùng nếu cả hai DEBUG và TEST đều không được định nghĩa thì các đoạn mã nguồn giữa #else và #endif sẽ được biên dịch.

Câu hỏi và trả lời

Câu hỏi 1: Sự khác nhau giữa dựa trên thành phần (Component-Based) và hướng đối tượng (Object- Oriented)?

Trả lời 1: Phát triển dựa trên thành phần có thể được xem như là mở rộng của lập trình hướng đối tượng. Một thành phần là một khối mã nguồn riêng có thể thực hiện một nhiệm vụ đặc biệt. Lập trình dựa trên thành phần bao gồm việc tạo nhiều các thành phần tự hoạt động có thể được dùng lại. Sau đó chúng ta có thể liên kết chúng lại để xây dựng các ứng dụng.

Câu hỏi 2: Những ngôn ngữ nào khác được xem như là hướng đối tượng?

Trả lời 2: Các ngôn ngữ như là C++, Java, SmallTalk, Visual Basic.NET cũng có thể được sử dụng cho lập trình hướng đối tượng. Còn rất nhiều những ngôn ngữ khác nhưng không được phổ biến lắm.

Câu hỏi 3: Tại sao trong kiểu số không nên khai báo kiểu dữ liệu lớn thay vì dùng kiểu dữ liệu nhỏ hơn?

Trả lời 3: Mặc dù điều có thể xem là khá hợp lý, nhưng thật sự không hiệu quả lắm. Chúng ta không nên sử dụng nhiều tài nguyên bộ nhớ hơn mức cần thiết. Khi đó vừa lãng phí bộ nhớ lại vừa hạn chế tốc độ của chương trình.

Câu hỏi 4: Chuyện gì xảy ra nếu ta gán giá trị âm vào biến kiểu không dấu?

Trả lời 4: Chúng ta sẽ nhận được lỗi của trình biên dịch nói rằng không thể gán giá trị âm cho biến không dấu trong trường hợp ta gán giá trị hằng âm. Còn nếu trong trường hợp kết quả là âm được tính trong biểu thức khi chạy chương trình thì chúng ta sẽ nhận được lỗi dữ liệu. Việc kiểm tra và xử lý lỗi dữ liệu sẽ đươc trình bày trong các phần sau.

Câu hỏi 5: Những ngôn ngữ nào khác hỗ trở Common Type System (CTS) trong Common Language Runtime (CLR)?

Trả lời 5: Microsoft Visual Basic (Version 7), Visual C++.NET cùng hỗ trợ CTS. Thêm vào đó là một số phiên bản của ngôn ngữ khác cũng được chuyển vào CTS. Bao gồm Python, COBOL, Perl, Java. Chúng ta có thể xem trên trang web của Microsoft để biết thêm chi tiết.

Câu hỏi 6: Có phải còn những câu lệnh điều khiển khác?

Trả lời 6: Đúng, các câu lệnh này như sau: throw, try, catch và finally. Chúng ta sẽ được học trong chương xử lý ngoại lệ.

Câu hỏi 7: Có thể sử dụng chuỗi với câu lệnh switch?

Trả lời 7: Hoàn toàn được, chúng ta sử dụng biến giá trị chuỗi trong ***switch*** *rồi sau đó dùng giá trị chuỗi trong câu lệnh case. Lưu ý là chuỗi là những ký tự đơn giản nằm giữa hai dấu ngoặc nháy.*

Câu hỏi thêm

Câu hỏi 1: *Có bao nhiêu cách khai báo comment trong ngôn ngữ C#, cho biết chi tiết?*

Câu hỏi 2: *Những từ theo sau từ nào là từ khóa trong C#: field, cast, as, object, throw, football, do, get, set, basketball.*

Câu hỏi 3: *Những khái niệm chính của ngôn ngữ lập trình hướng đối tượng?*

Câu hỏi 4: *Sự khác nhau giữa hai lệnh Write và WriteLine?*

Câu hỏi 5: *C# chia làm mấy kiểu dữ liệu chính? Nếu ta tạo một lớp tên myClass thì lớp này được xếp vào kiểu dữ liệu nào?*

Câu hỏi 6: *Kiểu chuỗi trong C# là kiểu dữ liệu nào?*

Câu hỏi 7: *Dữ liệu của biến kiểu dữ liệu tham chiếu được lưu ở đâu trong bộ nhớ?*

Câu hỏi 8: *Sự khác nhau giữa lớp và cấu trúc trong C#? Khi nào thì dùng cấu trúc tốt hơn là dùng class?*

Câu hỏi 8: *Sự khác nhau giữa kiểu unsigned và signed trong kiểu số nguyên?*

Câu hỏi 9: *Kiểu dữ liệu nào nhỏ nhất có thể lưu trữ được giá trị 45?*

Câu hỏi 10: *Số lớn nhất, và nhỏ nhất của kiểu int là số nào?*

Câu hỏi 11: *Có bao nhiêu bit trong một byte?*

Câu hỏi 12: *Kiểu dữ liệu nào trong .NET tương ứng với kiểu int trong C#?*

Câu hỏi 13: *Những từ khóa nào làm thay đổi luồng của chương trình?*

Câu hỏi 14: *Kết quả của 15%4 là bao nhiêu?*

Câu hỏi 15: *Sự khác nhau giữa chuyển đổi tường minh và chuyển đổi ngầm định?*

Câu hỏi 16: *Có thể chuyển từ một giá trị long sang giá trị int hay không?*

Câu hỏi 17: *Số lần tối thiểu các lệnh trong* ***while*** *được thực hiện?*

Câu hỏi 18: *Số lần tối thiểu các lệnh trong do* ***while*** *được thực hiện?*

Câu hỏi 19: *Lệnh nào dùng để thoát ra khỏi vòng lặp?*

Câu hỏi 20: *Lệnh nào dùng để qua vòng lặp kế tiếp?*

Câu hỏi 21: *Khi nào dùng biến và khi nào dùng hằng?*

Câu hỏi 22: *Cho biết giá trị CanhCut trong kiểu liệt kê sau:*

```
enum LoaiChim
{
    HaiAu,
    BoiCa,
    DaiBang = 50,
    CanhCut
}
```

Câu hỏi 23: *Cho biết các lệnh phân nhánh trong C#?*

Bài tập

Bài tập 1: *Nhập vào, biên dịch và chạy chương trình. Hãy cho biết chương trình làm điều gì?*

```
class BaiTap3_1
{
    public static void Main()
    {
        int x = 0;
        for(x = 1; x < 10; x++)
        {
            System.Console.Write("{0:03}", x);
        }
    }
}
```

Bài tập 2: Tìm lỗi của chương trình sau? sửa lỗi và biên dịch chương trình.

```
class BaiTap3_2
{
    public static void Main()
    {
        for(int i=0; i < 10 ; i++)
            System.Console.WriteLine("so :{1}", i);
    }
}
```

Bài tập 3: Tìm lỗi của chương trình sau. Sửa lỗi và biên dịch lại chương trình.

```
using System;
class BaiTap3_3
{
    public static void Main()
    {
        double myDouble;
        decimal myDecimal;
        myDouble = 3.14;
        myDecimal = 3.14;
        Console.WriteLine("My Double: {0}", myDouble);
        Console.WriteLine("My Decimal: {0}", myDecimal);
    }
}
```

Bài tập 4: Tìm lỗi của chương trình sau. Sửa lỗi và biên dịch lại chương trình.

```
class BaiTap3_4
{
    static void Main()
    {
        int value;
        if (value > 100);
            System.Console.WriteLine("Number is greater than 100");
    }
}
```

Bài tập 5: Viết chương trình hiển thị ra màn hình 3 kiểu sau:

```
*                    $ $ $ $ $ $                  *
* *                  $ $ $ $ $                  * * *
* * *                $ $ $ $                  * * * * *
* * * *              $ $ $                  * * * * * * *
* * * * *            $ $                  * * * * * * * * *
* * * * * *          $                  * * * * * * * * * * *
```

a) b) c)

Bài tập 6: Viết chương trình hiển ra trên màn hình.

1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
6 7 8 9 0 1 0 9 8 7 6
7 8 9 0 1 2 3 2 1 0 9 8 7
8 9 0 1 2 3 4 5 4 3 2 1 0 9 8
9 0 1 2 3 4 5 6 7 6 5 4 3 2 1 0 9
0 1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1 0

<u>Bài tập 7</u>: Viết chương trình in ký tự số (0..9) và ký tự chữ (a..z) với mã ký tự tương ứng của từng ký tự

Ví dụ:

'0' : 48

'1' : 49

....

<u>Bài tập 8</u>: Viết chương trình giải phương trình bậc nhất, cho phép người dùng nhập vào giá trị a, b.

<u>Bài tập 9</u>: Viết chương trình giải phương trình bậc hai, cho phép người dùng nhập vào giá trị a, b, c.

<u>Bài tập 10</u>: Viết chương trình tính chu vi và diện tích của các hình sau: đường tròn, hình chữ nhật, hình thang, tam giác.

Chương 4

XÂY DỰNG LỚP - ĐỐI TƯỢNG

- **Định nghĩa lớp**
    - **Thuộc tính truy cập**
    - **Tham số của phương thức**
- **Tạo đối tượng**
    - **Bộ khởi dựng**
    - **Khởi tạo biến thành viên**
    - **Bộ khởi dựng sao chép**
    - **Từ khóa this**
- **Sử dụng các thành viên static**
    - **Gọi phương thức static**
    - **Sử dụng bộ khởi dựng static**
    - **Sử dụng bộ khởi dựng private**
    - **Sử dụng thuộc tính static**
- **Hủy đối tượng**
- **Truyền tham số**
- **Nạp chồng phương thức**
- **Đóng gói dữ liệu với thành phần thuộc tính**
- **Thuộc tính chỉ đọc**
- **Câu hỏi & bài tập**

Chương 3 thảo luận rất nhiều kiểu dữ liệu cơ bản của ngôn ngữ C#, như int, long and char. Tuy nhiên trái tim và linh hồn của C# là khả năng tạo ra những kiểu dữ liệu mới, phức

tạp. 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 4 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 ({}).

✍*Ghi chú*: 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:

Ví dụ 4.1: 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	Giới hạn truy cập
public	Không hạn chế. Những thành viên được đánh dấu public 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 **private** thì chỉ được truy cập bởi các phương thức của lớp A.
protected	Thành viên trong lớp A được đánh dấu là **protected** thì 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 lớp nào 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 lớp nào trong cùng khối hợp ngữ của A.

Bảng 4.1: Thuộc tính truy cập.

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 trong ví dụ 4.1 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.

🗗 *Ví dụ 4.2: 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 3 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 4.1, 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 4.1 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.

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

Bảng 4.2: Giá trị mặc định của kiểu dữ liệu cơ bản.

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 4.3 được viết lại từ ví dụ 4.1 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.

Ví dụ 4.3: 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 4.2.

Ngoài ra trong chương trình 4.3 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.

Ví dụ 4.4: 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:

```
Hien tai:   5/6/2002   10:15:5
Thoi Gian:  5/6/2002   10:15:5

Hien tai:   5/6/2002   10:15:5
Thoi Gian:  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.

- Các thứ ba sử dụng tham chiếu **this** là *mảng chỉ mục* (indexer), phần này sẽ được trình bày chi tiết trong chương 9.

Sử dụng các thành viên tĩnh (static member)

Những thuộc tính và phương thức trong một lớp có thể là những thành viên thể hiện (instance members) hay những thành viên tĩnh (static members). Những thành viên thể hiện hay thành viên của đối tượng liên quan đến thể hiện của một kiểu dữ liệu. Trong khi thành viên tĩnh được xem như một phần của lớp. 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();
```

✍*Ghi chú*: 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.

✍*Ghi chú*: 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 4.2:

```
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 cấn 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.

✍*Ghi chú*: 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.

Theo ví dụ 4.4 ta có thể thêm một bộ khởi dựng tĩnh cho lớp ThoiGian như sau:

```
static ThoiGian()
{
    Ten = "Thoi gian";
}
```

Lưu ý rằng 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:

Ví dụ 4.5: 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.

Hủy 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.

Ví dụ 4.6: 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
    }
}
```

Trong phần khai báo đầu của ví dụ thì đối tượng Font được khai báo bên trong câu lệnh **using**. Khi câu lệnh **using** kết thúc, thì phương thức Dispose của đối tượng Font sẽ được gọi.

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.

Truyền tham số

Như đã thảo luận trong chương trước, tham số có kiểu dữ liệu là giá trị thì sẽ được truyền giá trị vào cho phương thức. Điều này có nghĩa rằng khi một đối tượng có kiểu là giá trị được truyền vào cho một phương thức, thì có một bản sao chép đối tượng đó được tạo ra bên trong phương thức. Một khi phương thức được thực hiện xong thì đối tượng sao chép này sẽ được hủy. Tuy nhiên, đây chỉ là trường hợp bình thường, ngôn ngữ C# còn cung cấp khả năng cho phép ta truyền các đối tượng có kiểu giá trị dưới hình thức là tham chiếu. 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.

Ví dụ 4.7: 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 4.7 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 4.8 sau sử dụng ba kiểu tham số trên.

Ví dụ 4.8: 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

Thông thường khi xây dựng các lớp, ta có mong muốn là tạo ra nhiều hàm có cùng tên. Cũng như hầu hết trong các ví dụ trước thì các lớp điều có nhiều hơn một phương thức khởi dựng. Như trong lớp Time có các phương thức khởi dựng nhận các tham số khác nhau, như tham số là đối tượng DateTime, hay tham số có thể được tùy chọn để thiết lập các giá trị của các biến thành viên thông qua các tham số nguyên. Tóm lại ta có thể xây dựng nhiều các phương thức cùng tên nhưng nhận các tham số khác nhau. Chức năng này được gọi là 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 4.9 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.

Ví dụ 4.9: 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;
    }
    // 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 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.

Ví dụ 4.10: 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.

Đóng gói dữ liệu với thành phần 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 4.11 bên dưới:

Ví dụ 4.11: 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 cặp 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 4.12 minh họa cho cách tiếp cận này.

Ví dụ 4.12: 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.

Câu hỏi và trả lời

Câu hỏi 1: Có phải chúng ta chỉ nên sử dụng lớp với các dữ liệu thành viên?

Trả lời 1: Nói chung là chúng ta không nên sử dụng lớp chỉ với dữ liệu thành viên. Ý nghĩa của môt lớp hay của lập trình hướng đối tượng là khả năng đóng gói các chức năng và dữ liệu vào trong một gói đơn.

Câu hỏi 2: Có phải tất cả những dữ liệu thành viên luôn luôn được khai báo là public để bên ngoài có thể truy cập chúng?

Trả lời 2: Nói chung là không. Do vấn đề che dấu dữ liệu trong lập trình hướng đối tượng, xu hướng là dữ liệu bên trong chỉ nên dùng cho các phương thức thành viên. Tuy nhiên, như chúng ta đã biết khái niệm thuộc tính cho phép các biến thành viên được truy cập từ bên ngoài thông qua hình thức như là phương thức.

Câu hỏi 3: Có phải có rất nhiều lớp được xây dựng sẵn và tôi có thể tìm chúng ở đâu?

Trả lời 3: Microsoft cung cấp rất nhiều các lớp gọi là các lớp cơ sở .NET. Những lớp này được tổ chức bên trong các namespace. Chúng ta có thể tìm tài liệu về các lớp này trong thư viện trực tuyến của Microsoft. Và một số lớp thường sử dụng cũng được trình bày lần lượt trong các ví dụ của giáo trình này.

Câu hỏi 4: Sự khác nhau giữa tham số (parameter) và đối mục (argument)?

Trả lời 4: Tham số được định nghĩa là những thứ được truyền vào trong một phương thức. Một tham số xuất hiện với định nghĩa của phương thức ở đầu phương thức. Một đối mục là giá trị được truyền vào phương thức. Chúng ta truyền những đối mục vào phương thức phù hợp với những tham số đã khai báo của phương thức.

Câu hỏi 5: Chúng ta có thể tạo phương thức bên ngoài của lớp hay không?

Trả lời 5: Mặc dù trong những ngôn ngữ khác, chúng ta có thể tạo các phương thức bên ngoài của lớp. Nhưng trong C# thì không, C# là hướng đối tượng, do vậy tất cả các mã nguồn phải được đặt bên trong một lớp.

Câu hỏi 6: Có phải những phương thức và lớp trong C# hoạt động tương tự như trong các ngôn ngữ khác như C++ hay Java?

Trả lời 6: Trong hầu hết các phần thì chúng tương tự như nhau. Tuy nhiên, mỗi ngôn ngữ cũng có những khác biệt riêng. Một ví dụ sự khác nhau là C# không cho phép tham số mặc định bên trong một phương thức. Trong ngôn ngữ C++ thì chúng ta có thể khai báo các tham số mặc định lúc định nghĩa phương thức và khi gọi phương thức thì có thể không cần truyền giá trị vào, phương thức sẽ dùng giá trị mặc định. Trong C# thì không được phép. Nói chung là còn nhiều sự khác nhau nữa, nhưng xin dành cho bạn đọc tự tìm hiểu.

Câu hỏi 7: Phương thức tĩnh có thể truy cập được thành viên nào và không truy cập được thành viên nào trong một lớp?

Trả lời 7: Phương thức tĩnh chỉ truy cập được các thành viên tĩnh trong một lớp.

Câu hỏi thêm

Câu hỏi 1: Sự khác nhau giữa thành viên được khai báo là public và các thành viên không được khai báo là public?

Câu hỏi 2: Từ khoá nào được sử dụng trong việc thực thi thuộc tính của lớp?

Câu hỏi 3: Những kiểu dữ liệu nào được trả về từ phương thức?

Câu hỏi 4: Sự khác nhau giữa truyền biến tham chiếu và truyền biến tham trị vào một phương thức?

Câu hỏi 5: Làm sao truyền tham chiếu với biến kiểu giá trị vào trong một phương thức?

Câu hỏi 6: Khi nào thì phương thức khởi dựng được gọi?
Câu hỏi 7: Phương thức khởi dựng tĩnh được gọi khi nào?
Câu hỏi 8: Có thể truyền biến chưa khởi tạo vào một hàm được không?
Câu hỏi 9: Sự khác nhau giữa một lớp và một đối tượng của lớp?
Câu hỏi 10: Thành viên nào trong một lớp có thể được truy cập mà không phải tạo thể hiện của lớp?
Câu hỏi 11: Lớp mà chúng ta xây dựng thuộc kiểu dữ liệu nào?
Câu hỏi 12: Từ khóa this được dùng làm gì trong một lớp?

Bài tập

Bài tập 1: Xây dựng một lớp đường tròn lưu giữ bán kính và tâm của đường tròn. Tạo các phương thức để tính chu vi, diện tích của đường tròn.
Bài tập 2: Thêm thuộc tính BanKinh vào lớp được tạo ra từ bài tập 1.
Bài tập 3: Tạo ra một lớp lưu trữ giá trị nguyên tên myNumber. Tạo thuộc tính cho thành viên này. Khi số được lưu trữ thì nhân cho 100. Và khi số được truy cập thì chia cho 100.
Bài tập 4: Chương trình sau có lỗi. Hãy sửa lỗi của chương trình và biên dịch chương trình. Dòng lệnh nào gây ra lỗi?

```
using System;
using System.Console;
class VD1
{
    public string first;
}
class Tester
{
    public static void Main()
    {
        VD1 vd = new VD1();
        Write("Nhap vao mot chuoi: ");
        vd.first = ReadLine();
        Write("Chuoi nhap vao: {0}", vd.first);
    }
}
```

Bài tập 5: Chương trình sau có lỗi. Hãy sửa lỗi của chương trình và biên dịch chương trình. Dòng lệnh nào gây ra lỗi?

```
class Class1
{
    public static void GetNumber(ref int x, ref int y)
    {
        x = 5;
        y = 10;
    }
    public static void Main()
    {
        int a = 0, b = 0;
        GetNumber(a, b);
        System.Console.WriteLine("a = {0} \nb = {1}", a, b);
    }
}
```

Câu hỏi 6: Chương trình sau đây có lỗi. Hãy sửa lỗi và cho biết lệnh nào phát sinh lỗi?

```
Class Tester
{
    public static void Main()
    {
        Display();
    }
    public static void Display()
    {
        System.Console.WriteLine("Hello!");
        return 0;
    }
}
```

Câu hỏi 7: Viết lớp giải phương trình bậc hai. Lớp này có các thuộc tính a, b, c và nghiệm x1, x2. Hãy xây dựng theo hướng đối tượng lớp trên. Lớp cho phép bên ngoài xem được các nghiệm của phương trình và cho phép thiết lập hay xem các giá trị a, b, c.

Chương 5

KẾ THỪA – ĐA HÌNH

- **Đặc biệt hóa và tổng quát hóa**
- **Sự kế thừa**
    - **Thực thi kế thừa**
    - **Gọi phương thức khởi dựng của lớp cơ sở**
    - **Gọi phương thức của lớp cơ sở**
    - **Điều khiển truy xuất**
- **Đa hình**
    - **Kiểu đa hình**
    - **Phương thức đa hình**
    - **Từ khóa new và override**
- **Lớp trừu tượng**
    - **Gốc của tất cả các lớp - lớp Object**
- **Boxing và Unboxing dữ liệu**
    - **Boxing thực hiện ngầm định**
    - **Unboxing phải thực hiện tường minh**
- **Các lớp lồng nhau**
- **Câu hỏi & bài tập**

Trong chương trước đã trình bày cách tạo ra những kiểu dữ liệu mới bằng việc xây dựng các lớp đối tượng. Tiếp theo chương này sẽ đưa chúng ta đi sâu vào mối quan hệ giữa những đối tượng trong thế giới thực và cách mô hình hóa những quan hệ trong xây dựng chương trình. Chương 5 cũng giới thiệu khái niệm đặc biệt hóa (specialization) được cài đặt trong ngôn ngữ C# thông qua sự kế thừa (inheritance).

Khái niệm đa hình (polymorphism) cũng được trình bày trong chương 5, đây là khái niệm quan trọng trong lập trình hướng đối tượng. Khái niệm này cho phép các thể hiện của lớp có liên hệ với nhau có thể được xử lý theo một cách tổng quát.

Cuối cùng là phần trình bày về các lớp cô lập (sealed class) không được đặt biệt hóa, hay các lớp trừu tượng sử dụng trong đặc biệt hóa. Lớp đối tượng Object là gốc của tất cả các lớp cũng được thảo luận ở phần cuối chương.

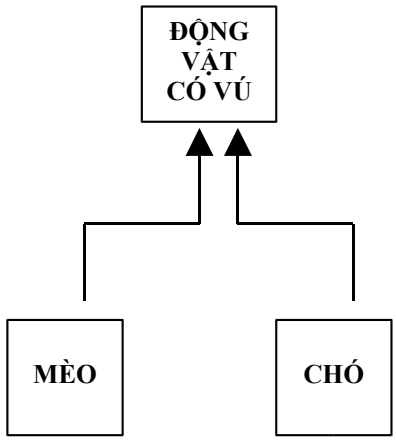
Đặc biệt hóa và tổng quát hóa

Lớp và các thể hiện của lớp tức đối tượng tuy không tồn tại trong cùng một khối, nhưng chúng tồn tại trong một mạng lưới sự phụ thuộc và quan hệ lẫn nhau. Ví dụ như con người và xã hội động vật cùng sống trong một thế giới có quan hệ loài với nhau.

Quan hệ là một (is-a) là một sự *đặc biệt hóa*. Khi chúng ta nói rằng mèo là một loại động vật có vú, có nghĩa là chúng ta đã nói rằng mèo là một trường hợp đặc biệt của loại động vật có vú. Nó có tất cả các đặc tính của bất cứ động vật có vú nào (như sinh ra con, có sữa mẹ và có lông...). Tuy nhiên, mèo có thêm các đặc tính riêng được xác định trong họ nhà mèo mà các họ động vật có vú khác không có được. Chó cũng là loại động vật có vú, chó cũng có tất cả các thuộc tính của động vật có vú, và riêng nó còn có thêm các thuộc tính riêng xác định họ loài chó mà khác với các thuộc tính đặc biệt của loài khác ví dụ như mèo chẳng hạn.

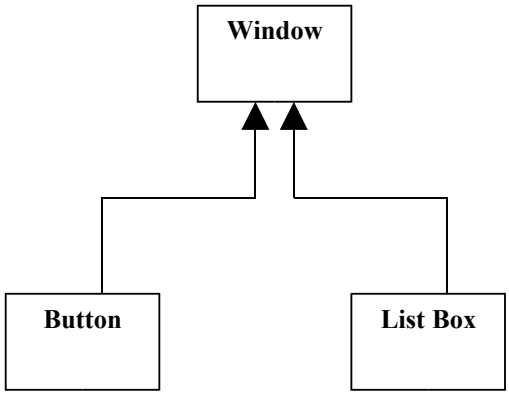
Quan hệ đặc biệt hóa và tổng quát hóa là hai mối quan hệ đối ngẫu và phân cấp với nhau. Chúng có quan hệ đối ngẫu vì đặc biệt được xem như là mặt ngược lại của tổng quát. Do đó, loài chó và mèo là trường hợp đặc biệt của động vật có vú. Ngược lại động vật có vú là trường hợp tổng quát từ các loài chó và mèo.

Mối quan hệ là phân cấp bởi vì chúng ta tạo ra một cây quan hệ, trong đó các trường hợp đặc biệt là những nhánh của trường hợp tổng quát. Trong cây phân cấp này nếu di chuyển lên trên cùng ta sẽ được trường hợp tổng quát hóa, và ngược lại nếu di chuyển xuống ngược nhánh thì ta được trường hợp đặc biệt hóa. Ta có sơ đồ phân cấp minh họa cho loài chó, mèo và động vật có vú như trên:



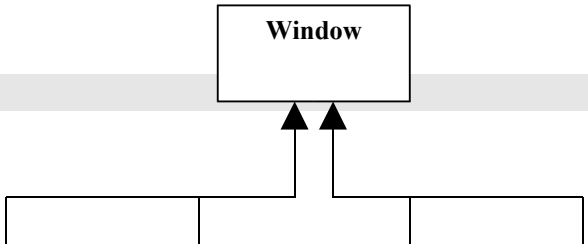
Tương tự, khi chúng ta nói rằng ListBox và Button là những Window, ta phải chỉ ra những đặc tính và hành vi của những Window có trong cả hai lớp trên. Hay nói cách khác, Window là tổng quát hóa chia xẻ những thuộc tính của hai lớp ListBox và Button, trong khi đó mỗi trường hợp đặc biệt ListBox và Button sẽ có riêng những thuộc tính và hành vi đặc thù khác.

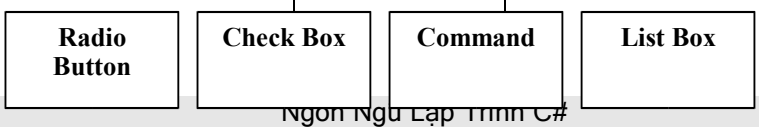
Ngôn ngữ mô hình hóa thống nhất (UML)
UML (Unified Modeling Language) là ngôn ngữ chuẩn hóa để mô tả cho một hệ thống hoặc thương mại. Trong chương này sử dụng một số phần của mô hình UML để trình bày các biểu đồ quan hệ giữa các lớp. Trong UML, những lớp được thể hiện như các khối hộp, tên của lớp được đặt trên cùng của khối hộp, và các phương thức hay các biến thành viên được đặt bên trong hộp. Như trong hình 5.1, mô hình quan hệ tổng quát hóa và đặc biệt hóa được trình bày qua UML, ghi chú rằng mũi tên đi từ các lớp đặc biệt hóa đến lớp tổng quát hóa.



Hình 5.2: Quan hệ giữa thành phần cửa sổ

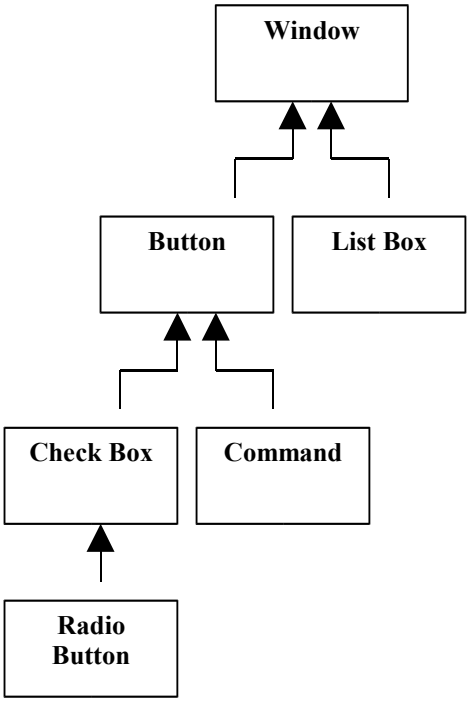
Thông thường lưu ý rằng khi hai lớp chia xẻ chức năng với nhau, thì chúng được trích ra các phần chung và đưa vào lớp cơ sở chia xẻ. Điều này hết sức có lợi, vì nó cung cấp khả năng cao để sử dụng lại các mã nguồn chung và dễ dàng duy trì mã nguồn.





Hình 5.3 Dẫn xuất từ Window

Giả sử chúng ta bắt đầu tạo một loạt các lớp đối tượng theo hình vẽ 5.3 như bên trên. Sau khi làm việc với RadioButton, CheckBox, và CommandButton một thời gian ta nhận thấy chúng chia xẻ nhiều thuộc tính và hành vi đặc biệt hơn Window nhưng lại khá tổng quát cho cả ba lớp này. Như vậy ta có thể chia các thuộc tính và hành vi thành một nhóm lớp cơ sở riêng lấy tên là Button. Sau đó ta sắp xếp lại cấu trúc kế thừa như hình vẽ 5.4. Đây là ví dụ về cách tổng quát hóa được sử dụng để phát triển hướng đối tượng.



Hình 5.4: Cây quan hệ lớp cửa sổ

Trong mô hình UML trên được vẽ lại quan hệ giữa các lớp. Trong đó cả hai lớp Button và ListBox điều dẫn xuất từ lớp Window, trong đó Button có trường hợp đặc biệt là CheckBox và Command. Cuối cùng thì RadioButton được dẫn xuất từ CheckBox. Chúng ta cũng có thể nói rằng RadioButton là một CheckBox, và tiếp tục CheckBox là một Button, và cuối cùng Button là Window.

Sự thiết kế trên không phải là duy nhất hay cách tốt nhất để tổ chức những đối tượng, nhưng đó là khởi điểm để hiểu về cách quan hệ giữa đối tượng với các đối tượng khác.

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.

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 5.1 sau minh họa việc tạo và sử dụng các lớp cơ sở và dẫn xuất.

Ví dụ 5.1: 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, left);
```

```
    }
    // 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 5.1 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.

Gọi phương thức khởi dựng của lớp cơ sở

Trong ví dụ 5.1, 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 5.1 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.

✍*Ghi chú*: 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.

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.

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**.

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.

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.

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++)
{
    winArray[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.

Ví dụ 5.2: 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 DrawWindow
```

```
    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
```

Lưu ý 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.

Ghi chú: 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( ) {…}
```

```
}
```

Lớp trừu tượng

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

Ví dụ 5.3: 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 5.3, 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.

✍*Ghi chú*: 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.

Phương thức	Chức năng
Equal( )	So sánh bằng nhau giữa hai đối tượng
GetHashCode( )	Cho phép những đối tượng cung cấp riêng những 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.

Bảng 5.1: Tóm tắt các phương thức của lớp Object.

Ví dụ 5.4 sau minh họa việc sử dụng phương thức ToString( ) thừa kế từ lớp Object.

Ví dụ 5.4: 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.

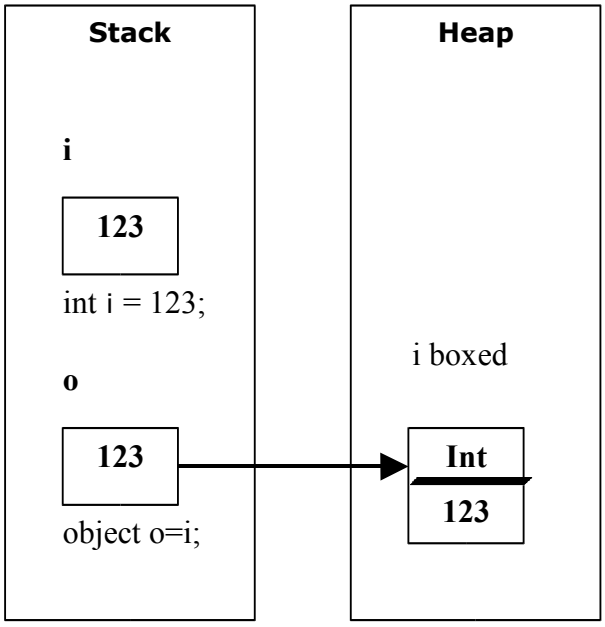
Các lớp không cần phải khai báo tường minh việc dẫn xuất từ lớp Object, việc kế thừa sẽ được đưa vào một cách ngầm định. Như lớp SomeClass trên ta không khai báo bất cứ dẫn xuất của lớp nào nhưng C# sẽ tự động đưa lớp Object thành lớp dẫn xuất. Do đó ta mới có thể phủ quyết phương thức ToString() của lớp Object.

Boxing và Unboxing dữ liệu

Boxing và unboxing là những xử lý cho phép kiểu dữ liệu giá trị (như int, long,...) được đối xử như kiểu dữ liệu tham chiếu (các đối tượng). Một giá trị được đưa vào bên trong của đối tượng, được gọi là Boxing. Trường hợp ngược lại, Unboxing sẽ chuyển từ đối tượng ra một giá trị. Xử lý này đã cho phép chúng ta gọi phương thức ToString( ) trên kiểu dữ liệu int trong ví dụ 5.4.

Boxing được thực hiện ngầm định

Boxing là một sự chuyển đổi ngầm định của một kiểu dữ liệu giá trị sang kiểu dữ liệu tham chiếu là đối tượng. Boxing một giá trị bằng cách tạo ra một thể hiển của đối tượng cần dùng và sao chép giá trị trên vào đối tượng mới tạo. Ta có hình vẽ sau minh họa quá trình Boxing một số nguyên.



Hình 5.5: Boxing số nguyên.

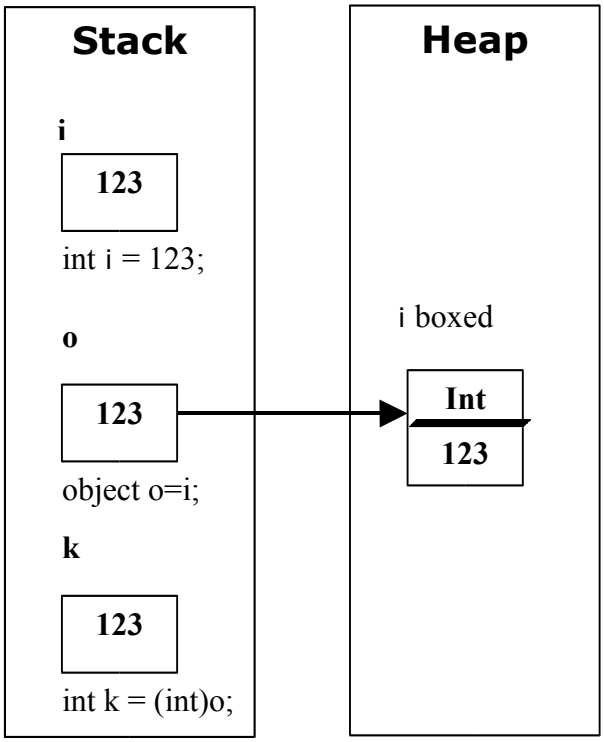
Boxing được thực hiện ngầm định khi chúng ta đặt một kiểu giá trị vào một tham chiếu đang chờ đợi và giá trị sẽ được đưa vào đối tượng một cách tự động ngầm định. Ví dụ, nếu chúng ta gán một kiểu dư liệu cơ bản như kiểu nguyên int vào một biến kiểu Object (điều này hoàn toàn hợp lệ vì kiểu int được dẫn xuất từ lớp Object) thì giá trị này sẽ được đưa vào biến Object, như minh họa sau:

```
using System;
class Boxing
{
    public static void Main()
    {
        int i = 123;
        Console.WriteLine("The object value = {0}", i);
    }
}
```

Unboxing phải được thực hiện tường minh

Việc đưa một giá trị vào một đối tượng được thực hiện một cách ngầm định. Và sự thực hiện ngược lại, unboxing, tức là đưa từ một đối tượng ra một giá trị phải được thực hiện một cách tường minh. Chúng ta phải thiết lập theo hai bước sau:

- Phải chắc chắn rằng đối tượng đã boxing đúng kiểu giá trị đưa ra.
- Sao chép giá trị từ thể hiện hay đối tượng vào biến kiểu giá trị.



Hình 5.6: Unboxing sau khi thực hiện Boxing.

Để thực hiện unboxing thành công, thì đối tượng được unboxing phải được tham chiếu đến một đối tượng, và đối tượng này đã được tạo ra bằng việc boxing một giá trị cùng với kiểu của giá trị được đưa ra. Boxing và Unboxing được minh họa trong ví dụ 5.5.

Ví dụ 5.5: Boxing và Unboxing.

```
---------------------------------------------------------------------------
using System;
public class UnboxingTest
{
    public static void Main()
    {
        int i = 123;
        // Boxing
        object o = i;
        // Unboxing phải được tường minh
        int k = (int) o;
        Console.WriteLine("k: {0}", k);
```

```
    }
}
```

Ví dụ 5.5 tạo một số nguyên i và thực hiện boxing ngầm định khi i được gán cho một đối tượng o. Sau đó giá trị được unboxing một cách tường minh và gán đến một biến nguyên int mới, và cuối cùng giá trị được hiển thị.

Thông thường, chúng ta sẽ bao bọc các hoạt động unboxing trong khối **try**, sẽ được trình bày trong Chương 13. Nếu một đối tượng được Unboxing là null hay là tham chiếu đến một đối tượng có kiểu dữ liệu khác, một InvalidCastException sẽ được phát sinh.

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.

✍ *Ghi chú*: 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 5.6 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.

🗗 *Ví dụ 5.6: 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.

Lưu ý là 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.

Câu hỏi và trả lời

Câu hỏi 1: Có cần thiết phải chỉ định từ khóa **override** trong phương thức phủ quyết của lớp dẫn xuất hay không?

Trả lời 1: Có, chúng ta phải khai báo rõ ràng từ khóa override với phương thức phủ quyết phương thức ảo (của lớp cơ sở) bên trong lớp dẫn xuất.

Câu hỏi 2: Lớp trừu tượng là thế nào? Có thể tạo đối tượng cho lớp trừu tượng hay không?

Trả lời 2: Lớp trừu tượng không có sự thực thi, các phương thức của nó được tạo ra chỉ là hình thức, tức là chỉ có khai báo, do vậy phần định nghĩa bắt buộc phải được thực hiện ở các lớp dẫn xuất từ lớp trừu tượng này. Do chỉ là lớp trừu tượng, không có sự thực thi nên chúng ta không thể tạo thể hiện hay tạo đối tượng cho lớp trừu tượng này.

Câu hỏi 3: Có phải khi tạo một lớp thì phải kế thừa từ một lớp nào không?

Trả lời 3: Không nhất thiết như vậy, tuy nhiên trong C#, thì tất cả các lớp được tạo điều phải dẫn xuất từ lớp Object. Cho dù chúng có được khai báo tường minh hay không. Do đó Object là lớp gốc của tất cả các lớp được xây dựng trong C#. Một điều thú vị là các kiểu dữ liệu giá trị như kiểu nguyên, thực, ký tự cũng được dẫn xuất từ Object.

Câu hỏi 4: Lớp lồng bên trong một lớp là như thế nào?

Trả lời 4: Lớp lồng bên trong một lớp hay còn gọi là lớp nội được khai báo với từ khóa internal, chứa bên trong phạm vi của một lớp. Lớp nội có thể truy cập được các thành viên private của lớp mà nó chứa bên trong

Câu hỏi 5: Có thể kế thừa từ một lớp cơ sở được viết trong ngôn ngữ khác ngôn ngữ C#?

Trả lời 5: Được, một trong những đặc tính của .NET là các lớp có thể kế thừa từ các lớp được viết từ ngôn ngữ khác. Do vậy, trong C# ta có thể kế thừa một lớp được viết từ ngôn ngữ khác của .NET. Và những ngôn ngữ khác cũng có thể kế thừa từ các lớp C# mà ta tạo ra.

Câu hỏi thêm

Câu hỏi 1: Sự đặt biệt hóa được sử dụng trong C# thông qua tính gì?

Câu hỏi 2: Khái niệm đa hình là gì? Khi nào thì cần sử dụng tính đa hình?

Câu hỏi 3: Hãy xây dựng cây phân cấp các lớp đối tượng sau: Xe_Toyota, Xe_Dream, Xe_Spacy, Xe_BMW, Xe_Fiat, Xe_DuLich, Xe_May, Xe?

Câu hỏi 4: Từ khóa new được sử dụng làm gì trong các lớp?
Câu hỏi 5: Một phương thức ảo trong lớp cơ sở có nhất thiết phải được phủ quyết trong lớp dẫn xuất hay không?
Câu hỏi 6: Lớp trừu tượng có cần thiết phải xây dựng hay không? Hãy cho một ví dụ về một lớp trừu tượng cho một số lớp.
Câu hỏi 7: Lớp cô lập là gì? Có thể khai báo protected cho các thành viên của nó được không?
Câu hỏi 8: Lớp Object cung cấp những phương thức nào mà các lớp khác thường xuyên kế thừa để sử dụng.
Câu hỏi 9: Thế nào là boxing và unboxing? Hãy cho biết hai ví dụ về quá trình này?

Bài tập

Bài tập 1: Hãy mở rộng ví dụ trong chương xây dựng thêm các đối tượng khác kế thừa lớp Window như: Label, TextBox, Scrollbar, toolbar, menu,...
Bài tập 2: Hãy xây dựng các lớp đối tượng trong câu hỏi 3, thiết lập các quan hệ kế thừa dựa trên cây kế thừa mà bạn xây dựng. Mỗi đối tượng chỉ cần một thuộc tính là myNane để cho biết tên của nó (như Xe_Toyota thì myName là "Toi la Toyota"...). Các đối tượng có phương thức Who() cho biết giá trị myName của nó. Hãy thực thi sự đa hình trên các lớp đó. Cuối cùng tạo một lớp Tester với hàm Main() để tạo một mảng các đối tượng Xe, đưa từng đối tượng cụ thể vào mảng đối tượng Xe, sau đó cho lặp từng đối tượng trong mảng để nó tự giới thiệu tên (bằng cách gọi hàm Who() của từng đối tượng).
Bài tập 3: Xây dựng các lớp đối tượng hình học như: điểm, đoạn thẳng, đường tròn, hình chữ nhật, hình vuông, tam giác, hình bình hành, hình thoi. Mỗi lớp có các thuộc tính riêng để xác định được hình vẽ biểu diễn của nó như đoạn thẳng thì có điểm đầu, điểm cuối.... Mỗi lớp thực thi một phương thức Draw() phủ quyết Draw() của lớp cơ sở gốc của các hình mà nó dẫn xuất. Hãy xây dựng lớp cơ sở của các lớp trên và thực thi đa hình với phương thức Draw(). Sau đó tạo lớp Tester cùng với hàm Main() để thử nghiệm đa hình giống như bài tập 2 ở trên.
Bài tập 4: Chương trình sau đây có lỗi. Hãy sửa lỗi biên dịch và chạy chương trình. Cho biết lệnh nào gây ra lỗi. Và nguyên nhân gây ra lỗi?

```
using System;
abstract public class Animal
{
      public Animal(string name)
      {
        this.name = name;
      }
```

```
        // phương thức trừu tượng minh họa việc
        // đưa tên của đối tượng
        abstract public void Who();
        // biến thành viên protected
        protected string name;
}
// lớp Dog dẫn xuất từ lớp Animal
public class Dog : Animal
{
        // hàm khởi dựng lấy hai tham số
        public Dog(string name, string color) : base(name)
        {
          this.color = color;
        }
        // phủ quyết phương thức trừu tượng Who()
        public override void Who( )
        {
          Console.WriteLine("Gu gu! Toi la {0} co mau long {1}", name, color);
        }
        // biến private của lớp
        private string color;
}
public class Cat : Animal
{
        // hàm khởi dựng lấy hai tham số
        public Cat(string name, int weight) : base(name)
        {
          this.weight = weight;
        }
        // phủ quyết phương thức trừu tượng Who()
        public override void Who( )
        {
          Console.WriteLine("Meo meo! Toi la {0} can nang {1}", name, weight);
        }
        // biến private của lớp
        private int weight;
}
public class Tester
```

```
{
      static void Main()
      {
        Animal[] Arr = new Animal[3];
        Arr[0] = new Dog("Lu Lu", "Vang");
        Arr[1] = new Cat("Mun", 5);
        Arr[2] = new Animal("Noname");
        for( int i=0; i <3 ; i++)
        {
         Arr[i].Who();
        }
      }
}
```

Chương 6

NẠP CHỒNG TOÁN TỬ

- **Sử dụng từ khóa operator**
- **Hỗ trợ ngôn ngữ .NET khác**
- **Sử dụng toán tử**
- **Toán tử so sánh bằng**
- **Toán tử chuyển đổi**
- **Câu hỏi & bài tập**

Hướng thiết kế của ngôn ngữ C# là tất cả các lớp do người dùng định nghĩa (user-defined classes) có tất cả các chức năng của các lớp được xây dựng sẵn. Ví dụ, giả sử chúng ta định nghĩa một lớp để thể hiện một phân số. Để đảm bảo rằng lớp này có tất cả các chức năng tương tự như các lớp được xây dựng sẵn, nghĩa là chúng ta cho phép thực hiện các phép toán số học trên các thể hiện của phân số chúng ta (như các phép toán cộng phân số, nhân hai phân số,...) và chuyển đổi qua lại giữa phân số và kiểu dữ liệu xây dựng sẵn như kiểu nguyên (int). Dĩ nhiên là chúng ta có thể dễ dàng thực hiện các toán tử bằng cách gọi một phương thức, tương tự như câu lệnh sau:

```
Fraction theSum = firstFraction.Add( secondFraction );
```

Mặc dù cách thực hiện này không sai nhưng về trực quan thì rất tệ không được tự nhiên như kiểu dữ liệu được xây dựng sẵn. Cách thực hiện sau sẽ tốt hơn rất nhiều nếu ta thiết kế được:

```
Fraction  theSum = firstFraction + secondFraction;
```

Cách thực hiện này xem trực quan hơn và giống với cách thực hiện của các lớp được xây dựng sẵn, giống như khi thực hiện phép cộng giữa hai số nguyên int.

Trong chương này chúng ta sẽ tìm hiểu kỹ thuật thêm các toán tử chuẩn vào kiểu dữ liệu do người dùng định nghĩa. Và chúng ta sẽ tìm hiểu các toán tử chuyển đổi để chuyển đổi kiểu dữ liệu do người dùng định nghĩa một cách tường minh hay ngầm định sang các kiểu dữ liệu khác.

Sử dụng từ khóa operator

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.

✍*Ghi chú*: 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.

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.

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.

Ví dụ, 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. Ví dụ, như 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 6.1 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 hay bằng
<=	Compare	Toán tử so sánh nhỏ hơn hay bằng
*=	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
/=	Divide	Toán tử chia rồi gán
--	Decrement	Toán tử giảm
++	Increment	Toán tử tăng
-	Negate	Toán tử phủ định một ngôi
+	Plus	Toán tử cộng một ngôi
~	OnesComplement	Toán tử bù

Bảng 6.1: Tóm tắt một số toán tử trong C#.

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.

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 6.1 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.

Ví dụ 6.1: 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 khộng 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 3/4 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.

Câu hỏi và trả lời

Câu hỏi 1: Có phải khi xây dựng các lớp chúng ta chỉ cần dùng nạp chồng toán tử với các chức năng tính toán ?

Trả lời 1: Đúng là như vậy, việc thực hiện nạp chồng toán tử rất tự nhiên và trực quan. Tuy nhiên một số ngôn ngữ .NET như VB.NET không hỗ trợ việc nạp chồng toán tử nên, tốt nhất nếu muốn cho lớp trong C# của chúng ta có thể được gọi từ ngôn ngữ khác không hỗ trợ nạp chồng toán tử thì nên xây dựng các phương thức tương đương để thực hiện cùng chức năng như: Add, Sub, Mul,..

Câu hỏi 2: Những điều lưu ý nào khi sử dụng nạp chồng toán tử trong một lớp?

Trả lời 2: Nói chung là khi nào thật cần thiết và ít gây ra sự nhầm lẫn. Ví dụ như ta xây dựng lớp Employee có nhiều thuộc tính số như lương, thâm niên, tuổi... Chúng ta muốn xây dựng toán tử ++ cho lương nhưng có thể làm nhầm lẫn với việc tăng số năm công tác, hay tăng tuổi. Do vậy việc sử dụng nạp chồng toán tử cũng phải cân nhắc tránh gây nhầm lẫn. Tốt nhất là sử dụng trong lớp có ít thuộc tính số...

Câu hỏi 3: Khi xây dựng toán tử so sánh thì có phải chỉ cần dùng toán tử so sánh bằng?

Trả lời 3: Đúng là nếu cần dùng toán tử so sánh nào thì chúng ta có thể chỉ tạo ra duy nhất toán tử so sánh đó mà thôi. Tuy nhiên, tốt hơn là chúng ta cũng nên xây dựng thêm toán tử so sánh khác như: so sánh khác, so sánh nhỏ hơn, so sánh lớn hơn...Việc này sẽ làm cho lớp của chúng ta hoàn thiện hơn.

Câu hỏi thêm

Câu hỏi 1: Khi nào sử dụng toán tử chuyển đổi? Thế nào là chuyển đổi tường minh và chuyển đổi ngầm định?

Câu hỏi 2: Có thể tạo ra ký hiện toán tử riêng của ta và thực thi nạp chồng toán tử đó hay không?

Câu hỏi 3: Có bao nhiêu toán tử mà .NET quy định? Ký hiệu của từng toán tử?

Bài tập

Bài tập 1: Hãy tiếp tục phát triển lớp Fraction trong ví dụ của chương bằng cách thêm các toán tử khác như trừ, nhân, chia, so sánh...

Bài tập 2: Xây dựng lớp điểm trong không gian hai chiều, với các toán tử cộng, trừ, nhân, chia.

Bài tập 3: Tương tự như bài tập 2 nhưng điểm nằm trong không gian 3 chiều.

Bài tập 4: Xây dựng lớp số phúc (số ảo) với các phép toán cộng, trừ, nhân, chia.

Chương 7

CẤU TRÚC

- **Định nghĩa một cấu trúc**
- **Tạo cấu trúc**
  - **Cấu trúc là một kiểu giá trị**
  - **Gọi bộ khởi dựng mặc định**
  - **Tạo cấu trúc không gọi new**
- **Câu hỏi & bài tập**

Cấu trúc là kiểu dữ liệu đơn giản do người dùng định nghĩa, kích thước nhỏ dùng để thay thế cho lớp. Những cấu trúc thì tương tự như lớp cũng chứa các phương thức, những thuộc tính, các trường, các toán tử, các kiểu dữ liệu lồng bên trong và bộ chỉ mục (indexer).

Có một số sự khác nhau quan trọng giữa những lớp và cấu trúc. Ví dụ, cấu trúc thì không hỗ trợ kế thừa và bộ hủy giống như kiểu lớp. Một điều quan trọng nhất là trong khi lớp là kiểu dữ liệu tham chiếu, thì cấu trúc là kiểu dữ liệu giá trị (Chương 3 đã thảo luận về kiểu dữ liệu tham chiếu và kiểu dữ liệu giá trị). Do đó cấu trúc thường dùng để thể hiển các đối tượng không đòi hỏi một ngữ nghĩa tham chiếu, hay một lớp nhỏ mà khi đặt vào trong stack thì có lợi hơn là đặt trong bộ nhớ heap.

Một sự nhận xét được rút ra là chúng ta chỉ nên sử dụng những cấu trúc chỉ với những kiểu dữ liệu nhỏ, và những hành vi hay thuộc tính của nó giống như các kiểu dữ liệu được xây dựng sẵn.

Cấu trúc có hiệu quả khi chúng ta sử dụng chúng trong mảng bộ nhớ (Chương 9). Tuy nhiên, cấu trúc sẽ kém hiệu quả khi chúng ta sử dụng dạng tập hợp (collections). Tập hợp được xây dựng hướng tới các kiểu dữ liệu tham chiếu.

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

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 7.1 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.

Ví dụ 7.1 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.

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 7.1, 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.

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.

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.

✍*Ghi chú*: 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.

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 7.1 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 7.2 sau minh họa việc tạo một cấu trúc không sử dụng toán tử **new**.

✍*Ghi chú*: 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.

🗗 *Ví dụ 7.2: 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 7.2 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.

Câu hỏi và trả lời

Câu hỏi 1: Có sự khác nhau giữa cấu trúc và lớp?

Trả lời 1: Đúng có một số sự khác nhau giữa cấu trúc và lớp. Như đã đề cập trong lý thuyết thì lớp là kiểu dữ liệu tham chiếu còn cấu trúc là kiểu dữ liệu giá trị. Điều này được xem là sự khác nhau căn bản giữa cấu trúc và lớp. Ngoài ra cấu trúc cũng không cho phép có hàm hủy và tạo bộ khởi dựng không tham số tường minh. Cấu trúc cũng khác lớp là cấu trúc là

kiểu cô lập tường minh, tức là không cho phép kế thừa từ nó. Và nó cũng không kế thừa được từ bất cứ lớp nào khác. Mặc nhiên, các cấu trúc vẫn kế thừa từ Object như bất cứ kiểu dữ liệu giá trị nào khác trong C#/.

Câu hỏi 2: Trong hai dạng mảng và tập hợp thì lại nào chứa cấu trúc tốt hơn?

Trả lời 2: Cấu trúc có hiệu quả khi sử dụng trong mảng hơn là lưu chúng dưới dạng tập hợp. Dạng tập hợp tốt với kiểu dữ liệu tham chiếu.

Câu hỏi 3: Cấu trúc được lưu trữ ở đâu?

Trả lời 3: Cấu trúc như đã đề cập là kiểu dữ liệu giá trị nên nó được lưu trữ trên stack của chương trình. Ngược với kiểu tham chiếu được đặt trên heap.

Câu hỏi 4: Khi truyền cấu trúc cho một phương thức thì dưới hình thức nào?

Trả lời 4: Do là kiểu giá trị nên khi truyền một đối tượng cấu trúc cho một phương thức thì nó được truyền dưới dạng tham trị chứ không phải tham chiếu.

Câu hỏi 5: Vậy làm thế nào truyền cấu trúc dưới dạng tham chiếu cho một phương thức?

Trả lời 5: Cũng giống như truyền tham chiếu một kiểu giá trị như int, long, char. Ta khai báo khóa ref cho các tham số kiểu cấu trúc. Và khi gọi phương thức thì thêm từ khóa ref vào trước đối mục cấu trúc được truyền vào.

Câu hỏi thêm

Câu hỏi 1: Chúng ta có thể khởi tạo giá trị ban đầu cho các biến thành viên của nó như bên dưới được không? Nếu không được tại sao?

```
struct myStruct
{
    private int mNum = 100;
    ....
}
```

Câu hỏi 2: Sự khác nhau giữa kiểu dữ liệu tham chiếu và kiểu dữ liệu giá trị?

Câu hỏi 3: Sự khác nhau giữa bộ khởi dựng của cấu trúc và bộ khởi dựng của lớp?

Câu hỏi 4: Có nhất thiết phải dùng từ khóa new để tạo đối tượng kiểu cấu trúc hay không? Nếu không thì còn cách nào khác nữa?

Câu hỏi 5: Quá trình boxing và unboxing có diễn ra với một đối tượng là kiểu cấu trúc hay không?

Bài tập

Bài tập 1: Chương trình sau đây có lỗi. Hãy sửa lỗi, biên dịch, và chạy chương trình. Đoạn lệnh nào gây ra lỗi?

```
---------------------------------------------------------------------------
using System;
struct TheStruct
{
```

```
    public int x;
    public TheStruct()
    {
        x = 10;
    }
}
class TestClass
{
    public static void structtaker( TheStruct s)
    {
        s.x = 5;
    }
    public static void Main()
    {
        TheStruct a = new TheStruct();
        a.x = 1;
        structtaker( a);
        Console.WriteLine("a.x = {0}", a.x);
    }
}
```

Bài tập 2: Hãy tính kết quả bằng tay mà chương trình sau xuất ra. Sau đó biên dịch và chạy chương trình để đối sánh kết quả.

```
using System;
class TheClass
{
    public int x;
}

struct TheStruct
{
    public int x;
}
class TestClass
{
    public static void structtaker( TheStruct s)
    {
```

```
        s.x = 5;
    }
    public static void classtaker(TheClass c)
    {
        c.x = 5;
    }
    public static void Main()
    {
        TheStruct a = new TheStruct();
        TheClass b = new TheClass();
        a.x = 1;
        b.x = 1;
        structtaker( a);
        classtaker(b);
        Console.WriteLine("a.x = {0}", a.x);
        Console.WriteLine("b.x = {0}", b.x);
    }
}
```

Bài tập 3: Hãy sửa chương trình trong bài tập 2 để kết quả giá trị a.x của đối tượng a được thay đổi khi ra khỏi hàm structtaker(). Dùng truyền tham chiếu cho cấu trúc.

Chương 8

THỰC THI GIAO DIỆN

- **Thực thi giao diện**
  - **Thực thi nhiều giao diện**
  - **Mở rộng giao diện**
  - **Kết hợp các giao diện**
- **Truy cập phương thức giao diện**
  - **Gán đối tượng cho một giao diện**
  - **Toán tử is**
  - **Toán tử as**
  - **Giao diện đối lập với trừu tượng**
- **Thực thi phủ quyết giao diện**
- **Thực thi giao diện tường minh**
  - **Lựa chọn thể hiện phương thức giao diện**
  - **Ẩ n thành viên**
- **Câu hỏi & bài tập**

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

Một giao diện đưa ra một sự thay thế cho các lớp trừu tượng để tạo ra các sự ràng buộc giữa những lớp và các thành phần client của nó. Những ràng buộc này được khai báo bằng cách sử dụng từ khóa **interface**, từ khóa này khai báo một kiểu dữ liệu tham chiếu để đóng gói các ràng buộc.

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

Khi một lớp thực thi một giao diện, lớp này phải thực thi tất cả các phương thức của giao diện. Đây là một bắt buộc mà các lớp phải thực hiện.

Trong chương này chúng ta sẽ thảo luận cách tạo, thực thi và sử dụng các giao diện. Ngoài ra chúng ta cũng sẽ bàn tới cách thực thi nhiều giao diện cùng với cách kết hợp và mở rộng giao diện. Và cuối cùng là các minh họa dùng để kiểm tra khi một lớp thực thi một giao diện.

Thực thi một giao diện

Cú pháp để định nghĩa một giao diện như sau:

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

Phần thuộc tính chúng ta sẽ đề cập sau. Thành phần bổ sung truy cập bao gồm: **public**, **private**, **protected**, **internal**, và **protected internal** đã được nói đến trong Chương 4, ý nghĩa tương tự như các bổ sung truy cập của lớp.

Theo sau từ khóa **interface** là tên của giao diện. Thông thường tên của giao diện được bắt đầu với từ I hoa (điều này không bắt buộc nhưng việc đặt tên như vậy rất rõ ràng và dễ hiểu, tránh nhầm lẫn với các thành phần khác). Ví dụ một số giao diện có tên như sau: IStorable, ICloneable,...

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

Giả sử chúng ta muốn tạo một giao diện nhằm mô tả những phương thức và thuộc tính của một lớp cần thiết để lưu trữ và truy cập từ một cơ sở dữ liệu hay các thành phần lưu trữ dữ liệu khác như là một tập tin. Chúng ta quyết định gọi giao diện này là IStorage.

Trong giao diện này chúng ta xác nhận hai phương thức: Read() và Write(), khai báo này sẽ được xuất hiện trong phần thân của giao diện như sau:

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

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

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

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

Bây giờ trách nhiệm của chúng ta, với vai trò là người xây dựng lớp Document phải cung cấp một thực thi có ý nghĩa thực sự cho những phương thức của giao diện IStorable. Chúng ta phải thực thi tất cả các phương thức của giao diện, nếu không trình biên dịch sẽ báo một lỗi. Sau đây là đoạn chương trình minh họa việc xây dựng lớp Document thực thi giao diện IStorable.

Ví dụ 8.1: Sử dụng một giao diện.

```
--------------------------------------------------------------------------
using System;
// khai báo giao diện
interface IStorable
{
    // giao diện không khai báo bổ sung truy cập
    // phương thức là public và không thực thi
    void Read();
    void Write(object obj);
    int Status
    {
        get;
        set;
    }
}
// tạo một lớp thực thi giao diện IStorable
public class Document : IStorable
{
    public Document( string s)
    {
        Console.WriteLine("Creating document with: {0}", s);
```

```
        }
        // thực thi phương thức Read()
        public void Read()
        {
            Console.WriteLine("Implement the Read Method for IStorable");
        }
        // thực thi phương thức Write
        public void Write( object o)
        {
            Console.WriteLine("Impleting the Write Method for IStorable");
        }
        // thực thi thuộc tính
        public int Status
        {
            get
            {
                return status;
            }
            set
            {
                status = value;
            }
        }
        // lưu trữ giá trị thuộc tính
        private int status = 0;
}
public class Tester
{
        static void Main()
        {
            // truy cập phương thức trong đối tượng Document
            Document doc = new Document("Test Document");
            doc.Status = -1;
            doc.Read();
            Console.WriteLine("Document Status: {0}", doc.Status);
            // gán cho một giao diện và sử dụng giao diện
            IStorable isDoc = (IStorable) doc;
            isDoc.Status = 0;
```

```
            isDoc.Read();
            Console.WriteLine("IStorable Status: {0}", isDoc.Status);
        }
}
```

Kết quả:

```
Creating document with: Test Document
Implementing the Read Method for IStorable
Document Status: -1
Implementing the Read Method for IStorable
IStorable Status: 0
```

Ví dụ 8.1 định nghĩa một giao diện IStorable với hai phương thức Read(), Write() và một thuộc tính tên là Status có kiểu là số nguyên.. Lưu ý rằng trong phần khai báo thuộc tính không có phần thực thi cho get() và set() mà chỉ đơn giản là khai báo có hành vi là get() và set():

```
int Status { get; set;}
```

Ngoài ra phần định nghĩa các phương thức của giao diện không có phần bổ sung truy cập (ví dụ như: **public**, **protected**, **internal**, **private**). Việc cung cấp các bổ sung truy cập sẽ tạo ra một lỗi. Những phương thức của giao diện được ngầm định là **public** vì giao diện là những ràng buộc được sử dụng bởi những lớp khác. Chúng ta không thể tạo một thể hiện của giao diện, thay vào đó chúng ta sẽ tạo thể hiện của lớp có thực thi giao diện.

Một lớp thực thi giao diện phải đáp ứng đầy đủ và chính xác các ràng buộc đã khai báo trong giao diện. Lớp Document phải cung cấp cả hai phương thức Read() và Write() cùng với thuộc tính Status. Tuy nhiên cách thực hiện những yêu cầu này hoàn toàn phụ thuộc vào lớp Document. Mặc dù IStorage chỉ ra rằng lớp Document phải có một thuộc tính là Status nhưng nó không biết hay cũng không quan tâm đến việc lớp Document lưu trữ trạng thái thật sự của các biến thành viên, hay việc tìm kiếm trong cơ sở dữ liệu. Những chi tiết này phụ thuộc vào phần thực thi của lớp.

Thực thi nhiều giao diện

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

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

Do đó Document cũng phải thực thi những phương thức được xác nhận trong giao diện ICompressible:

```
public void Compress()
{
    Console.WriteLine("Implementing the Compress Method");
}
public void Decompress()
{
    Console.WriteLine("Implementing the Decompress Method");
}
```

Bổ sung thêm phần khai báo giao diện ICompressible và định nghĩa các phương thức của giao diện bên trong lớp Document. Sau khi tạo thể hiện lớp Document và gọi các phương thức từ giao diện ta có kết quả tương tự như sau:

```
Creating document with: Test Document
Implementing the Read Method for IStorable
Implementing Compress
```

Mở rộng giao diện

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

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

Các lớp khác có thể thực thi tự do giao diện ICompressible hay ILoggedCompressible tùy thuộc vào mục đích có cần thêm chức năng hay không. Nếu một lớp thực thi giao diện ILoggedCompressible, thì lớp này phải thực thi tất cả các phương thức của cả hai giao diện ICompressible và giao diện ILoggedCompressible. Những đối tượng của lớp thực thi giao diện ILoggedCompressible có thể được gán cho cả hai giao diện ILoggedCompressible và ICompressible.

Kết hợp các giao diện

Một cách tương tự, chúng ta có thể tạo giao diện mới bằng cách kết hợp các giao diện cũ và ta có thể thêm các phương thức hay các thuộc tính cho giao diện mới. Ví dụ, chúng ta quyết định tạo một giao diện IStorableCompressible. Giao diện mới này sẽ kết hợp những

phương thức của cả hai giao diện và cũng thêm vào một phương thức mới để lưu trữ kích thước nguyên thuỷ của các dữ liệu trước khi nén:

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

Ví dụ 8.2: Minh họa việc mở rộng và kết hợp các giao diện.

```
using System;
interface IStorable
{
        void Read();
        void Write(object obj);
        int Status { get; set;}
}
// giao diện mới
interface ICompressible
{
        void Compress();
        void Decompress();
}
// mở rộng giao diện
interface ILoggedCompressible : ICompressible
{
        void LogSavedBytes();
}
// kết hợp giao diện
interface IStorableCompressible : IStorable, ILoggedCompressible
{
        void LogOriginalSize();
}
interface IEncryptable
{
        void Encrypt();
        void Decrypt();
}
public class Document : IStorableCompressible, IEncryptable
{
```

```
// bộ khởi tạo lớp Document lấy một tham số
public Document( string s)
{
    Console.WriteLine("Creating document with: {0}", s);
}
// thực thi giao diện IStorable
public void Read()
{
    Console.WriteLine("Implementing the Read Method for IStorable");
}
public void Write( object o)
{
    Console.WriteLine("Implementing the Write Method for IStorable");
}
public int Status
{
    get
    {
        return status;
    }
    set
    {
        status = value;
    }
}
// thực thi ICompressible
public void Compress()
{
    Console.WriteLine("Implementing Compress");
}
public void Decompress()
{
    Console.WriteLine("Implementing Decompress");
}
// thực thi giao diện ILoggedCompressible
public void LogSavedBytes()
{
    Console.WriteLine("Implementing LogSavedBytes");
```

```
        }
        // thực thi giao diện IStorableCompressible
        public void LogOriginalSize()
        {
            Console.WriteLine("Implementing LogOriginalSize");
        }
        // thực thi giao diện
        public void Encrypt()
        {
            Console.WriteLine("Implementing Encrypt");
        }
        public void Decrypt()
        {
            Console.WriteLine("Implementing Decrypt");
        }
        // biến thành viên lưu dữ liệu cho thuộc tính
        private int status = 0;
}
public class Tester
{
        public static void Main()
        {
        // tạo đối tượng document
        Document doc = new Document("Test Document");
        // gán đối tượng cho giao diện
        IStorable isDoc = doc as IStorable;
        if ( isDoc != null)
        {
            isDoc.Read();
        }
        else
        {
            Console.WriteLine("IStorable not supported");
        }
        ICompressible icDoc = doc as ICompressible;
        if  ( icDoc != null )
        {
            icDoc.Compress();
```

```
}
else
{
    Console.WriteLine("Compressible not supported");
}
ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
if ( ilcDoc != null )
{
    ilcDoc.LogSavedBytes();
    ilcDoc.Compress();
    // ilcDoc.Read(); // không thể gọi được
}
else
{
    Console.WriteLine("LoggedCompressible not supported");
}
IStorableCompressible isc = doc as IStorableCompressible;
if ( isc != null )
{
    isc.LogOriginalSize();      // IStorableCompressible
    isc.LogSavedBytes();     // ILoggedCompressible
    isc.Compress();   // ICompress
    isc.Read();         // IStorable
}
else
{
    Console.WriteLine("StorableCompressible not supported");
}
IEncryptable ie = doc as IEncryptable;
if ( ie != null )
{
    ie.Encrypt();
}
else
{
    Console.WriteLine("Encryptable not supported");
}
}
```

}

Kết quả:

```
Creating document with: Test Document
Implementing the Read Method for IStorable
Implementing Compress
Implementing LogSavedBytes
Implementing Compress
Implementing LogOriginalSize
Implementing LogSaveBytes
Implementing Compress
Implementing the Read Method for IStorable
Implementing Encrypt
```

Ví dụ 8.2 bắt đầu bằng việc thực thi giao diện IStorable và giao diện ICompressible. Sau đó là phần mở rộng đến giao diện ILoggedCompressible rồi sau đó kết hợp cả hai vào giao diện IStorableCompressible. Và giao diện cuối cùng trong ví dụ là IEncrypt.

Chương trình Tester tạo đối tượng Document mới và sau đó gán lần lượt vào các giao diện khác nhau. Khi một đối tượng được gán cho giao diện ILoggedCompressible, chúng ta có thể dùng giao diện này để gọi các phương thức của giao diện ICompressible bởi vì ILogged-Compressible mở rộng và thừa kế các phương thức từ giao diện cơ sở:

```
ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
if ( ilcDoc != null )
{
    ilcDoc.LogSavedBytes();
    ilcDoc.Compress();
    // ilcDoc.Read(); // không thể gọi được
}
```

Tuy nhiên, ở đây chúng ta không thể gọi phương thức Read() bởi vì phương thức này của giao diện IStorable, không liên quan đến giao diện này. Nếu chúng ta thêm lệnh này vào thì chương trình sẽ biên dịch lỗi.

Nếu chúng ta gán vào giao diện IStorableCompressible, do giao diện này kết hợp hai giao diện IStorable và giao diện ICompressible, chúng ta có thể gọi tất cả những phương thức của IStorableCompressible, ICompressible, và IStorable:

```
IStorableCompressible isc = doc as IStorableCompressible;
if ( isc != null )
{
    isc.LogOriginalSize();      // IStorableCompressible
```

```
        isc.LogSaveBytes();  // ILoggedCompressible
        isc.Compress();   // ICompress
        isc.Read();       // IStorable
    }
```

Truy cập phương thức giao diện

Chúng ta có thể truy cập những thành viên của giao diện IStorable như thể là các thành viên của lớp Document:

```
Document doc = new Document("Test Document");
doc.status = -1;
doc.Read();
```

hay là ta có thể tạo thể hiện của giao diện bằng cách gán đối tượng Document cho một kiểu dữ liệu giao diện, và sau đó sử dụng giao diện này để truy cập các phương thức:

```
IStorable isDoc = (IStorable) doc;
isDoc.status = 0;
isDoc.Read();
```

✍*Ghi chú*: Cũng như đã nói trước đây, chúng ta không thể tạo thể hiện của giao diện một cách trực tiếp.Do đó chúng ta không thể thực hiện như sau:

```
IStorable isDoc = new IStorable();
```

Tuy nhiên chúng ta có thể tạo thể hiện của lớp thực thi như sau:

```
Document doc = new Document("Test Document");
```

Sau đó chúng ta có thể tạo thể hiện của giao diện bằng cách gán đối tượng thực thi đến kiểu dữ liệu giao diện, trong trường hợp này là IStorable:

```
IStorable isDoc = (IStorable) doc;
```

Chúng ta có thể kết hợp những bước trên như sau:

```
IStorable isDoc = (IStorable) new Document("Test Document");
```

Nói chung, cách thiết kế tốt nhất là quyết định truy cập những phương thức của giao diện thông qua tham chiếu của giao diện. Do vậy cách tốt nhất là sử dụng isDoc.Read(), hơn là sử dụng doc.Read() trong ví dụ trước. Truy cập thông qua giao diện cho phép chúng ta đối xử giao diện một cách đa hình. Nói cách khác, chúng ta tạo hai hay nhiều hơn những lớp thực thi giao diện, và sau đó bằng cách truy cập lớp này chỉ thông qua giao diện.

Gán đối tượng cho một giao diện

Trong nhiều trường hợp, chúng ta không biết trước một đối tượng có hỗ trợ một giao diện đưa ra. Ví dụ, giả sử chúng ta có một tập hợp những đối tượng Document, một vài đối tượng đã được lưu trữ và số còn lại thì chưa. Và giả sử chúng ta đã thêm giao diện giao diện thứ hai, ICompressible cho những đối tượng để nén dữ liệu và truyền qua mail nhanh chóng:

```
interface ICompressible
{
```

```
        void Compress();
        void Decompress();
    }
```

Nếu đưa ra một kiểu Document, và ta cũng không biết là lớp này có hỗ trợ giao diện IStorable hay ICompressible hoặc cả hai. Ta có thể có đoạn chương trình sau:

```
Document doc = new Document("Test Document");
IStorable isDoc = (IStorable) doc;
isDoc.Read();
ICompressible icDoc = (ICompressible) doc;
icDoc.Compress();
```

Nếu Document chỉ thực thi giao diện IStorable:

```
public class Document : IStorable
```

phép gán cho ICompressible vẫn được biên dịch bởi vì ICompressible là một giao diện hợp lệ. Tuy nhiên, do phép gán không hợp lệ nên khi chương trình chạy thì sẽ tạo ra một ngoại lệ (exception):

```
A exception of type System.InvalidCastException was thrown.
```

Phần ngoại lệ sẽ được trình bày trong Chương 11.

Toán tử is

Chúng ta muốn kiểm tra một đối tượng xem nó có hỗ trợ giao diện, để sau đó thực hiện các phương thức tương ứng. Trong ngôn ngữ C# có hai cách để thực hiện điều này. Phương pháp đầu tiên là sử dụng toán tử **is**.

Cú pháp của toán tử **is** là:

```
<biểu thức> is <kiểu dữ liệu>
```

Toán tử **is** trả về giá trị true nếu biểu thức thường là kiểu tham chiếu có thể được gán an toàn đến kiểu dữ liệu cần kiểm tra mà không phát sinh ra bất cứ ngoại lệ nào. Ví dụ 8.3 minh họa việc sử dụng toán tử **is** để kiểm tra Document có thực thi giao diện IStorable hay ICompressible.

🗗 *Ví dụ 8.3: Sử dụng toán tử is.*

```
---------------------------------------------------------------------------
using System;
interface IStorable
{
        void Read();
        void Write(object obj);
        int Status { get; set; }
}
// giao diện mới
```

```
interface ICompressible
{
        void Compress();
        void Decompress();
}
// Document thực thi IStorable
public class Document : IStorable
{
        public Document( string s)
        {
            Console.WriteLine("Creating document with: {0}", s);
        }
        // IStorable
        public void Read()
        {
            Console.WriteLine("Implementing the Read Method for IStorable");
        }
        // IStorable.WriteLine()
        public void Write( object o)
        {
            Console.WriteLine("Implementing the Write Method for IStorable");
        }
        // IStorable.Status
        public int Status
        {
            get
            {
                return status;
            }
            set
            {
                status = value;
            }
        }
        // bien thanh vien luu gia tri cua thuoc tinh Status
        private int status = 0;
}
public class Tester
```

```
{
        static void Main()
        {
            Document doc = new Document("Test Document");
            // chỉ gán khi an toàn
            if ( doc is IStorable )
            {
                IStorable isDoc = (IStorable) doc;
                isDoc.Read();
            }
            // việc kiểm tra này sẽ sai
            if ( doc is ICompressible )
            {
                ICompressible icDoc = (ICompressible) doc;
                icDoc.Compress();
            }
        }
}
```

Trong ví dụ 8.3, hàm Main() lúc này sẽ thực hiện việc gán với interface khi được kiểm tra hợp lệ. Việc kiểm tra này được thực hiện bởi câu lệnh **if**:

```
if ( doc is IStorable )
```

Biểu thức điều kiện sẽ trả về giá trị true và phép gán sẽ được thực hiện khi đối tượng có thực thi giao diện bên phải của toán tử **is**.

Tuy nhiên, việc sử dụng toán tử **is** đưa ra một việc không có hiệu quả. Để hiểu được điều này, chúng ta xem đoạn chương trình được biên dịch ra mã IL. Ở đây sẽ có một ngoại lệ nhỏ, các dòng bên dưới là sử dụng hệ thập lục phân:

```
IL_0023: isinst     ICompressible
IL_0028:  brfalse.s  IL_0039
IL_002a:  ldloc.0
IL_002b:  castclass  ICompressible
IL_0030:  stloc.2
IL_0031:  ldloc.2
IL_0032:  callvirt        instance void ICompressible::Compress()
IL_0037:  br.s       IL_0043
IL_0039:  ldstr      "Compressible not supported"
```

Điều quan trọng xảy ra là khi phép kiểm tra ICompressible ở dòng 23. Từ khóa **isinst** là mã MSIL tương ứng với toán tử **is**. Nếu việc kiểm tra đối tượng (doc) đúng kiểu của kiểu bên

phải. Thì chương trình sẽ chuyển đến dòng lệnh 2b để thực hiện tiếp và castclass được gọi. Điều không may là castcall cũng kiểm tra kiểu của đối tượng. Do đó việc kiểm tra sẽ được thực hiện hai lần. Giải pháp hiệu quả hơn là việc sử dụng toán tử **as**.

Toán tử as

Toán tử **as** kết hợp toán tử **is** và phép gán bằng cách đầu tiên kiểm tra hợp lệ phép gán (kiểm tra toán tử **is** trả về true) rồi sau đó phép gán được thực hiện. Nếu phép gán không hợp lệ (khi phép gán trả ề giá trị false), thì toán tử **as** trả về giá trị null.

✍*Ghi chú*: Từ khóa null thể hiện một tham chiếu không tham chiếu đến đâu cả (null reference). Đối tượng có giá trị null tức là không tham chiếu đến bất kỳ đối tượng nào.

Sử dụng toán tử **as** để loại bỏ việc thực hiện các xử lý ngoại lệ. Đồng thời cũng né tránh việc thực hiện kiểm tra dư thừa hai lần. Do vậy, việc sử dụng tối ưu của phép gán cho giao diện là sử dụng **as**.

Cú pháp sử dụng toán tử **as** như sau:

<biểu thức> **as** <kiểu dữ liệu>

Đoạn chương trình sau thay thế việc sử dụng toán tử **is** bằng toán tử **as** và sau đó thực hiện việc kiểm tra xem giao diện được gán có null hay không:

```
static void Main()
{
    Document doc = new Document("Test Document");
    IStorable isDoc = doc as IStorable;
    if ( isDoc != null )
    {
        isDoc.Read();
    }
    else
    {
        Console.WriteLine("IStorable not supported");
    }
    ICompressible icDoc = doc as ICompressible;
    if ( icDoc != null)
    {
        icDoc.Compress();
    }
    else
    {
        Console.WriteLine("Compressible not supported");
    }
```

```
}
```

Ta có thể so sánh đoạn mã IL sau với đoạn mã IL sử dụng toán tử **is** trước sẽ thấy đoạn mã sau có nhiều hiệu quả hơn:

```
IL_0023:  isinst     ICompressible
IL_0028:  stloc.2
IL_0029:  ldloc.2
IL_002a:  brfalse.s  IL_0034
IL_002c:  ldloc.2
IL_002d:  callvirt   instance void ICompressible::Compress()
```

✍*Ghi chú*: Nếu mục đích của chúng ta là kiểm tra một đối tượng có hỗ trợ một giao diện và sau đó là thực hiện việc gán cho một giao diện, thì cách tốt nhất là sử dụng toán tử **as** là hiệu quả nhất. Tuy nhiên, nếu chúng ta chỉ muốn kiểm tra kiểu dữ liệu và không thực hiện phép gán ngay lúc đó. Có lẽ chúng ta chỉ muốn thực hiện việc kiểm tra nhưng không thực hiện việc gán, đơn giản là chúng ta muốn thêm nó vào danh sách nếu chúng thực sự là một giao diện. Trong trường hợp này, sử dụng toán tử **is** là cách lựa chọn tốt nhất.

Giao diện đối lập với lớp trừu tượng

Giao diện rất giống như các lớp trừu tượng. Thật vậy, chúng ta có thể thay thế khai báo của IStorable trở thành một lớp trừu tượng:

```
abstract class Storable
{
    abstract public void Read();
    abstract public void Write();
}
```

Bây giờ lớp Document có thể thừa kế từ lớp trừu tượng IStorable, và cũng không có gì khác nhiều so với việc sử dụng giao diện.

Tuy nhiên, giả sử chúng ta mua một lớp List từ một hãng thứ ba và chúng ta muốn kết hợp với lớp có sẵn như Storable. Trong ngôn ngữ C++ chúng ta có thể tạo ra một lớp StorableList kế thừa từ List và cả Storable. Nhưng trong ngôn ngữ C# chúng ta không thể làm được, chúng ta không thể kế thừa từ lớp trừu tượng Storable và từ lớp List bởi vì trong C# không cho phép thực hiện đa kế thừa từ những lớp.

Tuy nhiên, ngôn ngữ C# cho phép chúng ta thực thi bất cứ những giao diện nào và dẫn xuất từ một lớp cơ sở. Do đó, bằng cách làm cho Storable là một giao diện, chúng ta có thể kế thừa từ lớp List và cũng từ IStorable. Ta có thể tạo lớp StorableList như sau:

```
public class StorableList : List, IStorable
{
    // phương thức List...
    .....
```

```
        public void Read()
        {...}
        public void Write( object o)
        {...}
        //....
    }
```

Thực thi phủ quyết giao diện

Khi thực thi một lớp chúng ta có thể tự do đánh dấu bất kỳ hay tất cả các phương thức thực thi giao diện như là một phương thức ảo. Ví dụ, lớp Document thực thi giao diện IStorable và có thể đánh dấu các phương thức Read() và Write() như là phương thức ảo. Lớp Document có thể đọc và viết nội dung của nó vào một kiểu dữ liệu File. Những người phát triển sau có thể dẫn xuất một kiểu dữ liệu mới từ lớp Document, có thể là lớp Note hay lớp EmailMessage, và những người này mong muốn lớp Note đọc và viết vào cơ sở dữ liệu hơn là vào một tập tin.

Ví dụ 8.4 mở rộng từ ví dụ 8.3 và minh họa việc phủ quyết một thực thi giao diện. Phương thức Read() được đánh dấu như phương thức ảo và thực thi bởi Document.Read() và cuối cùng là được phủ quyết trong kiểu dữ liệu Note được dẫn xuất từ Document.

Ví dụ 8.4: Phủ quyết thực thi giao diện.

```
---------------------------------------------------------------------------
using System;
interface IStorable
{
        void Read();
        void Write();
}
// lớp Document đơn giản thực thi giao diện IStorable
public class Document : IStorable
{
        // bộ khởi dựng
        public Document( string s)
        {
            Console.WriteLine("Creating document with: {0}", s);
        }
        // đánh dấu phương thức Read ảo
        public virtual void Read()
        {
            Console.WriteLine("Document Read Method for IStorable");
```

```
        }
        // không phải phương thức ảo
        public void Write()
        {
            Console.WriteLine("Document Write Method for IStorable");
        }
}
// lớp dẫn xuất từ Document
public class Note : Document
{
        public Note( string s) : base(s)
        {
            Console.WriteLine("Creating note with: {0}", s);
        }
        // phủ quyết phương thức Read()
        public override void Read()
        {
            Console.WriteLine("Overriding the Read Method for Note!");
        }
        // thực thi một phương thức Write riêng của lớp
        public void Write()
        {
            Console.WriteLine("Implementing the Write method for Note!");
        }
}
public class Tester
{
        static void Main()
        {
            // tạo một đối tượng Document
            Document theNote = new Note("Test Note");
            IStorable isNote = theNote as IStorable;
            if ( isNote != null)
            {
                isNote.Read();
                isNote.Write();
            }
            Console.WriteLine("\n");
```

```
            // trực tiếp gọi phương thức
            theNote.Read();
            theNote.Write();
            Console.WriteLine("\n");
            // tạo đối tượng Note
            Note note2 = new Note("Second Test");
            IStorable isNote2 = note2 as IStorable;
            if ( isNote != null )
            {
                isNote2.Read();
                isNote2.Write();
            }
            Console.WriteLine("\n");
            // trực tiếp gọi phương thức
            note2.Read();
            note2.Write();
        }
}
```

Kết quả:

```
Creating document with: Test Note
Creating note with: Test Note
Overriding the Read method for Note!
Document Write Method for IStorable

Overriding the Read method for Note!
Document Write Method for IStorable

Creating document with: Second Test
Creating note with: Second Test
Overriding the Read method for Note!
Document Write Method for IStorable

Overriding the Read method for Note!
Implementing the Write method for Note!
```

Trong ví dụ trên, lớp Document thực thi một giao diện đơn giản là IStorable:

```
interface IStorable
```

```
{
    void Read();
    void Write();
}
```

Người thiết kế của lớp Document thực thi phương thức Read() là phương thức ảo nhưng không tạo phương thức Write() tương tự như vậy:

```
public virtual void Read()
```

Trong ứng dụng thế giới thực, chúng ta cũng đánh dấu cả hai phương thức này là phương thức ảo. Tuy nhiên trong ví dụ này chúng ta minh họa việc người phát triển có thể tùy ý chọn các phương thức ảo của giao diện mà lớp thực thi.

Một lớp mới Note dẫn xuất từ Document:

```
public class Note : Document
```

Việc phủ quyết phương thức Read() trong lớp Note là không cần thiết, nhưng ở đây ta tự do làm điều này:

```
public override void Read()
```

Trong lớp Tester, phương thức Read() và Write() được gọi theo bốn cách sau:

- Thông qua lớp cơ sở tham chiếu đến đối tượng của lớp dẫn xuất
- Thông qua một giao diện tạo từ lớp cơ sở tham chiếu đến đối tượng dẫn xuất
- Thông qua một đối tượng dẫn xuất
- Thông qua giao diện tạo từ đối tượng dẫn xuất

Thực hiện cách gọi thứ nhất, một tham chiếu Document được tạo ra, và địa chỉ của một đối tượng mới là lớp dẫn xuất Note được tạo trên heap và gán trở lại cho đối tượng Document:

```
Document theNote = new Note("Test Note");
```

Môt tham chiếu giao diện được tạo ra và toán tử **as** được sử dụng để gán Document cho tham chiếu giao diện IStorable:

```
IStorable isNote = theNote as IStorable;
```

Sau đó gọi phương thức Read() và Write() thông qua giao diện. Kết xuất của phương thức Read() được thực hiện một cách đa hình nhưng phương thức Write() thì không, do đó ta có kết xuất sau:

```
Overriding the Read method for Note!
Document Write Method for IStorable
```

Phương thức Read() và Write() cũng được gọi trực tiếp từ bản thân đối tượng:

```
theNote.Read();
theNote.Write();
```

và một lần nữa chúng ta thấy việc thực thi đa hình làm việc:

```
Overriding the Read method for Note!
Document Write Method for IStorable
```

Trong trường hợp này, phương thức Read() của lớp Note được gọi, và phương thức Write() của lớp Document được gọi.

Để chứng tỏ rằng kết quả này của phương thức phủ quyết, chúng ta tiếp tục tạo đối tượng Note thứ hai và lúc này ta gán cho một tham chiếu Note. Điều này được sử dụng để minh họa cho những trường hợp cuối cùng (gọi thông qua đối tượng dẫn xuất và gọi thông qua giao diện được tạo từ đối tượng dẫn xuất):

```
Note note2 = new Note("Second Test");
```

Một lần nữa, khi chúng ta gán cho một tham chiếu, phương thức phủ quyết Read() được gọi. Tuy nhiên, khi những phương thức được gọi trực tiếp từ đối tượng Note:

```
note2.Read();
note2.Write();
```

kết quả cho ta thấy rằng cách phương thức của Note được gọi chứ không phải của một phương thức Document:

```
Overriding the Read method for Note!
Implementing the Write method dor Note!
```

Thực thi giao diện tường minh

Trong việc thực thi giao diện cho tới giờ, những lớp thực thi (trong trường hợp này là Document) tạo ra các phương thức thành viên cùng ký hiệu và kiểu trả về như là phương thức được mô tả trong giao diên. Chúng ta không cần thiết khai báo tường minh rằng đây là một thực thi của một giao diện, việc này được hiểu ngầm bởi trình biên dịch.

Tuy nhiên, có vấn đề xảy ra khi một lớp thực thi hai giao diện và cả hai giao diện này có các phương thức cùng một ký hiệu. Ví dụ 8.5 tạo ra hai giao diện: IStorable và ITalk. Sau đó thực thi phương thức Read() trong giao diện ITalk để đọc ra tiếng nội dung của một cuốn sách. Không may là phương thức này sẽ tranh chấp với phương thức Read() của IStorable mà Document phải thực thi.

Bởi vì cả hai phương thức IStorable và ITalk có cùng phương thức Read(),việc thực thi lớp Document phải sử dụng thực thi tường minh cho mỗi phương thức. Với việc thực thi tường minh, lớp thực thi Document sẽ khai báo tường minh cho mỗi phương thức:

```
void ITalk.Read();
```

Điều này sẽ giải quyết việc tranh chấp, nhưng nó sẽ tạo ra hàng loạt các hiệu ứng thú vị.

Đầu tiên, không cần thiết sử dụng thực thi tường minh với những phương thức khác của Talk:

```
public void Talk();
```

vì không có sự tranh chấp cho nên ta khai báo như thông thường.

Điều quan trọng là các phương thức thực thi tường minh không có bổ sung truy cập:

```
void ITalk.Read();
```

Phương thức này được hiểu ngầm là **public**.

Thật vậy, một phương thức được khai báo tường minh thì sẽ không được khai báo với các từ khóa bổ sung truy cập: **abstract**, **virtual**, **override**, và **new**.

Một điều quan trọng khác là chúng ta không thể truy cập phương thức thực thi tường minh thông qua chính đối tượng. Khi chúng ta viết:

```
theDoc.Read();
```

Trình biên dịch chỉ hiểu rằng chúng ta thực thi phương thức giao diện ngầm định cho IStorable. Chỉ một cách duy nhất truy cập các phương thức thực thi tường minh là thông qua việc gán cho giao diện để thực thi:

```
ITalk itDoc = theDoc as ITalk;
if ( itDoc != null )
{
    itDoc.Read();
}
```

Sử dụng thực thi tường minh được áp dụng trong ví dụ 8.5

Ví dụ 8.5: Thực thi tường minh.

```
---------------------------------------------------------------------------
using System;
interface IStorable
{
        void  Read();
        void Write();
}
interface ITalk
{
        void Talk();
        void Read();
}
// lớp Document thực thi hai giao diện
public class Document : IStorable, ITalk
{
        // bộ khởi dựng
        public Document( string s)
        {
            Console.WriteLine("Creating document with: {0}",s);
        }
        // tạo phương thức ảo
        public virtual void Read()
        {
```

```
            Console.WriteLine("Implementing IStorable.Read");
        }
        // thực thi bình thường
        public void Write()
        {
            Console.WriteLine("Implementing IStorable.Write");
        }
        // thực thi tường minh
        void ITalk.Read()
        {
            Console.WriteLine("Implementing ITalk.Read");
        }
        public void Talk()
        {
            Console.WriteLine("Implementing ITalk.Talk");
        }
}
public class Tester
{
        static void Main()
        {
            // tạo đối tượng Document
            Document theDoc = new Document("Test Document");
            IStorable isDoc = theDoc as IStorable;
            if ( isDoc != null )
            {
                isDoc.Read();
            }
            ITalk itDoc = theDoc as ITalk;
            if ( itDoc != null )
            {
                itDoc.Read();
            }
            theDoc.Read();
            theDoc.Talk();
        }
}
```

Kết quả:

```
Creating document with: Test Document
Implementing IStorable.Read
Implementing ITalk.Read
Implementing IStorable.Read
Implementing ITalk.Talk
---------------------------------------------------------------------------
```

Lựa chọn việc thể hiện phương thức giao diện

Những người thiết kế lớp có thể thu được lợi khi một giao diện được thực thi thông qua thực thi tường minh và không cho phép các thành phần client của lớp truy cập trừ phi sử dụng thông qua việc gán cho giao diện.

Giả sử nghĩa của đối tượng Document chỉ ra rằng nó thực thi giao diện IStorable, nhưng không muốn phương thức Read() và Write() là phần giao diện **public** của lớp Document. Chúng ta có thể sử dụng thực thi tường minh để chắc chắn chỉ có thể truy cập thông qua việc gán cho giao diện. Điều này cho phép chúng ta lưu trữ ngữ nghĩa của lớp Document trong khi vẫn có thể thực thi được giao diện IStorable. Nếu thành phần client muốn đối tượng thực thi giao diện IStorable, nó có thể thực hiện gán tường minh cho giao diện để gọi các phương thức thực thi giao diện. Nhưng khi sử dụng đối tượng Document thì nghĩa là không có phương thức Read() và Write().

Thật vậy, chúng ta có thể lựa chọn thể hiện những phương thức thông qua thực thi tường minh, do đó chúng ta có thể trưng bày một vài phương thức thực thi như là một phần của lớp Document và một số phương thức khác thì không. Trong ví dụ 8.5, đối tượng Document trưng bày phương thức Talk() như là phương thức của lớp Document, nhưng phương thức Talk.Read() chỉ được thể hiện thông qua gán cho giao diện. Thậm chí nếu IStorable không có phương thức Read(), chúng ta cũng có thể chọn thực thi tường minh phương thức Read() để phương thức không được thể hiện ra bên ngoài như các phương thức của Document.

Chúng ta lưu ý rằng vì thực thi giao diện tường minh ngăn ngừa việc sử dụng từ khóa **virtual**, một lớp dẫn xuất có thể được hỗ trợ để thực thi lại phương thức. Do đó, nếu Note dẫn xuất từ Document, nó có thể được thực thi lại phương thức Talk.Read() bởi vì lớp Document thực thi phương thức Talk.Read() không phải ảo.

Ẩ n thành viên

Ngôn ngữ C# cho phép ẩn các thành viên của giao diện. Ví dụ, chúng ta có một giao diện IBase với một thuộc tính P:

```
interface IBase
```

```
{
    int P { get; set;}
}
```

và sau đó chúng ta dẫn xuất từ giao diện này ra một giao diện khác, IDerived, giao diện mới này làm ẩn thuộc tính P với một phương thức mới P():

```
interface IDerived : IBase
{
    new int P();
}
```

Việc cài đặt này là một ý tưởng tốt, bây giờ chúng ta có thể ẩn thuộc tính P trong lớp cơ sở. Một thực thi của giao diện dẫn xuất này đòi hỏi tối thiểu một thành viên giao diện tường minh. Chúng ta có thể sử dụng thực thi tường minh cho thuộc tính của lớp cơ sở hoặc của phương thức dẫn xuất, hoặc chúng ta có thể sử dụng thực thi tường minh cho cả hai. Do đó, ba phiên bản được viết sau đều hợp lệ:

```
class myClass : IDerived
{
    // thực thi tường minh cho thuộc tính cơ sở
    int IBase.p { get{...}}
    // thực thi ngầm định phương thức dẫn xuất
    public int P() {...}
}

class myClass : IDerived
{
    // thực thi ngầm định cho thuộc tính cơ sở
    public int P { get{...}}
    // thực thi tường minh phương thức dẫn xuất
    int IDerived.P() {...}
}
class myClass : IDerived
{
    // thực thi tường minh cho thuộc tính cơ sở
    int IBase.P { get{...}}
    // thực thi tường minh phương thức dẫn xuất
    int IDerived.P(){...}
}
```

Truy cập lớp không cho dẫn xuất và kiểu giá trị

Nói chung, việc truy cập những phương thức của một giao diện thông qua việc gán cho giao diện thì thường được thích hơn. Ngoại trừ đối với kiểu giá trị (như cấu trúc) hoặc với các lớp không cho dẫn xuất (sealed class). Trong trường hợp này, cách ưu chuộng hơn là gọi phương thức giao diện thông qua đối tượng.

Khi chúng ta thực thi một giao diện trong một cấu trúc, là chúng ta đang thực thi nó trong một kiểu dữ liệu giá trị. Khi chúng ta gán cho môt tham chiếu giao diện, có một boxing ngầm định của đối tượng. Chẳng may khi chúng ta sử dụng giao diện để bổ sung đối tượng, nó là một đối tượng đã boxing, không phải là đối tượng nguyên thủy cần được bổ sung. Xa hơn nữa, nếu chúng ta thay đổi kiểu dữ liệu giá trị, thì kiểu dữ liệu được boxing vẫn không thay đổi. Ví dụ 8.6 tạo ra một cấu trúc và thực thi một giao diện IStorable và minh họa việc boxing ngầm định khi gán một cấu trúc cho một tham chiếu giao diện.

Ví dụ 8.6: Tham chiếu đến kiểu dữ liệu giá trị.

```
---------------------------------------------------------------------------
using System;
// khai báo một giao diện đơn
interface IStorable
{
        void Read();
        int Status { get; set;}
}
// thực thi thông qua cấu trúc
public struct myStruct : IStorable
{
        public void Read()
        {
            Console.WriteLine("Implementing IStorable.Read");
        }
        public int Status
        {
            get
            {
                return status;
            }
            set
            {
                status = value;
            }
        }
```

```
        // biến thành viên lưu giá trị thuộc tính Status
        private int status;
}
public class Tester
{
        static void Main()
        {
            // tạo một đối tượng myStruct
            myStruct theStruct = new myStruct();
            theStruct.Status = -1; // khởi tạo
            Console.WriteLine("theStruct.Status: {0}", theStruct.Status);
            // thay đổi giá trị
            theStruct.Status = 2;
            Console.WriteLine("Changed object");
            Console.WriteLine("theStruct.Status: {0}", theStruct.Status);
            // gán cho giao diện
            // boxing ngầm định
            IStorable isTemp = (IStorable) theStruct;
            // thiết lập giá trị thông qua tham chiếu giao diện
            isTemp.Status = 4;
            Console.WriteLine("Changed interface");
            Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
                theStruct.Status, isTemp.Status);
            // thay đổi giá trị một lần nữa
            theStruct.Status = 6;
            Console.WriteLine("Changed object.");
            Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
                theStruct.Status, isTemp.Status);
        }
}
```

Kết quả:

```
theStruct.Status: -1
Changed object.
theStruct.Status: 2
Changed interface
theStruct.Status: 2, isTemp: 4
Changed object
```

```
theStruct.Status: 6, isTemp: 4
---------------------------------------------------------------------------
```

Trong ví dụ 8.6, giao diện IStorable có một phương thức Read() và môt thuộc tính là Status. Giao diện này được thực thi bởi một cấu trúc tên là myStruct:

```
public struct myStruct : IStorable
```

Đoạn mã nguồn thú vị bên trong Tester. Chúng ta bắt đầu bằng việc tạo một thể hiện của cấu trúc và khởi tạo thuộc tính là –1, sau đó giá trị của status được in ra:0

```
myStruct theStruct = new myStruct();
theStruct.Status = -1; // khởi tạo
Console.WriteLine("theStruct.Status: {0}", theStruct.status);
```

Kết quả là giá trị của status được thiết lập:

```
theStruct.Status = -1;
```

Kế tiếp chúng ta truy cập thuộc tính để thay đổi status, một lần nữa thông qua đối tượng giá trị:

```
// thay đổi giá trị
theStruct.Status = 2;
Console.WriteLine("Changed object");
Console.WriteLine("theStruct.Status: {0}", theStruct.Status);
```

kết quả chỉ ra sự thay đổi:

```
Changed object
theStruct.Status: 2
```

Tại điểm này, chúng ta tạo ra một tham chiếu đến giao diện IStorable, một đối tượng giá trị theStruct được boxing ngầm và gán lại cho tham chiếu giao diện. Sau đó chúng ta dùng giao diện để thay đổi giá trị của status bằng 4:

```
// gán cho một giao diện
// boxing ngầm định
IStorable isTemp = (IStorable) theStruct;
// thiết lập giá trị thông qua tham chiếu giao diện
isTemp.Status = 4;
Console.WriteLine("Changed interface");
Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
        theStruct.Status, isTemp.Status);
```

như chúng ta đã thấy kết quả thực hiện có một điểm khác biệt:

```
Changed interface
theStruct.Status: 2, isTemp: 4
```

Điều xảy ra là: đối tượng được giao diện tham chiếu đến thay đổi giá trị status bằng 4, nhưng đối tượng giá trị cấu trúc không thay đổi.Thậm chí có nhiều thú vị hơn khi chúng ta truy cập phương thức thông qua bản thân đối tượng:

```
    // than đổi giá trị lần nữa
    theStruct.Status = 6;
    Console.WriteLine("Changed object");
    Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
            theStruct.Status, isTemp.Status);
```

kết quả đối tượng giá trị thay đổi nhưng đối tượng được boxing và được giao diện tham chiếu không thay đổi:

```
    Changed object
    theStruct.Status: 6,  isTemp: 4
```

Ta thử xem đoạn mã IL để hiểu tham về cách thực hiện trên:

Ví dụ 8.7: MSIL phát sinh từ ví dụ 8.6.

```
---------------------------------------------------------------------------
method private hidebysig static void Main() il managed
{
    .entrypoint
    // Code size   206 (0xce)
    .maxstack     4
    .local       ([0] value class myStruct theStruct,
                  [1] class IStorable isTemp,
                  [2] int32 V_2)
    IL_0000:      ldloca.s    theStruct
    IL_0002:      iniobj      myStruct
    IL_0008:      ldloca.s    theStruct
    IL_000a:      ldc.i4.ml
    IL_000b:      call        instance void myStruct::set_status(int32)
    IL_0010:      ldstr       "theStruct.Status: {0}"
    IL_0015:      ldloca.s    theStruct
    IL_0017:      call        instance int32 myStruct::get_status()
    IL_001c:      stloc.2
    IL_001d:      ldloca.s    V_2
    IL_001f:      box         [mscorlib]System.Int32
    IL_0024:      call        void [mscorlib] System.Console::WriteLine
        (class System.String, class System.Object)
    IL_0029:      ldloca.s    theStruct
    IL_002b:      ldc.i4.2
    IL_002c:      call        instance void myStruct::set_status(int32)
```

```
IL_0031:    ldstr     "Changed object"
IL_0036:    call      void [mscorlib]System.Console::WriteLine
    (class System.String)
IL_003b:    ldstr     "theStruct.Status: {0}"
IL_0040:    ldloca.s  theStruct
IL_0042:    call      instance int32 myStruct::get_status()
IL_0047:    stloc.2
IL_0048:    ldloca.s  V_2
IL_004a:    box       [mscorlib]System.Int32
IL_004f:    call      void [mscorlib]System.Console::WriteLine
    (class System.String, class System.Object)
IL_0054:    ldloca.s  theStruct
IL_0056:    box       myStruct
IL_005b:    stloc.1
IL_005c:    ldloc.1
IL_005d:    ldc.i4.4
IL_005e:    callvirt  instance void IStorable::set_status(int32)
IL_0063:    ldstr     "Changed interface"
IL_0068:    call      void [mscorlib]System.Console::WriteLine
    (class System.String)
IL_006d:    ldstr     "theStruct.Status: {0}, isTemp: {1}"
IL_0072:    ldloca.s  theStruct
IL_0074:    call      instance int32 mySystem::get_status()
IL_0079:    stloc.2
IL_007a:    ldloca.s  V_2
IL_007c:    box       [mscorlib]System.Int32
IL_0081:    ldloc.1
IL_0082:    callvirt      instance int32 IStorable::get_status()
IL_0087:    stloc.2
IL_0088:    ldloca.s  V_2
IL_008a:    box       [mscorlib]System.Int32
IL_008f:    call      void [mscorlib]System.Console::WriteLine
    (class System.String, class System.Object, class System.Object)
IL_0094:    ldloca.s  theStruct
IL_0096:    ldc.i4.6
IL_0097:    call      instance void myStruct::set_status(int32)
IL_009c:    ldstr     "Changed object"
IL_00a1:    call      void [mscorlib]System.Console::WriteLine
```

```
        (class System.String)
    IL_00a6:    ldstr       "theStruct.Status: {0}, isTemp: {1}"
    IL_00ab:    ldloca.s    theStruct
    IL_00ad:    call        instance int32 myStruct::get_status()
    IL_00b2:    stloc.2
    IL_00b3:    ldloca.s    V_2
    IL_00b5:    box         [mscorlib]System.Int32
    IL_00ba:    ldloc.1
    IL_00bb:    callvirt        instance int32 IStorable::get_status()
    IL_00c0:    stloc.2
    IL_00c1:    ldloca.s    V_2
    IL_00c3:    box         [mscorlib]System.Int32
    IL_00c8:    call        void [mscorlib]System.Console::WriteLine
        (class System.String, class System.Object, class System.Object)
    IL_00cd:    ret
} // end fo method Tester::Main
---------------------------------------------------------------------------
```

Trong dòng lệnh IL_00b, giá trị của status được thiết lập thông qua việc gọi đối tượng giá trị. Tiếp theo chúng ta thấy lệnh gọi thứ hai ở dòng IL_0017. Lưu ý rằng việc gọi WriteLine() dẫn đến việc boxing một giá trị nguyên để phương thức GetString của lớp object được gọi.

Điều muốn nhấn mạnh là ở dòng lệnh IL_0056 khi một cấu trúc myStruct đã được boxing. Việc boxing này tạo ra một kiểu dữ liệu tham chiếu cho tham chiếu giao diện. Và điều quan trọng là ở dòng IL_005e lúc này IStorable::set_status được gọi chứ không phải là myStruct::setStatus.

Điều quan trọng muốn trình bày ở đây là khi chúng ta thực thi một giao diện với một kiểu giá trị, phải chắc chắn rằng truy cập các thành viên của giao diện thông qua đối tượng hơn là thông qua một tham chiếu giao diện.

Câu hỏi và trả lời

Câu hỏi 1: So sánh giữa lớp và giao diện?

Trả lời 1: Giao diện khác với lớp ở một số điểm sau: giao diện không cung cấp bất cứ sự thực thi mã nguồn nào cả. Điều này sẽ được thực hiện tại các lớp thực thi giao diện. Một giao diện đưa ra chỉ để nói rằng có cung cấp một số sự xác nhận hướng dẫn cho những điều gì đó xảy ra và không đi vào chi tiết. Một điều khác nữa là tất cả các thành viên của giao diện được giả sử là public ngầm định. Nếu chúng ta cố thay đổi thuộc tính truy cập của thành viên trong giao diện thì sẽ nhận được lỗi.

Giao diện chỉ chứa những phương thức, thuộc tính, sự kiện, chỉ mục. Và không chứa dữ liệu thành viên, bộ khởi dựng, và bộ hủy. Chúng cũng không chứa bất cứ thành viên static nào cả.

Câu hỏi 2: Sự khác nhau giữa giao diện và lớp trừu tượng?

Trả lời 2: Sự khác nhau cơ bản là sự kế thừa. Một lớp có thể kế thừa nhiều giao diện cùng một lúc, nhưng không thể kế thừa nhiều hơn một lớp trừu tượng.

Câu hỏi 3: Các lớp thực thi giao diện sẽ phải làm gì?

Trả lời 3: Các lớp thực thi giao diện phải cung cấp các phần thực thi chi tiết cho các phương thức, thuộc tính, chỉ mục, sự kiện được khai báo trong giao diện.

Câu hỏi 4: Có bao nhiêu cách gọi một phương thức được khai báo trong giao diện?

Trả lời 4: Có 4 cách gọi phương thức được khai báo trong giao diện:

- *Thông qua lớp cơ sở tham chiếu đến đối tượng của lớp dẫn xuất*
- *Thông qua một giao diện tạo từ lớp cơ sở tham chiếu đến đối tượng dẫn xuất*
- *Thông qua một đối tượng dẫn xuất*
- *Thông qua giao diện tạo từ đối tượng dẫn xuất*

Câu hỏi 5: Các thành viên của giao diện có thể có những thuộc tính truy cập nào?

Trả lời 5: Mặc định các thành viên của giao diện là public. Vì mục tiêu của giao diện là xây dựng cho các lớp khác sử dụng. Nếu chúng ta thay đổi thuộc tính này như là internal, protected hay private thì sẽ gây ra lỗi.

Câu hỏi 6: Chúng ta có thể tạo thể hiện của giao diện một cách trực tiếp được không?

Trả lời 6: Không thể tạo thể hiện của giao diện trực tiếp bằng khai báo new được. Chúng ta chỉ có thể tạo thể hiện giao diện thông qua một phép gán với đối tượng thực thi giao diện.

Câu hỏi thêm

Câu hỏi 1: Toán tử is được dùng làm gì trong giao diện?

Câu hỏi 2: Toán tử as có lợi hơn toán tử is về mặt nào khi được sử dụng liện quan đến giao diện?

Câu hỏi 3: Giao diện là kiểu dữ liệu tham chiếu hay kiểu giá trị?

Câu hỏi 4: Khi thực thi giao diện với cấu trúc. Thì truy cập các thành viên của giao diện thông qua đối tượng hay thông qua tham chiếu giao diện là tốt nhất?

Câu hỏi 5: Số giao diện có thể được kế thừa cho một lớp?

Câu hỏi 6: Việc thực thi giao diện tường minh là thực thi như thế nào? Trong trường hợp nào thì cần thực hiện tường minh?

Bài tập

Bài tập 1: Hãy viết một giao diện khai báo một thuộc tính ID chứa chuỗi giá trị. Viết một lớp Employee thực thi giao diện đó.

Bài tập 2: Đoạn mã nguồn sau đây có lỗi hãy sử lỗi và hãy cho biết tại sao có lỗi này. Sau khi sửa lỗi hãy viết một lớp Circle thực thi giao diện này?

```
public interface IDimensions
{
```

```
    long width;
    long height;
    double Area();
    double Circumference();
    int Side();
}
```

Bài tập 3: Chương trình sau đây có lỗi hãy sử lỗi, biên dịch và chạy lại chương trình? Giải thích tại sao chương trình có lỗi.

```
using System;
interface IPoint
{
    // Property signatures:
    int x
    {
        get;
        set;
    }
    int y
    {
        get;
        set;
    }
}
class MyPoint : IPoint
{
    // Fields:
    private int myX;
    private int myY;
    // Constructor:
    public MyPoint(int x, int y)
    {
        myX = x;
        myY = y;
    }
    // Property implementation:
    public int x
```

```
    {
        get
        {
            return myX;
        }
        set
        {
            myX = value;
        }
    }
    public int y
    {
        get
        {
            return myY;
        }
        set
        {
            myY = value;
        }
    }
}
class MainClass
{
    private static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.x, p.y);
    }
    public static void Main()
    {
        MyPoint p = new MyPoint(2,3);
        Console.Write("My Point: ");
        PrintPoint(p);
        IPoint p2 = new IPoint();
        PrintPoint(p2);
    }
}
```

<u>Bài tập 4</u>: Xây dựng một giao diện IDisplay có khai báo thuộc tính Name kiểu chuỗi. Hãy viết hai lớp Dog và Cat thực thi giao diện IDisplay, cho biết thuộc tính Name là tên của đối tượng.

Chương 9

MẢNG, CHỈ MỤC, VÀ TẬP HỢP

- **Mảng**
  - **Khai báo mảng**
  - **Giá trị mặc định**
  - **Truy cập các thành phần trong mảng**
  - **Khởi tạo thành phần trong mảng**
  - **Sử dụng từ khóa params**
- **Câu lệnh lặp foreach**
- **Mảng đa chiều**
  - **Mảng đa chiều cùng kích thước**
  - **Mảng đa chiều kích thước khác nhau**
  - **Chuyển đổi mảng**
  - **System.Array**
- **Bộ chỉ mục**
  - **Bộ chỉ mục và phép gán**
  - **Sử dụng kiểu chỉ số khác**
- **Giao diện tập hợp**
- **Câu hỏi & bài tập**

Môi trường .NET cung cấp rất đa dạng số lượng các lớp về tập hợp, bao gồm: Array, ArrayList, Queue, Stack, BitArray, NameValueCollection, và StringCollection.

Trong số đó tập hợp đơn giản nhất là Array, đây là kiểu dữ liệu tập hợp mà ngôn ngữ C# hỗ trợ xây dựng sẵn. Chương này chúng ta sẽ tìm hiểu cách làm việc với mảng một chiều, mảng đa chiều, và mảng các mảng (jagged array). Chúng ta cũng được giới thiệu phần chỉ mục indexer, đây là cách thiết lập để làm cho việc truy cập những thuộc tính giống nhau trở nên đơn giản hơn, một lớp được chỉ mục thì giống như một mảng.

.NET cũng cung cấp nhiều các giao diện, như IEnumerable và ICollection. Những phần thực thi của các giao diện này cung cấp các tiêu chuẩn để tương tác với các tập hợp. Trong chương này chúng ta sẽ được cách sử dụng hiệu quả của các giao diện. Cũng thông qua chương này chúng ta sẽ được giới thiệu cách sử dụng chung của các tập hợp trong .NET, bao gồm: ArrayList, Dictionary, Hashtable, Queue, và Stack.

Mảng

Mảng là một tập hợp có thứ tự của những đối tượng, tất cả các đối tượng này cùng một kiểu. Mảng trong ngôn ngữ C# có một vài sự khác biệt so với mảng trong ngôn ngữ C++ và một số ngôn ngữ khác, bởi vì chúng là những đối tượng. Điều này sẽ cung cấp cho mảng sử dụng các phương thức và những thuộc tính.

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

Thành viên	Mô tả
BinarySearch()	Phương thức tĩnh public tìm kiếm một mảng một chiều đã sắp thứ tự.
Clear()	Phương thức tĩnh public thiết lập các thành phần của mảng về 0 hay null.
Copy()	Phương thức tĩnh public đã nạp chồng thực hiện sao chép một vùng của mảng vào mảng khác.
CreateInstance()	Phương thức tĩnh public đã nạp chồng tạo một thể hiện mới cho mảng
IndexOf()	Phương thức tĩnh public trả về chỉ mục của thể hiện đầu tiên chứa giá trị trong mảng một chiều
LastIndexOf()	Phương thức tĩnh public trả về chỉ mục của thể hiện cuối cùng của giá trị trong mảng một chiều
Reverse()	Phương thức tĩnh public đảo thứ tự của các thành phần trong mảng một chiều
Sort()	Phương thức tĩnh public sắp xếp giá trị trong mảng một chiều.
IsFixedSize	Thuộc tính public giá trị bool thể hiện mảng có kích thước cố định hay không.
IsReadOnly	Thuộc tính public giá trị bool thể hiện mảng chỉ đọc hay không

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
SyncRoot	Thuộc tính public chứa đối tượng dùng để đồng bộ truy cập trong mảng
GetEnumerator()	Phương thức public trả về IEnumerator
GetLength()	Phương thức public trả về kích thước của một chiều cố định trong mảng
GetLowerBound()	Phương thức public trả về cận dưới của chiều xác định trong mảng
GetUpperBound()	Phương thức public trả về cận trên của chiều xác định trong mảng
Initialize()	Khởi tạo tất cả giá trị trong mảng kiểu giá trị bằng cách gọi bộ khởi dụng mặc định của từng giá trị.
SetValue()	Phương thức public thiết lập giá trị cho một thành phần xác định trong mảng.

Bảng 9.1: Các phương thức và thuộc tính của System.Array.

Khai báo mảng

Chúng ta có thể khai báo một mảng trong C# với cú pháp theo sau:

```
<kiểu dữ liệu>[] <tên mảng>
```

Ví dụ 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.

✍*Ghi chú*: 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.

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];
```

✍*Ghi chú*: 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.

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 9.1 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.

Ví dụ 9.1: 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.

Khởi tạo thành phần của 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 9.3 cung cấp tất cả mã nguồn để minh họa sử dụng cú pháp **params**.

Ví dụ 9.3: 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
```

Câu lệnh lặp foreach

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

Cú pháp sử dụng lệnh lặp **foreach** như sau:

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

Do vậy, chúng ta có thể cải tiến ví dụ 9.1 trước bằng cách thay việc sử dụng vòng lặp **for** bằng vòng lặp **foreach** để truy cập đến từng thành phần trong mảng.

Ví dụ 9.2: 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 9.2 cũng tương tự như ví dụ 9.1. 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.

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.

Ví dụ 9.4: 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];
```

Lưu ý 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 9.5 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.

Ví dụ 9.5: 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 9.4, 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> [] [] ...
```

Ví dụ, 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 9.6 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.

Ví dụ 9.6: 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][];
```

Chú ý rằng 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.

✍*Ghi chú*: 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 9.7 minh họa việc chuyển đổi một mảng kiểu Button đến một mảng những đối tượng.

🗗 *Ví dụ 9.7: 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 9.7 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 9.8 minh họa việc sử dụng hai phương thức để thao tác đối tượng chuỗi.

Ví dụ 9.8: 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

Bộ chỉ mục

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 9.9 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.

✍*Ghi chú*: 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.

🗗 *Ví dụ 9.9: 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 viện 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 9.10 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 9.9 trước vẫn còn tồn tại.

Ví dụ 9.10: 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
            }
            return strings[index];
        }
        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 9.10 thì tương tự như ví dụ 9.9 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 chuỗi 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 9.10 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	Mục đích
IEnumerable	Liệt kê thông qua một tập hợp bằng cách sử dụng **foreach**.
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 để sắp 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 như Hashtable và SortedList.
IDictionaryEnumerator	Cho phép liệt kê dùng câu lệnh **foreach** qua tập hợp hỗ trợ IDictionary.

Bảng 9.2: Giao diện cho tập hợp.

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.

Ví dụ 9.11: 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.

Danh sách mảng

Một vấn đề hạn chế của kiểu dữ liệu mảng là kích thước cố định. Nếu chúng ta không biết trước số lượng đối tượng trong một mảng sẽ được lưu giữ, thì sẽ khó khăn vì có thể chúng ta khai báo kích thước của mảng quá nhỏ (vượt quá kích thước lưu trữ của mảng) hoặc là kích thước quá lớn (dẫn đến lãng phí bộ nhớ). Chương trình của chúng ta có thể hỏi người dùng về kích thước, hoặc thu những input từ trong một web site.Tuy nhiên việc xác định số lượng của đối tượng trong những session có tính chất tương tác động là không thích hợp. 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:

Phương thức- thuộc tính	Mục đích
Adapter()	Phương thức static tạo một wrapper ArrayList cho đối tượng 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 thể thêm 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 thành phần 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 cố định 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 hay không
Item()	Thiết lập hay truy cập thành phần trong mảng tại vị trí xác định. Đây là bộ chỉ mục cho lớp ArrayList.
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồng bộ 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 một ICollection vào cuối của ArrayList
BinarySearch()	Phương thức nạp chồng public sử dụng tìm kiếm nhị phận để định vị một thành phần xác định trong ArrayList được sắp 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 hay không
CopyTo()	Phương thức public nạp chồng để sao chép một ArrayList đến một mảng một chiều.
GetEnumerator()	Phương thức public nạp chồng trả về một enumerator dùng để 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 tiên xuất 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 những thành phần trong ArrayList.
Sort()	Sắp xếp ArrayList.
ToArray()	Sao chép những thành phần của ArrayList đến một mảng mới.
TrimToSize()	Thiết lập kích thước thật sự chứa các thành phần trong ArrayList

Bảng 9.3: Các phương thức và thuộc tính của ArrayList

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 9.12 sau minh họa sử dụng ArrayList.

Ví dụ 9.12: 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 co 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 9.13 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.

Ví dụ 9.13: 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 Employee-Comparer, 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 9.14 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.

Ví dụ 9.14: 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)

Hàng đợi là một tập hợp trong đó có thứ tự vào trước và ra trước (FIFO). Tương tự như là những người mua vé tàu, họ xếp thành một hàng, người nào vào trước thì sẽ mua trước và ra trước. 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.

Phương thức- thuộc tính	Mục đích
Synchronized()	Phương thức static trả về một Queue wrapper được thread-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
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồng bộ 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.
CopyTo()	Sao chép những thành phần của hàng đợi đến mảng một chiều 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

Bảng 9.4: Những phương thức và thuộc tính của Queue.

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 9.15 minh họa việc sử dụng hàng đợi.

Ví dụ 9.15: 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- thuộc tính	Mục đích
Synchronized()	Phương thức static trả về một Stack wrapper được 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 xác định ngăn xếp được đồng bộ
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồng bộ truy cập Stack.
Clear()	Xóa tất cả các thành phần trong ngăn xếp
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong mảng.
CopyTo()	Sao chép những thành phần của ngăn xếp đến mảng một chiều 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 không xóa nó.
ToArray()	Sao chép những thành phần qua một mảng mới

Bảng 9.5 : Phương thức và thuộc tính của lớp Stack.

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.

Ví dụ 9.16: 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 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();
```

Kiểu từ điển

Từ điển là kiểu tập hợp trong đó có hai thành phần chính liên hệ với nhau là khóa và giá trị. Trong từ điển ngôn ngữ như Oxford thì sự liên hệ giữa từ (khóa) và phần định nghĩa từ (giá trị). 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. Ví dụ, 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 9.6 dưới.

Phương thức- thuộc tính	Mục đích
Synchronized()	Phương thức static trả về một Hashtable wrapper được thread-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 để đồng 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 giá trị trong hashtable.
Add()	Thêm một thành phần mới với khóa và giá trị xác định.
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 xác định
CopyTo()	Sao chép những thành phần của hashtable đến mảng một chiều đã tồn tại
GetEnumerator()	Trả về một enumerator cho hashtable.
GetObjectData()	Thực thi ISerializable và trả về dữ liệu cần thiết để lưu trữ.
OnDeserialization	Thực thi ISerializable và phát sinh sự kiện deserialization khi hoàn thành.
Remove()	Xóa một thành phần với khóa xác định.

Bảng 9.6: Phương thức và thuộc tính của lớp Hashtable.

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 9.17 minh họa việc thêm một item vào trong bảng Hashtable và sau đó truy cập lại chúng thông qua thuộc tính Item.

Ví dụ 9.17: 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 9.17 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 9.18 minh họa như sau.

Ví dụ 9.18 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 9.19 minh họa như sau:

Ví dụ 9.16: 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
```

Câu hỏi và trả lời

Câu hỏi 1: Điều phân biệt giữa mảng và các thành phần bên trong một mảng?

Trả lời 1: Mảng là kiểu dữ liệu tham chiếu, còn các thành phần bên trong mảng được cấp phát dựa theo kiểu dữ liệu của chúng. Do vậy một mảng của kiểu dữ liệu tham chiếu sẽ không chứa giá trị gì cả mà chỉ tham chiếu đến những thành phần được tạo ra trên heap.

Câu hỏi 2: Một lớp có bộ chỉ mục khác một mảng như thế nào?

Trả lời 2: Hoàn toàn khác nhau, một mảng chỉ đơn thuần là một đối tượng tham chiếu đến những đối tượng khác cùng kiểu dữ liệu. Trong khi một lớp có bộ chỉ mục thì nó chứa một mảng các giá trị nào đó, và cho phép bên ngoài truy cập mảng này thông qua bộ chỉ mục. Một lớp như vậy không chỉ có một mảng đơn thuần mà còn có những thuộc tính khác, các phương thức...Nói chung là nếu ta chỉ cần thao tác đơn thuần trên từng phần riên lẻ của một mảng thì nên dùng mảng. Còn nếu chúng ta cần thực hiện một số chức năng nào đó có liên quan tới một mảng thì ta có thể xây dựng một lớp có chứa một mảng và hỗ trợ bộ chỉ mục.

Câu hỏi 3: Giao diện tập hợp là gì? Có phải .NET cung cấp một số giao diện chuẩn hay không?

Trả lời 3: Giao diện tập hợp cũng là một giao diện nhưng nó chỉ đưa ra các quy định thao tác trên tập hợp như: so sánh, liệt kê trên tập hợp, tạo các tập hợp... NET cung cấp một số giao diện cho tập hợp như: IEnumerable, ICollection, IComparer, IList....

Câu hỏi thêm

Câu hỏi 1: Từ khoá params được sử dụng làm gì?

Câu hỏi 2: Ý nghĩa của lệnh lặp foreach? Lệnh này được sử dụng với kiểu dữ liệu nào?

Câu hỏi 3: Có mấy kiểu mảng đa chiều trong ngôn ngữ C#. Hãy cho biết từng loại và khi nào thì sử dụng từng loại cho thích hợp.

Câu hỏi 4: Cách tạo ra mảng đa chiều không cùng kích thước?

Câu hỏi 5: Hãy cho biết sự khác nhau giữa hai cách gọi Arr[i][j] và Arr[i, j]?

Câu hỏi 6: Có thể dùng lệnh foreach để xuất ra tất cả các thành phần của mảng đa chiều không cùng kích thước hay không? Nếu được thì phải làm như thế nào?

Câu hỏi 7: Kiểu dữ liệu nào có thể làm chỉ mục trong bộ chỉ mục của một lớp?

Câu hỏi 8: Làm thế nào để biết kích thước của một mảng?

Câu hỏi 9: Liệt kê những giao diện tập hợp mà .NET cung cấp? Cho biết ý nghĩa của từng giao diện?

Câu hỏi 10: Có cách nào tạo một mảng mà không cần khai báo trước kích thước của mảng? Và trong quá trình thực hiện trên mảng chúng ta có thể tăng động kích thước của mảng hay không?

Câu hỏi 11: Nếu mảng có 31 phần tử thì dung lượng của đối tượng ArrayList là bao nhiêu? Trường hợp có 33 phần tử?

Câu hỏi 12: Hàng đợi là gì? Chúng được sắp xếp theo kiểu thứ tự nào? Ứng dụng của hàng đợi ?

Câu hỏi 13: Ngăn xếp là gì? Chúng được sắp xếp theo kiểu thứ tự nào? Ứng dụng của ngăn xếp?

Câu hỏi 14: Phương thức Peek() trong hàng đợi và ngăn xếp có ý nghĩa gì?

Câu hỏi 15: Kiểu dữ liệu nào cho phép truy cập một giá trị thông qua khóa của nó? Lớp nào trong .NET hỗ trợ kiểu dữ liệu này?

Câu hỏi 16: Cách lấy tập giá trị trong một đối tượng Hashtable?
Câu hỏi 17: Cách lấy tập khóa trong một đối tượng Hastable?
Câu hỏi 18: Khóa có phải là duy nhất trong một Hastable hay không?
Câu hỏi 19: Nếu hai vùng có chung một khóa thì chúng được tìm kiếm theo kiểu nào? Và tốc độ tìm kiếm?
Câu hỏi 20: Hashtable thực thi các giao diện tập hợp nào?
Câu hỏi 21: Phương thức nào thực hiện việc tạo các khoá trong một Hashtable?

Bài tập

Bài tập 1: Viết một chương trình tạo một mảng một chiều nguyên chứa giá trị ngẫu nhiên. Sắp xếp các thành phần trong mảng theo thứ tự tăng dần và hiển thị kết quả. Làm tương tự với trường hợp sắp xếp mảng theo thứ tự giảm dần.
Bài tập 2: Viết một chương trình tạo một mảng một chiều nguyên chứa giá trị ngẫu nhiên. Sắp xếp chúng theo thứ tự số âm thì tăng còn số dương thì giảm dần. Hiển thị kết quả ra màn hình.
Bài tập 3: Viết một chương trình tìm số lớn nhất và nhỏ nhất trong mảng hai chiều có kích thước cố định. Các thành phần của mảng được phát sinh ngẫu nhiên.
Bài tập 4: Viết chương trình cộng hai ma trận nxm, tức là mảng hai chiều có kích thước n dòng, m cột. Các giá trị của hai mảng phát sinh ngẫu nhiên, cho biết kết quả cộng hai ma trận?
Bài tập 4: Viết chương trình cho phép người dùng nhập vào một ma trận nxm, sao đó tìm kiếm một giá trị nào đó theo yêu cầu người dùng, kết quả của việc tìm kiếm là giá trị và thứ tự của giá trị tìm được trong ma trận.
Bài tập 5: Viết chương trình tạo một mảng hai chiều không cùng kích thước. Cố định số dòng của mảng là 5, còn từng dòng có kích thước bằng giá trị của dòng, tứ là dòng thứ nhất có kích thước 1 (tức là có 1 cột), dòng thứ hai có kích thước là 2 (tức là 2 cột).... Các giá trị phát sinh ngẫu nhiên. Hãy xuất kết quả của ma trận theo kiểu sau:

a[i][j] = <giá trị aij>
...

Việc xuất kết quả của ma trận trên có thể thực hiện bằng vòng lặp foreach được không? Nếu được thì hãy viết đoạn chương trình xuất ra kết quả?
Bài tập 6: Viết chương trình tạo ra một mảng lưu trữ 30 điểm số của học sinh. Tính trung bình điểm của tất cả học sinh. Xuất kết quả từng điểm và điểm trung bình.
Bài tập 7: Viết một chương trình tạo ra một lớp tên là LopHoc, trong đó có khai báo bộ chỉ mục chỉ đến tên của từng học viên trong lớp. Cho phép một lớp có tối đa 30 học viên. Tạo chương trình minh họa cho phép người dùng nhập vào tên của từng học viên. Xuất kết quả danh sánh học viên của lớp thông qua bộ chỉ mục.

Bài tập 8: Viết chương trình sử dụng ArrayList để tạo một mảng. Chương trình tạo ra một vòng lặp cho phép người dùng nhập vào các giá trị cho mảng. Hãy xuất kết quả mảng cùng với giá trị Count, và Capacity của mảng. Ta có thể thiết lập giá trị Capacity nhỏ hơn giá trị Count được không?

Bài tập 9: Viết chương trình tạo ra đối tượng Queue tên là myQueue. Khởi tạo myQueue có 5 giá trị ngẫu nhiên. Hãy thực hiện các bước sau, mỗi bước thực hiện phải xuất tình trạng của myQueue:

1. *Lấy một giá trị ra.*
2. *Lấy tiếp một giá trị nữa.*
3. *Xem một giá trị ở đầu queue.*
4. *Đưa vào queue một giá trị.*

Bài tập 10: Viết chương trình tạo đối tượng Stack tên là myStack. Khởi tạo myStack có 5 giá trị ngẫu nhiên. Hãy thực hiện các bước sau, mỗi bước thực hiện phải xuất tình trạng của myStack:

1. *Lấy một giá trị ra.*
2. *Lấy tiếp một giá trị nữa.*
3. *Xem một giá trị ở đầu stack.*
4. *Đưa vào stack một giá trị.*

Bài tập 11: Viết chương trình sử dụng kiểu dữ liệu từ điển để quản lý thông tin của một lớp học. Trong đó khóa là chuỗi mã số học viên còn giá trị là tên của học viên. Viết chương trình minh họa cho phép nhập vào 10 học viên, và cho phép người dùng tìm kiếm tên của học viên thông qua mã số học viên.

Chương 10

XỬ LÝ CHUỖI

- **Lớp đối tượng string**
  - **Tạo một chuỗi**
  - **Tạo chuỗi dùng phương thức ToString**
  - **Thao tác trên chuỗi**
  - **Tìm một chuỗi con**
  - **Chia chuỗi**
  - **Thao tác trên chuỗi dùng StringBuilder**
- **Các biểu thức quy tắc**
  - **Sử dụng biểu thức quy tắc qua lớp Regex**
  - **Sử dụng Regex để tìm tập hợp**
  - **Sử dụng Regex để gom nhóm**
  - **Sử dụng lớp CaptureCollection**
- **Câu hỏi & bài tập**

Có một thời gian người ta luôn nghĩ rằng máy tính chỉ dành riêng cho việc thao tác các giá trị dạng số. Các máy tính đầu tiên là được thiết kế để sử dụng tính toán số lượng lớn như tính toán quỹ đạo của tên lửa trong quốc phòng. Và ngôn ngữ lập trình được giảng dạy ở khoa toán của các đại học lớn.

Ngày nay, hầu hết các chương trình liên quan đến nhiều chuỗi ký tự hơn là các chuỗi các con số. Thông thường các chuỗi được sử dụng cho việc xử lý từ ngữ, thao tác trên các sưu liệu, và tạo ra các trang web.

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.

Lớp đối tượng String

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(),...

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 dầu 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.

Tạo chuỗi dùng phương thức ToString của đối tượng

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).

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:

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 không quan tâm đến thứ tự.
Concat()	Phương thức public static để tạo chuỗi mới từ một hay nhiều chuỗi.
Copy()	Phương thức public static tạo ra một chuỗi mới bằng sao từ chuỗi khác.
Equal()	Phương thức public static kiểm tra xem hai chuỗi có cùng giá trị hay không.
Format()	Phương thức public static định dạng một chuỗi dùng ký tự lệnh định dạng xác định.
Intern()	Phương thức public static trả về tham chiếu đến thể hiện của chuỗi.
IsInterned()	Phương thức public static trả về tham chiếu của chuỗi
Join()	Phương thức public static kết nối các chuỗi xác định giữa 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 ký tự Unicode.
EndsWidth()	Chỉ ra vị trí của chuỗi xác định phù hợp với chuỗi đưa ra.
Insert()	Trả về chuỗi mới đã được chèn một chuỗi xác định.
LastIndexOf()	Chỉ ra vị trí xuất hiện cuối cùng của một chuỗi xác định trong chuỗi.
PadLeft()	Canh lề phải những ký tự trong chuỗi, chèn vào bên trái khoảng trắng hay các ký tự xác định.
PadRight()	Canh lề trái những ký tự trong chuỗi, chèn vào bên phải 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 định trong chuỗi.
StartWidth()	Xem chuỗi có bắt đầu bằng một số ký tự xác định hay không.
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.
Trim()	Xóa bỏ tất cả sự xuất hiện của tập hợp ký tự xác định từ vị trí đầu 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.

Bảng 10.1 : Phương thức và thuộc tính của lớp String

Trong ví dụ 10.1 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.

Ví dụ 10.1 : 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 10.1 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
```

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 10.2 minh họa việc sử dụng phương thức Substring() của chuỗi.

Ví dụ 10.2 : 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 10.2 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.

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.

Ví dụ 10.3 : 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.

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:

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 dạng của một đối tượng.
EnsureCapacity()	Đảm bảo rằng StringBuilder hiện thời có khả năng tối thiểu lớn như một giá trị xác định.
Insert()	Chèn một đối tượng vào một vị trí xác định
Replace()	Thay thế tất cả thể hiện của một ký tự xác định với những ký tự mới.

Bảng 10.2 Phương thức của lớp StringBuilder

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 10.4 thay thế đối tượng String bằng một đối tượng StringBuilder.

Ví dụ minh họa 10.4 : 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 10.3. 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 10.3 dùng String:

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

Các biểu thức quy tắc (Regular Expression)

Biểu thức qui tắc là một ngôn ngữ mạnh dùng để mô tả và thao tác văn bản. Một biểu thức qui tắc thường được áp dụng cho một chuỗi, hay một tập hợp các ký tự. Thông thường một chuỗi là toàn bộ văn bản hay tài liệu.

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 10.5.

Ví dụ minh họa 10.5: 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 10.4.

```
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 10.4 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 10.5. 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 10.6 sử dụng phương thức tĩnh

Ví dụ minh họa 10.6: 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 10.6 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 10.7 dưới đây.

Ví dụ minh họa 10.7: 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 10.7 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. Ví dụ, 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 10.8 như sau:

Ví dụ minh họa 10.8: 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 10.8 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.

Ví dụ minh họa 10.9: 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.

Câu hỏi và trả lời

Câu hỏi 1: Những tóm tắt cơ bản về chuỗi?

Trả lời 1: Chuỗi là kiểu dữ liệu thường được sử dụng nhất trong lập trình. Trong ngôn ngữ C#, chuỗi được hỗ trợ rất mạnh thông qua các lớp về chuỗi và biểu thức quy tắc. Chuỗi là kiểu dữ liệu tham chiếu, chứa các ký tự Unicode. Các thao tác trên đối tượng chuỗi không làm thay đổi giá trị của chuỗi mà ta chỉ nhận được kết quả trả về. Tuy nhiên, C# cung cấp lớp StringBuilder cho phép thao tác trực tiếp để bổ sung chuỗi.

Câu hỏi 2: Biểu thức quy tắc là gì?

Trả lời 2: Biểu thức quy tắc là ngôn ngữ dùng để mô tả và thao tác văn bản. Một biểu thức quy tắc thường được áp dụng cho một chuỗi văn bản hay toàn bộ tài liệu nào đó. Kết quả của việc áp dụng một biểu thức quy tắc là ta nhận được một chuỗi kết quả, chuỗi này có thể là chuỗi con của chuỗi áp dụng hay có thể là một chuỗi mới được bổ sung từ chuỗi ban đầu.

Câu hỏi 3: Thao tác thường xuyên thực hiện trên một chuỗi là thao tác nào?

Trả lời 3: Như nó bên trên, thao tác thường xuyên thực hiện trên một chuỗi là tìm kiếm chuỗi con thỏa quy tắc nào đó. Một ngôn ngữ nếu mạnh về thao tác trên chuỗi, chắc chắn phải cung cấp nhiều phương thức thao tác tốt để tìm kiếm các chuỗi con theo quy tắc. Ngôn ngữ C# cũng rất mạnh về điểm này, do chúng thừa hưởng từ các lớp thao tác trên chuỗi của . NET.

Câu hỏi thêm

Câu hỏi 1: Có bao nhiêu cách tạo chuỗi trong ngôn ngữ C#?

Câu hỏi 2: Chuỗi Verbatim là chuỗi như thế nào? Hãy cho một vài ví dụ minh họa về chuỗi này và diễn giải ý nghĩa của chúng?

Câu hỏi 3: Sự khác nhau cơ bản giữa một chuỗi tạo từ đối tượng string và StringBuilder?

Câu hỏi 4: Khi nào thì nên dùng chuỗi tạo từ lớp string và StringBuilder?

Câu hỏi 5: Một biểu thức quy tắc có bao nhiêu kiểu ký tự?

Câu hỏi 6: Một biểu thức quy tắc sau đây so khớp điều gì?

```
^(Name|Address|Phone|Fax):
```

Bài tập

Bài tập 1: Viết chương trình cho phép người dùng nhập vào một chuỗi. Sau đó đếm số ký tự xuất hiện của từng ký tự trong chuỗi như ví dụ sau:

'a' : 2
'g' : 5
'2' : 1
....

Bài tập 2: Viết chương trình tìm một chuỗi con trong một chuỗi cho trước. Chương trình cho phép người dùng nhập vào một chuỗi, và chuỗi con cần tìm. Kết quả là chuỗi con có tìm thấy hay không, nếu tìm thấy thì hãy đưa ra vị trí đầu tiên tìm thấy.

Bài tập 3: Viết chương trình tìm số lần xuất hiện một chuỗi con trong một chuỗi cho trước. Chương trình cho phép người dùng nhập vào một chuỗi và chuỗi con cần đếm. Kết quả hiển thị chuỗi, chuỗi con và các vị trí mà chuỗi con xuất hiện trong chuỗi.

Bài tập 4: Viết chương trình cho phép người dùng nhập vào một chuỗi, rồi thực hiện việc đảo các ký tự trong chuỗi theo thứ tự ngược lại.

Bài tập 5: Viết chương trình cắt các từ có nghĩa trong câu. Ví dụ như cho từ: "Thuc hanh lap trinh" thì cắt thành 4 chữ: "Thuc", "hanh", "lap", "trinh".

Bài tập 6: Hãy viết chương trình sử dụng biểu thức quy tắc để lấy ra chuỗi ngày/tháng/năm trong một chuỗi cho trước? Cho phép người dùng nhập vào một chuỗi rồi dùng biểu thức quy tắc vừa tạo ra thực hiện việc tìm kiếm.

Bài tập 7: Hãy viết chương trình sử dụng biểu thức quy tắc để lấy ra thời gian giờ:phút:giây trong một chuỗi cho trước? Chương trình cho phép người dùng nhập vào một chuỗi rồi dùng biểu thức quy tắc vừa tạo để thực hiện việc tìm kiếm.

Chương 11

CƠ CHẾ ỦY QUYỀN - SỰ KIỆN

- **Ủ y quyền**
  - **Sử dụng ủy quyền để xác nhận phương thức lúc thực thi**
  - **Ủ y quyền tĩnh**
  - **Dùng ủy quyền như thuộc tính**
  - **Thiết lập thứ tự thi hành với mảng ủy quyền**
  - **Muticasting**
- **Sự kiện**
  - **Cơ chế publishing – subscribing**
  - **Sự kiện & ủy quyền**
- **Câu hỏi & bài tập**

Trong lập trình chúng ta thường đối diện với tình huống là khi chúng ta muốn thực hiện một hành động nào đó, nhưng hiện tại thì chưa xác định được chính xác phương thức hay sự kiện trong đối tượng. Ví dụ như một nút lệnh button biết rằng nó phải thông báo cho vài đối tượng khi nó được nhấn, nhưng nó không biết đối tượng hay nhiều đối tượng nào cần được thông báo. Tốt hơn việc nối nút lệnh với đối tượng cụ thể, chúng ta có thể kết nối nút lệnh đến một cơ chế ủy quyền và sau đó thì chúng ta thực hiện việc ủy quyền đến phương thức cụ thể khi thực thi chương trình.

Trong thời kỳ đầu của máy tính, chương trình được thực hiện theo trình tự xử lý từng bước tuần tự cho đến khi hoàn thành, và nếu người dùng thực hiện một sự tương tác thì sẽ làm hạn chế sự điều khiển hoạt động khác của chương trình cho đến khi sự tương tác với người dùng chấm dứt.

Tuy nhiên, ngày nay với mô hình lập trình giao diện người dùng đồ họa (GUI: Graphical User Interface) đòi hỏi một cách tiếp cận khác, và được biết như là lập trình điều khiển sự kiện (event-driven programming). Chương trình hiện đại này đưa ra một giao diện tương tác với người dùng và sau đó thì chờ cho người sử dụng kích hoạt một hành động nào đó. Người sử dụng có thể thực hiện nhiều hành động khác nhau như: chọn các mục chọn trong menu, nhấn một nút lệnh, cập nhật các ô chứa văn bản,...Mỗi hành động như vậy sẽ dẫn đến một sự

kiện (event) được sinh ra. Một số các sự kiện khác cũng có thể được xuất hiện mà không cần hành động trực tiếp của người dùng. Các sự kiện này xuất hiện do các thiết bị như đồng hồ của máy tính phát ra theo chu kỳ thời gian, thư điện tử được nhận, hay đơn giản là báo một hành động sao chép tập tin hoàn thành,...

Một sự kiện được đóng gói như một ý tưởng "*chuyện gì đó xảy ra*" và chương trình phải đáp ứng lại với sự kiện đó. Cơ chế sự kiện và ủy quyền gắn liền với nhau, bởi vì khi một sự kiện xuất hiện thì cần phải phân phát sự kiện đến trình xử lý sự kiện tương ứng. Thông trường một trình xử lý sự kiện được thực thi trong C# như là một sự ủy quyền.

Ủ y quyền cho phép một lớp có thể yêu cầu một lớp khác làm một công việc nào đó, và khi thực hiện công việc đó thì phải báo cho lớp biết. Ủy quyền cũng co thể được sử dụng để xác nhận những phương thức chỉ được biết lúc thực thi chương trình, và chúng ta sẽ tìm hiểu kỹ vấn đề này trong phần chính của chương.

Ủ y quyền (delegate)

Trong ngôn ngữ C#, ủy quyền là lớp đối tượng đầu tiên (first-class object), được hỗ trợ đầy đủ bởi ngôn ngữ lập trình. Theo kỹ thuật thì ủy quyền là kiểu dữ liệu tham chiếu được dùng để đóng gói một phương thức với tham số và kiểu trả về xác định. Chúng ta có thể đóng gói bất cứ phương thức thích hợp nào vào trong một đối tượng ủy quyền. Trong ngôn ngữ C++ và những ngôn ngữ khác, chúng ta có thể làm được điều này bằng cách sử dụng con trỏ hàm (function pointer) và con trỏ đến hàm thành viên. Không giống như con trỏ hàm như trong C/C++, ủy quyền là hướng đối tượng, kiểu dữ liệu an toàn (type-safe) và bảo mật.

Một điều thú vị và hữu dụng của ủy quyền là nó không cần biết và cũng không quan tâm đến những lớp đối tượng mà nó tham chiếu tới. Điều cần quan tâm đến những đối tượng đó là các đối mục của phương thức và kiểu trả về phải phù hợp với đối tượng ủy quyền khai báo.

Để tạo một ủy quyền ta dùng từ khóa **delegate** theo sau là kiểu trả về tên phương thức được ủy quyền và các đối mục cần thiết:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

Khai báo trên định nghĩa một ủy quyền tên là WhichIsFirst, nó sẽ đóng gói bất cứ phương thức nào lấy hai tham số kiểu object và trả về giá trị int.

Một khi mà ủy quyền được định nghĩa, chúng ta có thể đóng gói một phương thức thành viên bằng việc tạo một thể hiện của ủy quyền này, truyền vào trong một phương thức có khai báo kiểu trả về và các đối mục cần thiết.

✍*Lưu ý*: Từ phần này về sau chúng ta quy ước có thể sử dụng qua lại giữa hai từ uỷ quyền và delegate với nhau.

Sử dụng ủy quyền để xác nhận phương thức lúc thực thi

Ủ y quyền như chúng ta đã biết là được dùng để xác định những loại phương thức có thể được dùng để xử lý các sự kiện và để thực hiện callback trong chương trình ứng dụng. Chúng

cũng có thể được sử dụng để xác định các phương thức tĩnh và các instance của phương thức mà chúng ta không biết trước cho đến khi chương trình thực hiện.

Giả sử minh họa như sau, chúng ta muốn tạo một lớp chứa đơn giản gọi là Pair lớp này lưu giữ và sắp xếp hai đối tượng được truyền vào cho chúng. Tạm thời lúc này chúng ta cũng không thể biết loại đối tượng mà một Pair lưu giữ. Nhưng bằng cách tạo ra các phương thức bên trong các đối tượng này thực hiện việc sắp xếp và được ủy quyền, chúng ta có thể ủy quyền thực hiện việc sắp thứ tự cho chính bản thân của đối tượng đó.

Những đối tượng khác nhau thì sẽ sắp xếp khác nhau. Ví dụ, một Pair chứa các đối tượng đếm có thể được sắp xếp theo thứ tự số, trong khi đó một Pair nút lệnh button có thể được sắp theo thứ tự alphabe tên của chúng. Mong muốn của người tạo ra lớp Pair là những đối tượng bên trong của Pair phải có trách nhiệm cho biết thứ tự của chúng cái nào là thứ tự đầu tiên và thứ hai. Để làm được điều này, chúng ta phải đảm bảo rằng các đối tượng bên trong Pair phải cung cấp một phương thức chỉ ra cho chúng ta biết cách sắp xếp các đối tượng.

Chúng ta định nghĩa phương thức yêu cầu bằng việc tạo một ủy quyền, ủy quyền này định nghĩa ký pháp và kiểu trả về của phương thức đối tượng (như button) để cung cấp và cho phép Pair xác định đối tượng nào đến trước đầu tiên và đối tượng nào là thứ hai.

Lớp Pair định nghĩa một ủy quyền, WhichIsFirst. Phương thức Sort sẽ lấy một tham số là thể hiện của WhichIsFirst. Khi một đối tượng Pair cần biết thứ tự của những đối tượng bên trong của nó thì nó sẽ yêu cầu ủy quyền truyền vào hai đối tượng chứa trong nó như là tham số. Trách nhiệm của việc xác định thứ tự của hai đối tượng được trao cho phương thức đóng gói bởi ủy quyền.

Để kiểm tra thực hiện cơ chế ủy quyền, chúng ta sẽ tạo ra hai lớp, lớp Cat và lớp Student. Hai lớp này có ít điểm chung với nhau, ngoại trừ cả hai thực thi những phương thức được đóng gói bởi WhichIsFirst. Do vậy cả hai đối tượng này có thể được lưu giữ bên trong của đối tượng Pair.

Trong chương trình thử nghiệm này chúng ta sẽ tạo ra hai đối tượng Student và hai đối tượng Cat và lưu chúng vào mỗi một đối tượng Pair. Sau đó chúng ta sẽ tạo những đối tượng ủy quyền để đóng gói những phương thức của chúng, những phương thức này phải phù hợp với ký pháp và kiểu trả về của ủy quyền. Sau cùng chúng ta sẽ yêu cầu những đối tượng Pair này sắp xếp những đối tượng Student và Cat, ta làm từng bước như sau:

Bắt đầu bằng việc tạo phương thức khởi dựng Pair lấy hai đối tượng và đưa chúng vào trong từng mảng riêng:

```
public class Pair
{
        // đưa vào 2 đối tượng theo thứ tự
        public Pair( object firstObjectr,  object secondObject)
        {
         thePair[0] = firstObject;
```

```
        thePair[1] = secondObject;
        }
        // biến lưu giữ hai đối tượng
        private object[] thePair = new object[2];
}
```

Tiếp theo là chúng ta phủ quyết phương thức ToString() để chứa giá trị mới của hai đối tượng mà Pair nắm giữ:

```
public override string ToString()
{
        // xuất thứ tự đối tượng thứ nhất trước đối tượng thứ hai
        return thePair[0].ToString() +”,” + thePair[1].ToString();
}
```

Bây giờ thì chúng ta đã có hai đối tượng bên trong của Pair và chúng ta có thể xuất giá trị của chúng ra màn hình. Tiếp tục là chúng ta sẽ thực hiện việc sắp xếp và in kết quả sắp xếp. Hiện tại thì không xác định được loại đối tượng mà chúng ta có, do đó chúng ta sẽ ủy quyền quyết định thứ tự sắp xếp cho chính bản thân các đối tượng mà Pair lưu giữ bên trong. Do vậy, chúng ta yêu cầu rằng mỗi đối tượng được lưu giữ bên trong Pair thực hiện việc kiểm tra xem đối tượng nào sắp trước. Phương thức này lấy hai tham số đối tượng và trả về giá trị kiểu liệt kê: theFirstComeFirst nếu đối tượng đầu tiên được đến trước và theSecondComeFirst nếu giá trị thứ hai đến trước.

Những phương thức yêu cầu sẽ được đóng gói bởi ủy quyền WhichIsFirst được định nghĩa bên trong lớp Pair:

```
public delegate comparison
WhichIsFirst( object obj1, object obj2);
```

Giá trị trả về là kiểu comparison đây là kiểu liệt kê:

```
public enum comparison
{
        theFirstComesFirst = 1,
        theSecondComesFirst = 2
}
```

Bất cứ phương thức tĩnh nào lấy hai tham số đối tượng object và trả về kiểu comparison có thể được đóng gói bởi ủy quyền vào lúc thực thi.

Lúc này chúng ta định nghĩa phương thức Sort cho lớp Pair:

```
public void Sort( WhichIsFirst  theDelegateFunc)
{
        if (theDelegateFunc(thePair[0], thePair[1]) ==
            comparison.theSecondComeFirst)
        {
```

```
        object temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
        }
}
```

Phương thức này lấy một tham số: một ủy quyền có kiểu WhichIsFirst với tên là theDelegateFunc. Phương thức Sort giao phó trách nhiệm quyết định thứ tự đến trước sau của hai đối tượng bên trong Pair đến phương thức được đóng gói bởi ủy quyền. Bên trong thân của Sort, phương thức ủy quyền được gọi và trả về một giá trị, giá trị này là một trong hai giá trị liệt kê của comparison.

Nếu giá trị trả về là theSecondComesFirst, đối tượng bên trong của Pair sẽ được hoán đổi vị trí, trường hợp ngược lại thì không làm gì cả.

Hãy tưởng tượng chúng ta đang sắp xếp những Student theo tên. Chúng ta viết một phương thức trả về theFirstComesFirst nếu tên của sinh viên đầu tiên đến trước và the-SecondComesFirst nếu tên của sinh viên thứ hai đến trước. Nếu chúng ta đưa vào là "Amy, Beth" thì phương thức trả về kết quả là theFirstComesFirst. Và ngược lại nếu chúng ta truyền "Beth, Amy" thì kết quả trả về là theSecondComesFirst. Khi chúng ta nhận được kết quả theSecondComesFirst, phương thức Sort sẽ đảo hai đối tượng này trong mảng, và thiết lập là Amy ở vị trí đầu còn Beth ở vị trí thứ hai.

Tiếp theo chúng ta sẽ thêm một phương thức ReverseSort, phương thức này đặt các mục trong mảng theo thứ tự đảo ngược lại:

```
public void ReverseSort( WhichIsFirst theDeleagteFunc)
{
        if ( theDelegateFunc( thePair[0], thePair[1]) ==
            comparison.theFirstComesFirst)
        {
        object temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
        }
}
```

Việc thực hiện cũng tương tự như phương thức Sort. Tuy nhiên, phương thức thực hiện việc hoán đổi nếu phương thức ủy quyền xác định là đối tượng trước tới trước. Do vậy, kết quả thực hiện của phương thức là đối tượng thứ hai sẽ đến trước. Lúc này nếu chúng ta truyền vào là "Amy, Beth", phương thức ủy quyền sẽ trả về theFirstComesFirst, và phương thức ReverseSort sẽ hoán đổi vị trí của hai đối tượng này, thiết lập Beth đến trước. Điều này cho phép chúng ta sử dụng cùng phương thức ủy quyền tương tự như Sort, mà không cần yêu cầu đối tượng hỗ trợ phương thức trả về giá trị được sắp ngược.

Lúc này điều cần thiết là chúng ta tạo ra vài đối tượng để sắp xếp. Ta tạo hai lớp đối tượng đơn giản như sau: lớp đối tượng Student và lớp đối tượng Cat. Gán cho đối tượng Student một tên vào lúc tạo:

```
public class Student
{
        public Student (string name)
        {
        this.name = name;
        }
}
```

Lớp đối tượng Student này yêu cầu hai phương thức, một là phương thức phủ quyết ToString (), và một phương thức khác được đóng gói như là phương thức ủy quyền.

Lớp Student phải phủ quyết phương thức ToString() để cho phương thức ToString() của lớp Pair sử dụng một cách chính xác. Việc thực thi này thì không có gì phức tạp mà chỉ đơn thuần là trả về tên của sinh viên:

```
public override string ToString()
{
        return name;
}
```

Student cũng phải thực thi một phương thức hỗ trợ cho Pair.Sort() có thể ủy quyền xác định thứ tự của hai đối tượng xem đối tượng nào đến trước:

```
public static comparison WhichStudentComesFirst(Object o1, Object o2)
{
        Student s1 = (Student) o1;
        Student s2 = (Student) o2;
        return ( String.Compare( s1.name, s2.name) <0 ?
            comparison.theFirstComesFirst :
            comparison.theSecondComesFirst);
}
```

String.Compare là phương thức của .NET trong lớp String, phương thức này so sánh hai chuỗi và trả về một giá trị nhỏ hơn 0 nếu chuỗi đầu tiên nhỏ hơn chuỗi thứ hai và lớn hơn 0 nếu chuỗi thứ hai nhỏ hơn, và giá trị là 0 nếu hai chuỗi bằng nhau. Phương thức này cũng đã được trình bày trong chương 10 về chuỗi. Theo lý luận trên thì giá trị trả về là theFirstComesFirst chỉ khi chuỗi thứ nhất nhỏ hơn, nếu hai chuỗi bằng nhau hay chuỗi thứ hai lớn hơn, thì phương thức này sẽ trả về cùng giá trị là theSecondComesFirst.

Ghi chú rằng phương thức WhichStudentComesFirst lấy hai tham số kiểu đối tượng và trả về giá trị kiểu liệt kê comparison. Điều này để làm tương ứng và phù hợp với phương thức được ủy quyền Pair.WhichIsFirst.

Lớp thứ hai là Cat, để phục vụ cho mục đích của chúng ta, thì Cat sẽ được sắp xếp theo trọng lượng, nhẹ đến trước nặng. Ta có khai báo lớp Cat như sau:

```
public class Cat
{
        public Cat( int weight)
        {
         this.weight = weight;
        }
        // sắp theo trọng lượng
        public static comparison WhichCatComesFirst(Object o1, Object o2)
        {
        Cat c1 = (Cat) o1;
        Cat c2 = (Cat) o2;
        return c1.weight > c2.weight ?
            theSecondComesFirst :
            theFirstComesFirst;
        }
        public override string ToString()
        {
        return weight.ToString();
        }
        // biến lưu giữ trọng lượng
        private int weight;
    }
```

Cũng tương tự như lớp Student thì lớp Cat cũng phủ quyết phương thức ToString() và thực thi một phương thức tĩnh với cú pháp tương ứng với phương thức ủy quyền. Và chúng ta cũng lưu ý là phương thức ủy quyền của Student và Cat là không cùng tên với nhau. Chúng ta không cần thiết phải làm cùng tên vì chúng ta sẽ gán đến phương thức ủy quyền lúc thực thi. Ví dụ minh họa 11.1 sau trình bày cách một phương thức ủy quyền được gọi.

Ví dụ 11.1: Làm việc với ủy quyền.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
{
    using System;
    // khai báo kiểu liệt kê
    public enum comparison
    {
            theFirstComesFirst =1,
```

```
        theSecondComesFirst = 2
    }
    // lớp Pair đơn giản lưu giữ 2 đối tượng
    public class Pair
    {
        // khai báo ủy quyền
        public delegate comparison WhichIsFirst( object obj1, object obj2);
        // truyền hai đối tượng vào bộ khởi dựng
        public Pair( object firstObject, object secondObject)
        {
         thePair[0] = firstObject;
         thePair[1] = secondObject;
        }
        // phương thức sắp xếp thứ tự của hai đối tượng
        // theo bất cứ tiêu chuẩn nào của đối tượng
        public void Sort( WhichIsFirst theDelegateFunc)
        {
         if (theDelegateFunc(thePair[0], thePair[1]) ==
                comparison.theSecondComesFirst)
         {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
        }
        // phương thức sắp xếp hai đối tượng theo
        // thứ tự nghịch đảo lại tiêu chuẩn sắp xếp
        public void ReverseSort( WhichIsFirst theDelegateFunc)
        {
         if (theDelegateFunc( thePair[0], thePair[1]) ==
                comparison.theFirstComesFirst)
         {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
        }
        // yêu cầu hai đối tượng đưa ra giá trị của nó
```

```
        public override string ToString()
        {
          return thePair[0].ToString() + ", "+ thePair[1].ToString();
        }
        // mảng lưu 2 đối tượng
        private object[] thePair = new object[2];
}
//lớp đối tượng Cat
public class Cat
{
    public Cat(int weight)
    {
        this.weight = weight;
     }
    // sắp theo thứ tự trọng lượng
    public static comparison WhichCatComesFirst(Object o1, Object o2)
    {
        Cat c1 = (Cat) o1;
        Cat c2 = (Cat) o2;
        return c1.weight > c2.weight ?
            comparison.theSecondComesFirst :
            comparison.theFirstComesFirst;
    }
    public override string ToString()
    {
        return weight.ToString();
    }
    // biến lưu trọng lượng
    private int weight;
   }
   // khai báo lớp Student
   public class Student
   {
        public Student( string name)
        {
          this.name = name;
        }
        // sắp theo thứ tự chữ cái
```

```
        public static comparison WhichStudentComesFirst( Object o1, Object o2)
        {
           Student s1 = (Student) o1;
           Student s2 = (Student) o2;
           return (String.Compare( s1.name, s2.name) <0 ?
                comparison.theFirstComesFirst :
                comparison.theSecondComesFirst);
        }
        public override string ToString()
        {
           return name;
        }
        // biến lưu tên
        private string name;
}
public class Test
{
    public static void Main()
    {
            // tạo ra hai đối tượng Student và Cat
            // đưa chúng vào hai đối tượng Pair
            Student  Thao = new Student("Thao");
            Student   Ba = new Student("Ba");
            Cat  Mun = new Cat(5);
            Cat  Ngao = new Cat(2);
            Pair studentPair = new Pair(Thao, Ba);
            Pair catPair = new Pair(Mun, Ngao);
            Console.WriteLine("Sinh vien \t\t\t: {0}", studentPair.ToString());
            Console.WriteLine("Meo \t\t\t: {0}", catPair.ToString());
            // tạo ủy quyền
            Pair.WhichIsFirst  theStudentDelegate = new
                Pair.WhichIsFirst( Student.WhichStudentComesFirst);
            Pair.WhichIsFirst  theCatDelegate = new
                Pair.WhichIsFirst( Cat.WhichCatComesFirst);
            // sắp xếp dùng ủy quyền
            studentPair.Sort( theStudentDelegate);
            Console.WriteLine("Sau khi sap xep studentPair\t\t:{0}",
                    studentPair.ToString());
```

```
                studentPair.ReverseSort(theStudentDelegate);
                Console.WriteLine("Sau khi sap xep nguoc studentPair\t\t:{0}",
                    studentPair.ToString());
                catPair.Sort( theCatDelegate);
                Console.WriteLine("Sau khi sap xep catPair\t\t:{0}",
                            catPair.ToString());
                catPair.ReverseSort(theCatDelegate);
                Console.WriteLine("Sau khi sap xep nguoc  catPair\t\t:{0}",
                    catPair.ToString());
            }
        }
}
```

Kết quả:

```
Sinh vien            : Thao, Ba
Meo                : 5, 2
Sau khi sap xep studentPair      : Ba, Thao
Sau khi sap xep nguoc studentPair   : Thao, Ba
Sau khi sap xep catPair    : 2, 5
Sau khi sap xep nguoc catPair    : 5, 2
```

Trong đoạn chương trình thử nghiệm trên chúng ta tạo ra hai đối tượng Student và hai đối tượng Cat sau đó đưa chúng vào hai đối tượng chứa Pair theo từng loại. Bộ khởi dựng của lớp Student lấy một chuỗi đại diện cho tên của sinh viên và bộ khởi dựng của lớp Cat thì lấy một số int đại diện cho trọng lượng của mèo.

```
Student  Thao = new Student("Thao");
Student   Ba = new Student("Ba");
Cat  Mun = new Cat("5");
Cat  Ngao = new Cat("2");
Pair studentPair = new Pair(Thao, Ba);
Pair catPair = new Pair(Mun, Ngao);
Console.WriteLine("Sinh vien \t\t\t: {0}", studentPair.ToString());
Console.WriteLine("Meo \t\t\t: {0}", catPair.ToString());
```

Sau đó chương trình in nội dung chứa bên trong của hai đối tượng chứa Pair, và chúng ta có thể thấy thứ tự như sau:

```
Sinh vien            : Thao, Ba
Meo                : 5, 2
```

Thứ tự xuất hiện của nó chính là thứ tự đưa vào. Tiếp theo chúng ta khởi tạo hai đối tượng ủy quyền:

```
Pair.WhichIsFirst  theStudentDelegate = new
            Pair.WhichIsFirst( Student.WhichStudentComesFirst);
Pair.WhichIsFirst  theCatDelegate = new
Pair.WhichIsFirst( Student.WhichCatComesFirst);
```

Ủ y quyền đầu tiên theStudentDelegate được tạo ra bằng cách truyền vào một phương thức tĩnh tương ứng của lớp Student. Đối tượng ủy quyền thứ hai, theCatDelegate được một phương thức tĩnh của lớp Cat.

Bây giờ ta đã có các đối tượng ủy quyền, chúng ta truyền ủy quyền đầu tiên cho phương thức Sort của đối tượng Pair, và sau đó là phương thức ReverseSort. Kết quả được xuất ra màn hình:

```
Sau khi sap xep studentPair      : Ba, Thao
Sau khi sap xep nguoc studentPair  : Thao, Ba
Sau khi sap xep catPair          : 2, 5
Sau khi sap xep nguoc catPair       : 5, 2
```

Ủ y quyền tĩnh

Như chúng ta đã thấy trong ví dụ minh hoạ 11.1 trước thì hai thể hiện phương thức ủy quyền được khai báo bên trong lớp gọi (chính xác là trong hàm Main của Test). Điều này có thể không cần thiết ta có thể sử dụng khai báo ủy quyền tĩnh từ hai lớp Student và Cat. Do vậy ta có thể bổ sung lớp Student bằng cách thêm vào:

```
public static readonly Pair.WhichIsFirst OrderStudents =
    new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

Ý nghĩa của lệnh trên là tạo một ủy quyền tĩnh tên là OrderStudents và có thuộc tính chỉ đọc **readonly**. Việc thêm vào thuộc tính **readonly** để ghi chú rằng một khi trường đã được tạo ra thì không được bổ sung sau đó.

Tương tự như vậy chúng ta có thể tạo ủy quyền tĩnh cho Cat như sau:

```
public static readonly Pair.WhichIsFirst OderCats =
    new Pair.WhichIsFirst( Cat.WhichCatComesFirst);
```

Bây giờ thì đã có hai trường tĩnh hiện diện bên trong các lớp Student và Cat, mỗi cái sẽ gắn với phương thức tương ứng bên trong lớp. Sau đó chúng ta có thể thực hiện ủy quyền mà không cần khai báo thể hiện ủy quyền cục bộ. Việc chuyển ủy quyền được thực hiện trong lệnh in đậm như sau:

```
studentPair.Sort( theStudentDelegate);
Console.WriteLine("Sau khi sap xep studentPair\t\t:{0}", studentPair.ToString());
studentPair.ReverseSort(Student.OrderStudents);
Console.WriteLine("Sau khi sap xep nguoc studentPair\t\t:{0}",
```

```
            studentPair.ToString());
    catPair.Sort( theCatDelegate);
    Console.WriteLine("Sau khi sap xep catPair\t\t:{0}", catPair.ToString());
    catPair.ReverseSort(Cat.OrderCats);
     Console.WriteLine("Sau khi sap xep nguoc  catPair\t\t:{0}", catPair.ToString());
```

Kết quả thực hiện tương tự như trong ví dụ 11.1

Sử dụng ủy quyền như thuộc tính

Đối với ủy quyền tĩnh thì chúng bắt buộc phải được tạo thể hiện, do tính chất tĩnh, mà không cần biết là chúng có được sử dụng hay không, như lớp Student và Cat trong ví dụ bên trên. Chúng ta có thể phát triển những lớp này tốt hơn bằng cách thay thế ủy quyền tĩnh từ trường thành thuộc tính.

Với lớp Student ta có thể chuyển khai báo:

```
    public static readonly Pair.WhichIsFirst OrderStudent =
        new Pair.WhichIsFirst( Student.WhichStudentComesFirst);
```

thành khai báo như sau:

```
    public static Pair.WhichIsFirst OrderStudents
    {
            get
            {
            return new Pair.WhichIsFirst( WhichStudentComesFirst);
            }
    }
```

Tương tự như vậy chúng ta thực hiện thay thế với lớp Cat:

```
    public static Pair.WhichIsFirst  OderCats
    {
            get
            {
            return new Pair.WhichIsFirst( WhichCatComesFirst);
            }
    }
```

Khi truyền cho phương thức thì không thay đổi:

```
    studentPair.Sort( Student.OderStudents);
    catPair.Sort( Cat.OrderCats);
```

Khi thuộc tính OrderStudents được truy cập thì ủy quyền được tạo ra:

```
    return new Pair.WhichIsFirst( WhichCatComesFirst);
```

Điều quan trọng ở đây là ủy quyền sẽ không được tạo cho đến khi nào nó được yêu cầu. Việc này cho phép lớp gọi (như lớp Test) quyết định khi nào cần thiết sử dụng một ủy quyền nhưng vẫn cho phép việc tạo ủy quyền là trách nhiệm của lớp Student hay lớp Cat.

Thiết lập thứ tự thi hành với mảng ủy quyền

Ủ y quyền có thể giúp chúng ta tạo ra một hệ thống trong đó người sử dụng có thể quyết định đến thứ tự của các hoạt động khi thực thi. Giả sử chúng ta có một hệ thống xử lý ảnh trong đó các ảnh có thể được thao tác bởi một phương pháp được định nghĩa tốt như là: làm mờ, làm đậm, xoay, lọc ảnh,... Giả sử rằng, thứ tự khi sử dụng các hiệu ứng này được áp dụng cho ảnh là quan trọng. Người sử dụng muốn lựa chọn những hiệu ứng này từ menu, anh ta chọn tất cả các hiệu ứng tùy thích, và sau đó yêu cầu bộ xứ lý ảnh thực hiện lần lượt các hiệu ứng mà anh ta đã xác định.

Chúng ta có thể tạo những ủy quyền cho mỗi hoạt động và sau đó thêm chúng vào một tập hợp được sắp, như là một mảng chẳng hạn, theo một thứ tự mà ta muốn chúng thực thi. Một khi tất cả các ủy quyền được tạo ra và đưa vào tập hợp, chúng ta dễ dàng lặp lần lượt qua các thành phần của mảng, và thực thi lần lượt từng phương thức ủy quyền.

Chúng ta bắt đầu bằng việc xây dựng một lớp Image thể hiện một ảnh sẽ được xử lý bởi lớp ImageProcessor:

```
public class Image
{
        public Image()
        {
          Console.WriteLine("An image created");
        }
}
```

Chúng ta có thể tưởng tượng rằng việc xuất ra chuỗi như vậy tương ứng với việc tạo một ảnh .gif hay .jpeg hay đại loại như vậy.

Sau đó lớp ImageProcessor khai báo một ủy quyền. Dĩ nhiên là chúng ta có thể định nghĩa một ủy quyền riêng trả về bất cứ kiểu dữ liệu nào hay lấy bất cứ tham số nào mà chúng ta muốn. Trong ví dụ này chúng ta định nghĩa một ủy quyền có thể đóng gói bất cứ phương thức không có giá trị trả về và cũng không nhận bất cứ tham số nào hết:

```
public delegate void DoEffect();
```

Tiếp tục lớp ImageProcessor khai báo một sô phương thức, và từng phương thức này phù hợp với ký pháp và kiểu trả về được khai báo bởi ủy quyền:

```
public static void Blur()
{
        Console.WriteLine("Blurring image");
}
```

```
public static void Filter()
{
        Console.WriteLine("Filtering image");
}
public static void Sharpen()
{
        Console.WriteLine("Sharpening image");
}
public static void Rotate()
{
        Console.WriteLine("Rotating image");
}
```

Lớp ImageProcessor cần thiết có một mảng để lưu giữ các ủy quyền mà người sử dụng chọn, một biến lưu giữ số hiệu ứng được chọn và dĩ nhiên là có một biến ảnh để xử lý:

```
DoEffect[]  arrayOfEffects;
Image image;
int numEffectsRegistered = 0;
```

ImageProcessor cũng cần một phương thức để thêm các ủy quyền vào trong mảng:

```
public void AddToEffects( DoEffect theEffect)
{
        if (numEffectsRegistered >=0)
        {
        throw new Exception("Too many members in array");
        }
        arrayOfEffects[numEffectsRegistered ++] = theEffect;
}
```

Ngoài ra còn cần một phương thức thật sự gọi các ủy quyền này:

```
public void ProcessImage()
{
        for (int i = 0; i < numEffectsRegistered; i++)
        {
         arrayOfEffects[i]();
        }
}
```

Cuối cùng, chúng ta khai báo những ủy quyền tĩnh, để các client gọi, và chặn chúng lại để xử lý những phương thức:

```
public DoEffect  BlurEffect = new DoEffect(Blur);
public DoEffect  SharpenEffect = new DoEffect(Sharpen);
```

```
    public DoEffect FilterEffect = new DoEffect(Filter);
    public DoEffect RotateEffect = new DoEffect(Rotate);
```

Việc chọn các thao tác diễn ra trong quá trình tương tác ở thành phần giao diện người sử dụng. Trong ví dụ này chúng ta mô phỏng bằng cách chọn các hiệu ứng, thêm chúng vào trong mảng, và ProcessImage.

Ví dụ minh họa 11.2: Sử dụng mảng ủy quyền.

```
-------------------------------------------------------------------------
namespace Programming_CSharp
{
    using System;
    // khai báo lớp ảnh
    public class Image
    {
            public Image()
            {
            Console.WriteLine("An image created");
            }
    }
    // lớp xử lý ảnh
    public class ImageProcessor
    {
            // khai báo ủy quyền
            public delegate  void DoEffect();
            // tạo các ủy quyền tĩnh
            public DoEffect  BlurEffect = new DoEffect(Blur);
            public DoEffect  SharpenEffect = new DoEffect(Sharpen);
            public DoEffect FilterEffect = new DoEffect(Filter);
            public DoEffect RotateEffect = new DoEffect(Rotate);
            // bộ khởi dựng khởi tạo ảnh và mảng
            public ImageProcessor(Image image)
            {
            this.image = image;
            arrayOfEffects = new DoEffect[10];
            }
            // thêm hiệu ứng vào trong mảng
            public void AddToEffects( DoEffect theEffect)
            {
                if (numEffectsRegistered >=0)
```

```
            {
                throw new Exception("Too many members in array");
            }
            arrayOfEffects[numEffectsRegistered ++] = theEffect;
        }
        // các phương thức xử lý ảnh
        public static void Blur()
        {
            Console.WriteLine("Blurring image");
        }
        public static void Filter()
        {
            Console.WriteLine("Filtering image");
        }
        public static void Sharpen()
        {
            Console.WriteLine("Sharpening image");
        }
        public static void Rotate()
        {
            Console.WriteLine("Rotating image");
        }
        // gọi các ủy quyền để thực hiện hiệu ứng
        public void ProcessImage()
        {
            for (int i = 0; i < numEffectsRegistered; i++)
            {
                arrayOfEffects[i]();
            }
        }
        // biến thành viên
        private DoEffect[] arrayOfEffects;
        private Image image;
        private int numEffectsRegistered = 0;
    }
    // lớp Test để kiểm chứng chương trình
    public class Test
    {
```

```
        public static void Main()
        {
         Image theImage = new Image();
         // do không có GUI để thực hiện chúng ta sẽ chọn lần
         // lượt các hành động và thực hiện
         ImageProcessor  theProc = new ImageProcessor(theImage);
         theProc.AddToEffects(theProc.BlurEffect);
         theProc.AddToEffects(theProc.FilterEffect);
         theProc.AddToEffects(theProc.RotateEffect);
         theProc.AddToEffects(theProc.SharpenEffect);
         theProc.ProcessImage();
        }
   }
}
```

Kết quả:

```
An image created
Blurring image
Filtering image
Rotate image
Sharpening image
```

Trong ví dụ trên, đối tượng ImageProcessor được tạo ra và những hiệu ứng được thêm vào. Nếu người dùng chọn làm mờ trước khi lọc ảnh, thì đơn giản là được đưa vào mảng ủy quyền theo thứ tự tương ứng. Tương tự như vậy, bất cứ hành động lựa chọn nào của người dùng mong muốn, ta đưa thêm nhiều ủy quyền vào trong tập hợp.

Chúng ta có thể tưởng tượng việc hiển thị thứ tự hành động này trong một danh sách listbox và cho phép người sử dụng sắp xếp lại phương thức, di chuyển chúng lên xuống trong danh sách. Khi các hành động này được sắp xếp lại thì chúng ta chỉ cần thay đổi thứ tự trong tập hợp. Ngoài ra ta cũng có thể đưa các hoạt động này vào trong cơ sở dữ liệu rồi sau đó đọc chúng lúc thực hiện.

Ủ y quyền dễ dàng cung cấp động cho ta các phương thức được gọi theo một thứ tự xác định

Multicasting

Cơ chế multicasting cho phép gọi hai phương thức thực thi thông qua một ủy quyền đơn. Điều này trở nên quan trọng khi xử lý các sự kiện, sẽ được thảo luận trong phần cuối của chương.

Mục đích chính là có một ủy quyền có thể gọi thực hiện nhiều hơn một phương thức. Điều này hoàn toàn khác với việc có một tập hợp các ủy quyền, vì mỗi trong số chúng chỉ gọi được duy nhất một phương thức. Trong ví dụ trước, tập hợp được sử dụng để lưu giữ các ủy quyền khác nhau. Tập hợp này cũng có thể thêm một ủy quyền nhiều hơn một lần, và sử dụng tập hợp để sắp xếp lại các ủy quyền và điều khiển thứ tự hành động được gọi.

Với Multicasting chúng ta có thể tạo một ủy quyền đơn và cho phép gọi nhiều phương thức được đóng. Ví dụ, khi một nút lệnh được nhấn chúng ta có thể muốn thực hiện nhiều hơn một hành động. Để làm được điều này chúng ta có thể đưa cho button một tập hợp các ủy quyền, nhưng để sáng rõ hơn và dễ dàng hơn là tạo một ủy quyền Multicast.

Bất cứ ủy quyền nào trả về giá trị void là ủy quyền multicast, mặc dù vậy ta có thể đối xử với nó như là ủy quyền bình thường cũng không sao. Hai ủy quyền Multicast có thể được kết hợp với nhau bằng phép toán cộng (+). Kết quả là một ủy quyền Multicast mới và gọi đến tất cả các phương thức thực thi nguyên thủy của cả hai bên. Ví dụ, giả sử Writer và Logger là ủy quyền trả về giá trị void, dòng lệnh theo sau sẽ kết hợp chúng lại với nhau và tạo ra một ủy quyền Multicast mới:

```
myMulticastDelegate = Writer + Logger;
```

Chúng ta cũng có thể thêm những ủy quyền vào trong ủy quyền Multicast bằng toán tử cộng bằng (+=). Phép toán này sẽ thêm ủy quyền ở phía bên phải của toán tử vào ủy quyền Multicast ở bên trái. Ví dụ minh họa như sau, giả sử có Transmitter và myMulticastDelegate là những ủy quyền, lệnh tiếp theo sau đây sẽ thực hiện việc thêm ủy quyền Transmitter vào trong myMulticastDelegate:

```
myMulticastDelegate += Transmitter;
```

Để hiểu rõ ủy quyền Multicast được tạo ra và sử dụng, chúng ta sẽ từng bước tìm hiểu thông qua ví dụ 11.3 bên dưới, trong ví dụ minh họa này chúng ta sẽ tạo ra một lớp có tên gọi là MyClassWithDelegate lớp này định nghĩa một delegate, delegate này lấy một tham số là chuỗi và không có giá trị trả về:

```
void delegate void StringDelegate( string  s);
```

Sau đó chúng ta định một lớp gọi là MyImplementingClass lớp này có ba phương thức, tất cả các phương thức này đều trả về giá trị void và nhận một chuỗi làm tham số: WriteString, LogString, và Transmitting. Phương thức đầu tiên viết một chuỗi xuất ra màn hình tiêu chuẩn, chuỗi thứ hai mô phỏng viết vào một log file, và phương thức thứ ba mô phỏng việc chuyển một chuỗi qua Internet. Chúng ta tạo thể hiện delegate để gọi những phương thức tương ứng:

```
Writer("String passed to Writer\n");
Logger("String passed to Logger\n");
Transmitter("String passed to Transmitter\n");
```

Để xem cách kết hợp các delegate, chúng ta tạo một thể hiện delegate khác:

```
MyClassWithDelegate.StringDelegate  myMulticastDelegate;
```

và gán cho delegate này kết quả của phép cộng hai delegate cho trước:

```
myMulticastDelegate = Writer + Logger;
```

Tiếp theo chúng ta thêm vào delegate này một delegate nữa bằng cách sử dụng toán tử (+=):

```
myMulticastDelegate += Transmitter;
```

Cuối cùng, chúng ta thực hiện việc xóa deleagate bằng sử dụng toán tử (-=):

```
DelegateCollector -= Logger;
```

Sau đây là toàn bộ ví dụ minh họa.

Ví dụ 11.3: Kết hợp các delegate.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
{
    using System;
    public class MyClassWithDelegate
    {
        // khai báo delegate
        public delegate void StringDelegate(string s);
    }
    public class MyImplementingClass
    {
        public static void WriteString( string s)
        {
            Console.WriteLine("Writing string {0}", s);
        }
        public static void LogString( string s)
        {
            Console.WriteLine("Logging string {0}", s);
        }
        public static void TransmitString( string s)
        {
            Console.WriteLine("Transmitting string {0}", s);
        }
    }
    public class Test
    {
        public static void Main()
        {
            // định nghĩa 3 StringDelegate
            MyClassWithDelegate.StringDelegate  Writer, Logger, Transmitter;
```

```
        // định nghĩa một StringDelegate khác thực hiện Multicasting
        MyClassWithDelegate.StringDelegate  myMulticastDelegate;
        // tạo thể hiện của 3 delegate đầu tiên và truyền vào phương thức thực thi
        Writer = new MyClassWithDelegate.StringDelegate(
                MyImplementingClass.WriteString);
        Logger = new MyClassWithDelegate.StringDelegate(
                MyImplementingClass.LogString);
        Transmitter = new MyClassWithDelegate.StringDelegate(
                MyImplementingClass.TransmitString);
        // gọi phương thức delegate Writer
        Writer("String passed to Writer\n");
        // gọi phương thức delegate Logger
        Logger("String passed to Logger\n");
        //gọi phương thức delegate Transmitter
        Transmitter("String passed to Transmitter\n");
        // thông báo người dùng rằng đã kết hợp hai delegate vào
        // trong một multicast delegate
        Console.WriteLine("myMulticastDelegate = Writer + Logger");
        // kết hợp hai delegate
        myMulticastDelegate = Writer + Logger;
        // gọi phương thức delegate, hai phương thức sẽ được thực hiện
        myMulticastDelegate("First string passed to Collector");
        // bảo với người sử dụng rằng đã thêm delegate thứ 3 vào
        // trong Multicast delegate
        Console.WriteLine("\nmyMulticastDeleagte += Transmitter");
        // thêm delegate thứ ba vào
        myMulticastDelegate += Transmitter;
        // gọi thực thi Multicast delegate, cùng một lúc ba
        // phương thức sẽ cùng được gọi thực hiện
        myMulticastDelegate("Second string passed to Collector");
        // bảo với người sử dụng rằng xóa delegate Logger
        Console.WriteLine("\nmyMulticastDelegate -= Logger");
        // xóa delegate Logger
        myMulticastDelegate -= Logger;
        // gọi lại delegate, lúc này chỉ còn thực hiện hai phương thức
        myMulticastDelegate("Third string passed to Collector");
    }// end Main
}// end class
```

```
}// end namespace
```

Kết quả:

```
Writing string String passed to Writer
Logging string String passed to Logger
Transmitting string String passed to Transmitter
myMulticastDelegate = Writer + Logger
Writing string First string passed to Collector
Logging string First string passed to Collector
myMulticastDelegate += Transmitter
Writing string Second string passed to Collector
Logging string Second string passed to Collector
Transmitting string Second string passed to Collector
myMulticastDelegate -= Logger
Writing string Third string passed to Collector
Transmitting string Third string passed to Collector
```

Trong ví dụ trên, những thể hiện delegate được định nghĩa và ba delegate đầu tiên Writer, Logger, và Transmitter được gọi ra. Delegate thứ tư myMulticastDelegate được gán bằng cách kết hợp hai delegate đầu, và khi nó được gọi, thì dẫn đến là cả hai delegate cũng được gọi. Khi delegate thứ ba được thêm vào, và kết quả là khi myMulticastDelegate được gọi thì tất cả ba phương thức delegate cũng được thực hiện. Cuối cùng, khi Logger được xóa khỏi delegate, và khi myMulticastDelegate được gọi thì chỉ có hai phương thức thực thi.

Multicast delegate được thể hiện tốt nhất trong việc ứng dụng xử lý các sự kiện. Khi một sự kiện ví dụ như một nút lệnh được nhấn, thì một multicast delegate tương ứng sẽ gọi đến một loạt các phương thức xử lý sự kiện để đáp ứng lại với các sự kiện này.

Sự kiện

Trong môi trường giao diện đồ họa (Graphical User Interfaces: GUIs), Windows hay trong trình duyệt web đòi hỏi các chương trình phải đáp ứng các sự kiện. Một sự kiện có thể là một nút lệnh được nhấn, một mục trong menu được chọn, hành động sao chép tập tin hoàn thành,...Hay nói ngắn gọn là một hành động nào đó xảy ra, và ta phải đáp ứng lại sự kiện đó. Chúng ta không thể đoán trước được khi nào thì các sự kiện sẽ xuất hiện. Hệ thống sẽ chờ cho đến khi nhận được sự kiện, và sẽ chuyển vào cho trình xử lý sự kiện thực hiện.

Trong môi trường giao diện đồ họa, bất cứ thành phần nào cũng có thể đưa ra sự kiện. Ví dụ, khi chúng ta kích vào một nút lệnh, nó có thể đưa ra sự kiện Click. Khi chúng ta thêm một mục vào danh sách, nó sẽ đưa ra sự kiện ListChanged.

Cơ chế publishing và subscribing

Trong ngôn ngữ C#, bất cứ đối tượng nào cũng có thể publish một tập hợp các sự kiện để cho các lớp khác có thể đăng ký. Khi một lớp publish đưa ra một sự kiện, thì tất cả các lớp đã đăng ký sẽ được nhận sự cảnh báo.

✍*Ghi chú*: Tác giả Gamma (Addison Wesley, 1995) mô tả cơ chế này như sau: "*Định nghĩa một đến nhiều sự phụ thuộc giữa những đối tượng do đó khi một đối tượng thay đổi trạng thái, tất cả các đối tượng khác phụ thuộc vào nó sẽ được cảnh báo và cập nhật một cách tự động*".

Với cơ chế này, đối tượng của chúng ta có thể nói rằng "*Ở đây có những thứ mà tôi có thể thông báo cho bạn*" và những lớp khác có thể đăng ký đáp rằng "*Vâng, hãy báo cho tôi biết khi chuyện đó xảy ra*". Ví dụ, một nút lệnh có thể cảnh báo cho bất cứ thành phần nào khi nó được nhấn. Nút lệnh này được gọi là publisher bởi vì nó phân phát sự kiện Click và những lớp khác là các lớp subscriber vì chúng đăng ký nhận sự kiện Click này.

Sự kiện và delegate

Những sự kiện trong C# được thực thi với những delegate. Lớp publisher định nghĩa một delegate và những lớp subscriber phải thực thi. Khi một sự kiện xuất hiện thì phương thức của lớp subscriber được gọi thông qua delegate.

Một phương thức được dùng để xử lý các sự kiện thì được là trình xử lý sự kiện (event handler). Chúng ta có thể khai báo trình xử lý sự kiện này như chúng ta đã làm với bất cứ delegate khác.

Theo quy ước, những trình xử lý sự kiện trong .NET Framework trả về giá trị void và lấy hai tham số. Tham số đầu tiên là nguồn dẫn đến sự kiện, đây chính là đối tượng publisher. Và tham số thứ hai là đối tượng dẫn xuất từ lớp EventArgs. Yêu cầu chúng ta phải thực hiện trình xử lý sự kiện theo mẫu như trên.

EventArgs là lớp cơ sở cho tất cả các dữ liệu về sự kiện, lớp EventArgs thừa kế tất cả các phương thức của nó từ Object, và thêm vào một trường public static empty thể hiện một sự kiện không có trạng thái (cho phép sử dụng hiệu quả những sự kiện không trạng thái). Lớp dẫn xuất từ EventArgs chứa những thông tin về sự kiện.

Sự kiện là thuộc tính của lớp phát ra sự kiện. Từ khóa **event** điều khiển cách thuộc tính sự kiện được truy cập bởi các lớp subscriber. Từ khóa event được thiết kế để duy trì cho cách thể hiện publish/ subscribe.

Giả sử chúng ta muốn tạo một lớp Clock dùng những sự kiện để cảnh báo những lớp subscriber bất cứ khi nào đồng hồ hệ thống thay đổi giá trị trong một giây. Gọi sự kiện này là OnSecondChange. Chúng ta khai báo sự kiện và kiểu delegate xử lý sự kiện của nó như sau:

```
[attributes] [modifiers] event type
    member- name
```

Ví dụ khai báo như sau:

```
public event SecondChangeHandler   OnSecondChange;
```

Trong ví dụ này ta không dùng thuộc tính, modifier ở đây là **abstract**, **new**, **override**, **static**, **virtual**, hay là một trong bốn access modifier, trong trường hợp này **public**. Modifier được theo sau bởi từ khóa **event**.

Trường type trong trường hợp ví dụ này là delegate mà chúng ta muốn liên hệ với sự kiện, ở đây là SecondChangeHandler.

Tên thành viên là tên của sự kiện, trong trường hợp này là OnSecondChange. Thông thường, tên sự kiện bắt đầu với từ On.

Tóm lại, trong sự khai báo này OnSecondChange là sự kiện được thực thi bởi delegate có kiểu là SecondChangeHandler.

Ta có khai báo cho delegate này như sau:

```
public delegate void SecondChangeHandler( object clock,
            TimeInfoEventArgs timeInformation);
```

Như đã nói trước đây, theo quy ước một trình xử lý sự kiện phải trả về giá trị void và phải lấy hai tham số: nguồn phát ra sự kiện (trong trường hợp này là clock) và một đối tượng dẫn xuất từ EventArgs, là TimeInfoEventArgs. Lớp TimeInfoEventArgs được định nghĩa như sau:

```
public class TimeInfoEventArgs : EventArgs
{
        public TimeInfoEventArgs(int hour, int minute, int second)
        {
                this.hour = hour;
                this.minute = minute;
                this.second = second;
        }
        public readonly int hour;
        public readonly int minute;
        public readonly int second;
    }
```

Đối tượng TimeInfoEventArgs sẽ có thông tin về giờ phút giây hiện thời. Nó định nghĩa một bộ khởi tạo, ba phương thức, một biến nguyên **readonly**.

Ngoài việc thêm vào một sự kiện và delegate, lớp đối tượng Clock có ba biến thành viên là : hour, minute, và second. Cuối cùng là một phương thức Run():

```
public void Run()
{
        for(;;)
        {
           // ngừng 10 giây
           Thread.Sleep( 10 );
```

```
            // lấy thời gian hiện hành
            System.DateTime  dt = System.DateTime.Now;
            // nếu giây thay đổi cảnh báo cho subscriber
            if ( dt.Second != second)
            {
                // tạo TimeInfoEventArgs để truyền
                // cho subscriber
                TimeInfoEventArgs  timeInformation =
                    new TimeInfoEventArgs( dt.Hour, dt.Minute, dt.Second);
                // nếu có bất cứ lớp nào đăng ký thì cảnh báo
                if ( OnSecondChange != null)
                {
                   OnSecondChange( this, timeInformation);
                }
            }
            // cập nhật trạng thái
            this.second = dt.Second;
            this.minute = dt.Minute;
            this.hour = dt.Hour;
        }
    }
```

Phương thức Run tạo vòng lặp vô hạn để kiểm tra định kỳ thời gian hệ thống. Nếu thời gian thay đổi từ đối tượng Clock hiện hành, thì nó sẽ cảnh báo cho tất cả các subscriber và sau đó cập nhật lại những trạng thái của nó.

Bước đầu tiên là ngừng 10 giây:

```
Thread.Sleep(10);
```

Ở đây chúng ta sử dụng phương thức tĩnh của lớp Thread từ System.Threading của .NET. Sử dụng phương thức Sleep() để kéo dài khoảng cách giữa hai lần thực hiện vòng lặp.

Sau khi ngừng 10 mili giây, phương thức sẽ kiểm tra thời gian hiện hành:

```
System.DateTime  dt = System.DateTime.Now;
```

Cứ khoảng 100 lần kiểm tra, thì một giây sẽ được gia tăng. Phương thức ghi nhận sự thay đổi và cảnh báo đến những subscriber của nó. Để làm được điều này, đầu tiên phải tạo ra một đối tượng TimeInfoEventArgs:

```
if ( dt.Second != second)
{
    // tạo TimeInfoEventArgs để truyền cho các subscriber
    TimeInfoEventArgs  timeInformation =
        new TimeInfoEventArgs( dt.Hour, dt.Minute, dt.Second);
```

```
    }
```

Và để cảnh báo cho những subscriber bằng cách kích hoạt sự kiện OnSecondChange:

```
// cảnh báo cho các subscriber
if ( OnSecondChange != null)
{
        OnSecondChange( this, timeInformation);
}
```

Nếu một sự kiện không có bất cứ lớp subscriber nào đăng ký thì nó ước lượng giá trị là null. Phần kiểm tra bên trên xác định giá trị của sự kiện có phải là null hay không, để đảm bảo rằng có tồn tại lớp đăng ký nhận sự kiện trước khi gọi sự kiện OnSecondChange.

Chúng ta lưu ý rằng OnSecondChange lấy hai tham số: nguồn phát ra sự kiện và đối tượng dẫn xuất từ lớp EventArgs. Ở đây chúng ta có thể thấy rằng tham chiếu **this** của lớp clock được truyền bởi vì clock là nguồn phát ra sự kiện. Tham số thứ hai là đối tượng TimeInfo-EventArgs được tạo ra ở dòng lệnh bên trên.

Một sự kiện được phát ra thì sẽ gọi bất cứ phương thức nào được đăng ký với lớp Clock thông qua delegate, chúng ta sẽ kiểm tra điều này sau.

Một khi mà sự kiện được phát ra, chúng ta sẽ cập nhật lại trạng thái của lớp Class:

```
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
```

Sau cùng là chúng ta xây dựng những lớp có thể đăng ký vào các sự kiện này. Chúng ta sẽ tạo hai lớp. Lớp đầu tiên là lớp DisplayClock. Chức năng chính của lớp này không phải là lưu giữ thời gian mà chỉ để hiển thị thời gian hiện hành ra màn hình console. Để đơn giản chúng ta chỉ tạo hai phương thức cho lớp này. Phương thức thứ nhất có tên là Subscribe, phương thức chịu trách nhiệm đăng ký một sự kiện OnSecondChange của lớp Clock. Phương thức thứ hai được tạo ra là trình xử lý sự kiện TimeHasChanged:

```
public class DisplayClock
{
        public void Subscribe(Clock theClock)
        {
            theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
         }
       public void TimeHasChanged( object theClock, TimeInfoEventArgs  ti)
       {
            Console.WriteLine("Current Time: {0]:{1}:{2}",
                 ti.hour.ToString(), ti.minute.ToString(),
                     ti.Second.ToString());
```

```
        }
    }
```

Khi phương thức đầu tiên Subscribe được gọi, nó sẽ tạo ra một delegate SecondChange-Handler mới, và truyền vào phương thức xử lý sự kiện TimeHasChanged của nó. Sau đó nó sẽ đăng ký delegate với sự kiện OnSecondChange của Clock.

Lớp thứ hai mà chúng ta tạo cũng sẽ đáp ứng sự kiện này, tên là LogCurrentTime. Thông thường lớp này ghi lại sự kiện vào trong tập tin, nhưng với mục đích minh họa của chúng ta, nó sẽ ghi ra màn hình console:

```
public class LogCurrentTime
{
        public void  Subscribe(Clock theClock)
        {
            theClock.OnSecondChange +=
                new Clock.SecondChangeHandler(WriteLogEntry);
        }
        // thông thường phương thức này viết ra file
        // nhưng trong minh họa này chúng ta chỉ xuất
        // ra màn hình console mà thôi
        public void WriteLogEntry( object theClock, TimeInfoEventArgs  ti)
        {
            Console.WriteLine("Logging to file: {0}:{1}:{2}",
                ti.hour.ToString(),
                ti.minute.ToString(),
                ti.second.ToString());
        }
    }
```

Ghi chú rằng những sự kiện được thêm vào bằng cách sử dụng toán tử +=. Điều này cho phép những sự kiện mới được thêm vào sự kiện OnSecondChange của đối tượng Clock mà không có phá hủy bất cứ sự kiện nào đã được đăng ký. Khi LogCurrentTime đăng ký một sự kiện OnSecondChange, chúng ta không muốn việc đăng ký này làm mất đi sự đăng ký của lớp DisplayClock trước đó.

Tất cả phần còn lại cần thực hiện là tạo ra một lớp Clock, tạo mộ đối tượng DisplayClock và bảo nó đăng ký sự kiện. Sau đó chúng ta tạo ra một đối tượng LogCurrentTime và cũng đăng ký sự kiện tương tự. Cuối cùng thì thực thi phương thức Run của Clock. Tất cả phần trên được trình bày trong ví ụ 11.4.

Ví dụ 11.4: làm việc với những sự kiện.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
```

```
{
    using System;
    using System.Threading;
    // lớp lưu giữ thông tin về sự kiện, trong trường hợp
    // này nó chỉ lưu giữ những thông tin có giá trị lớp clock
    public class TimeInfoEventArgs : EventArgs
    {
            public TimeInfoEventArgs(int hour, int minute, int second)
            {
            this.hour  = hour;
            this.minute = minute;
            this.second = second;
            }
            public readonly int hour;
            public readonly int minute;
            public readonly int second;
    }
    // khai báo lớp Clock lớp này sẽ phát ra các sự kiện
    public class  Clock
    {
            // khai báo delegate mà các subscriber phải thực thi
            public delegate  void  SecondChangeHandler(object clock,
                TimeInfoEventArgs  timeInformation);
            // sự kiện mà chúng ta đưa ra
            public event  SecondChangeHandler  OnSecondChange;
            // thiết lập đồng hồ thực hiện, sẽ phát ra mỗi sự kiện trong mỗi giây
        public void Run()
        {
             for(;;)
             {
                // ngừng 10 giây
                Thread.Sleep( 10 );
                // lấy thời gian hiện hành
                System.DateTime  dt = System.DateTime.Now;
                // nếu giây thay đổi cảnh báo cho subscriber
                if ( dt.Second != second)
                {
                    // tạo TimeInfoEventArgs để truyền
```

```
                // cho subscriber
                TimeInfoEventArgs  timeInformation =
                    new TimeInfoEventArgs( dt.Hour, dt.Minute, dt.Second);
                // nếu có bất cứ lớp nào đăng ký thì cảnh báo
                if ( OnSecondChange != null)
                {
                    OnSecondChange( this, timeInformation);
                }
            }
            // cập nhật trạng thái
            this.second = dt.Second;
            this.minute = dt.Minute;
            this.hour = dt.Hour;
        }
    }
    private int hour;
    private int minute;
    private int second;
}
// lớp DisplayClock đăng ký sự kiện của clock.
// thực thi xử lý sự kiện bằng cách hiện thời gian hiện hành
public class DisplayClock
{
        public void Subscribe(Clock theClock)
        {
            theClock.OnSecondChange +=
                new Clock.SecondChangeHandler(TimeHasChanged);
        }
        public void TimeHasChanged( object theClock, TimeInfoEventArgs  ti)
        {
                Console.WriteLine("Current Time: {0}:{1}:{2}",
                    ti.hour.ToString(),
                    ti.minute.ToString(),
                    ti.second.ToString());
        }
}
// lớp đăng ký sự kiện thứ hai
public class LogCurrentTime
```

```
    {
        public void  Subscribe(Clock theClock)
        {
            theClock.OnSecondChange +=
                new Clock.SecondChangeHandler(WriteLogEntry);
        }
        // thông thường phương thức này viết ra file
        // nhưng trong minh họa này chúng ta chỉ xuất
       // ra màn hình console mà thôi
       public void WriteLogEntry( object theClock, TimeInfoEventArgs  ti)
      {
            Console.WriteLine("Logging to file: {0}:{1}:{2}",
                ti.hour.ToString(),
                ti.minute.ToString(),
                ti.second.ToString());
      }
    }
    // lớp Test minh họa sử dụng sự kiện
    public class Test
    {
            public static  void Main()
            {
            // tạo ra đối tượng clock
            Clock  theClock  = new Clock();
            // tạo đối tượng DisplayClock đăng ký
            // sự kiện và xử lý sự kiện
            DisplayClock  dc = new DisplayClock();
            dc.Subscribe(theClock);
            // tạo đối tượng LogCurrent và yêu cầu đăng
            // ký và xử lý sự kiện
            LogCurrentTime  lct = new LogCurrentTime();
            lct.Subscribe(theClock);
            // bắt đầu thực hiện vòng lặp và phát sinh sự kiện
            // trong mỗi giây đồng hồ
            theClock.Run();
            }
    }
}
```

Kết quả thực hiện có thể như sau:

```
Current Time: 11:54:20
Logging to file: 11:54:20
Current Time: 11:54:21
Logging to file: 11:54:21
Current Time: 11:54:22
Logging to file: 11:54:22
```

Điều quan trọng chính của ví dụ minh họa trên là việc tạo ra hai lớp đối tượng DisplayClock và lớp LogCurrentTime. Cả hai lớp này đều đăng ký một sự kiện Clock.OnSecondChange của lớp thứ ba là lớp Clock

Lợi ích của cơ chế publish/subscribe là bất cứ lớp nào cũng có thể được cảnh báo khi một sự kiện xuất hiện. Những lớp subscriber không cần biết cách mà Clock làm việc, và Clock cũng không cần biết cách mà các lớp subscriber đáp ứng với sự kiện mà nó đưa ra.

Publisher và subscriber được phân tách bởi delegate, đây là một sự mong đợi cao, nó làm cho mã lệnh linh hoạt và mạnh mẽ hơn. Lớp Clock có thể thay đổi cách dò thời gian mà không làm ảnh hưởng đến bất cứ lớp subscriber nào. Các lớp subscriber có thể thay đổi cách mà chúng đáp ứng với sự thay đổi của thời gian mà không tác động với Clock. Cả hai lớp này hoạt động độc lập với nhau, và làm cho đoạn chương trình dễ duy trì hơn.

Câu hỏi và trả lời

Câu hỏi 1: Tóm tắt những nét cơ bản về uỷ quyền?

Trả lời 1: Ủy quyền là một kiểu dữ liệu tham chiếu đươc dùng để đóng gói phương thức với các tham số và kiểu trả về xác định. Ủy quyền cũng tương tự như con trỏ hàm trong ngôn ngữ C++. Tuy nhiên, trong ngôn ngữ C# ủy quyền là kiểu dữ liệu hướng đối tượng, an toàn và bảo mật.

Câu hỏi 2: Con trỏ hàm là gì?

Trả lời 2: Trong ngôn ngữ như C hay C++, có một chức năng gọi là con trỏ hàm. Một con trỏ hàm được sử dụng để thiết lập cùng một nhiệm vụ như một ủy quyền. Tuy nhiên, con trỏ hàm trong C/C++ đơn giản không phải là một đối tượng. Còn ủy quyền trong C# là kiểu dữ liệu an toàn, được dùng để tham chiếu đến những phương thức, ủy quyền còn được sử dụng bởi những sự kiện.

Câu hỏi thêm

Câu hỏi 1: Có thể sử dụng ủy quyền như một thuộc tính hay không? Nếu có thể thì sử dụng như thế nào? Cho biết ý nghĩa?

Câu hỏi 2: Nếu có một số hoạt động cần được thực hiện theo một thứ tự nhất định thì ta phải làm thế nào để khi cần thực hiện thì gọi lần lượt thực hiện các hoạt động đó?

Câu hỏi 3: Công dụng của việc khai báo ủy quyền tĩnh? Khi nào thì nên khai báo ủy quyền tĩnh khi nào thì không nên?
Câu hỏi 4: Một ủy quyền có thể gọi được nhiều hơn một phương thức hay không? Chức năng nào trong C# hỗ trợ ủy quyền này?
Câu hỏi 5: Có phải tất cả các ủy quyền đều là ủy quyền Multicast hay không? Điều kiện để trở thành ủy quyền Multicast?
Câu hỏi 6: Các toán tử nào có thể dùng để thực hiện việc Multicast các ủy quyền?
Câu hỏi 7: Sự kiện là gì? Trong hệ thống ứng dụng nào thì sự kiện được sử dụng nhiều?
Câu hỏi 8: Những sự kiện trong C# được thực hiện thông qua cái gì?
Câu hỏi 9: Hãy tóm lược quá trình tạo một sự kiện và giải quyết sự kiện thông qua cơ chế ủy quyền trong C#?

Bài tập

Bài tập 1: Viết chương trình minh họa sử dụng ủy quyền để thực hiện việc sắp xếp các số nguyên trong một mảng?
Bài tập 2: Viết chương trình minh họa sử dụng ủy quyền để thực hiện việc chuyển các ký tự thường thành ký tự hoa trong một chuỗi?
Bài tập 3: Viết chương trình kết hợp giữa delegate và sự kiện để minh họa một đồng hồ điện tử thể hiện giờ hiện hành trên màn hình console.

Chương 12

CÁC LỚP CƠ SỞ .NET

- **Lớp đối tượng trong .NET Framework**
- **Lớp Timer**
- **Lớp về thư mục & hệ thống**
- **Lớp Math**
- **Lớp thao tác tập tin**
- **Làm việc với tập tin dữ liệu**
- **Câu hỏi & bài tập**

Cho đến lúc này thì chúng ta đã tìm hiểu khá nhiều các lớp đối tượng mà ngôn ngữ C# cung cấp cho chúng ta. Và hiện tại chúng ta đã có thể viết được các chương trình C# thuần túy dùng console làm giao diện kết xuất. Đối với việc tìm hiểu bất cứ ngôn ngữ lập trình nào thì việc viết các chương trình mà giao diện càng đơn giản thì càng tốt. Trong phần thứ hai (từ chương 14) của giáo trình chúng ta sẽ tìm hiểu xây dựng các ứng dụng Windows thông qua Visual C#.

Trong chương này chúng ta sẽ tìm hiểu các lớp cơ sở mà .NET cung cấp, các lớp này đơn giản giúp chúng ta thực hiện tốt các thao tác nhập xuất, các thao tác truy cập hệ thống, thực thi các phép toán học,...

Lớp đối tượng trong .NET Framework

NET Framework chứa số lượng nhiều những kiểu dữ lớp, những kiểu liệt kê, những cấu trúc, những giao diện và nhiều kiểu dữ liệu khác nữa. Thật vậy, có hàng ngàn số lượng các kiểu như trên. Những lớp này điều cho phép chúng ta sử dụng trong chương trình C#.

Chúng ta sẽ tìm hiểu một vài kiểu dữ liệu thường sử dụng trong chương này. Các lớp được trình bày thông qua các ví dụ minh họa đơn giản. Từ những ví dụ minh họa cách sử dụng các lớp cơ sở này chúng ta có thể mở rộng để tạo ra các chương trình phức tạp hơn.

Common Language Specification (CLR)

Những lớp bên trong Framework được viết với ngôn ngữ được xác nhận là chung nhất (CLR). CLR đã được đề cập vào phần đầu của sách khi chúng ta thảo luận về MS.NET trong chương 1.

CLS là một tập hợp các luật hay các quy tắc mà tất cả các ngôn ngữ thực hiện bên trong .NET platform phải tuân thủ theo. Tập hợp luật này cũng bao gồm kiểu dữ liệu hệ thống chung, các kiểu dữ liệu cơ bản mà chúng ta được tìm hiểu trong chương 3 - Nền tảng ngôn ngữ C#. Bằng cách đưa vào các tập luật này, môi trường thực thi chung sẽ có thể thực thi một chương trình mà không quan tâm đến cú pháp của ngôn ngữ được sử dụng.

Lợi ích theo sau của CLS là mã nguồn được viết trong một ngôn ngữ có thể được gọi sử dụng bởi một ngôn ngữ khác Bởi vì thông thường bên trong Framework với CLS, chúng có thể sử dụng không chỉ ngôn ngữ C# mà còn bất cứ ngôn ngữ tương thích với CLS như là Visual Basic.NET và JScript.NET.

Kiểu dữ liệu trong namespace

Mã nguồn bên trong Framework được tổ chức bên trong namespace. Có hàng trăm namespace bên trong Framework được sử dụng để tổ chức hàng ngàn lớp đối tượng và các kiểu dữ liệu khác.

Một vài namespace thì được lưu trữ bên trong namespace khác. Ví dụ chúng ta đã sử dụng kiểu dữ liệu DateTime được chứa trong namespace System. Kiểu Random cũng được chứa trong namespace System. Nhiều kiểu dữ liệu phục vụ cho thao tác nhập xuất cũng được lưu trữ trong một namespace chức trong namespace System là namespace System.IO. Nhiều kiểu dữ liệu thường dùng để làm việc với dữ liệu XML thì được đặt bên trong namespace System.XML. Chúng ta có thể tìm hiểu các namespace này trong các tài liệu trực tuyến của Microsoft như MSDN Online chẳng hạn.

Tiêu chuẩn ECMA

Không phải tất cả kiểu dữ liệu bên trong namespace thì cần thiết phải tương thích với tất cả những ngôn ngữ khác. Hơn thế nữa, những công cụ phát triển được tạo bởi những công ty khác cho ngôn ngữ C# có thể không bao hàm phải tương thích với mã nguồn thông thường.

Khi ngôn ngữ C# được hình thành. Microsoft xác nhận đưa ra một số lượng lớn các kiểu dữ liệu cho cùng một bảng tiêu chuẩn cho C# để chuẩn hóa. Bằng cách xác nhận những kiểu dữ liệu theo một tiêu chuẩn, điều này xem như việc mở cánh cửa cho những nhà phát triển khác tạo ra các công cụ và trình biên dịch C# cùng sử dụng những namespace và kiểu dữ liệu. Khi đó những mã nguồn bên trong những công cụ của Microsoft tương thích với bất cứ công cụ của các công ty khác.

Những lớp đối tượng được chuẩn hóa thì được định vị bên trong namespace System. Những namespace khác chứa những lớp không được chuẩn hóa. Nếu một lớp không phải là một phần của tiêu chuẩn, nó sẽ không được hỗ trợ trong tất cả hệ điều hành và môi trường thực thi mà chúng được viết để hỗ trợ C#. Ví dụ, Microsoft thêm vào một vài namespace với SDK của nó như Microsoft.VisualBasic, Microsoft.CSharp, Microsoft.Jscript và Microsoft.Win32. Những namespace này không phải là một phần của tiêu chuẩn ECMA. Do đó chúng có thể không có giá trị trong tất cả môi trường phát triển.

Tìm hiểu những lớp Framework

Như chúng ta đã biết là có hàng ngàn những lớp và những kiểu dữ liệu khác bên trong thư viện cơ sở. Có thể sẽ tốn vài cuốn sách có kích thước như giáo trình này để nói toàn bộ về chúng. Trước khi chúng ta tìm hiểu những lớp cơ bản, bạn có thể xem tổng quan tài liệu trực tuyến để biết thêm các lớp cơ cở. Tất cả các lớp và những kiểu dữ liệu khác được trình bày trong chương này điều là một phần của tiêu chuẩn được xác nhận bởi ECMA.

✍*Lưu ý*: Không những chúng ta có thể sử dụng những kiểu dữ liệu bên trong những lớp thư viện mà chúng ta còn có thể mở rộng những kiểu dữ liệu này.

Lớp Timer

Chúng ta bắt đầu với ví dụ đầu tiên 12.1. Ví dụ minh họa này hết sức đơn giản và được thiết kế không được tốt.

🗗 *Ví dụ 12.1: Hiển thị thời gian.*

```
---------------------------------------------------------------------------
// Timer01.cs: Hiển thị ngày và thời gian
// nhấn Ctrl+C để thoát
namespace Programming_CSharp
{
    using System;
    public class Tester
    {
            public static void Main()
            {
            while (true)
            {
                Console.WriteLine("\n {0}", DateTime.Now);
            }
            }
    }
}
---------------------------------------------------------------------------
```

💻 *Kết quả:*

```
12/24/2001   3:21:20 PM
.......
.......
---------------------------------------------------------------------------
```

Như chúng ta có thể thấy, kết quả chương trình được thực thi vào lúc 3:21 vào ngày 24 tháng 12. Danh sách này thể hiện một đồng hồ xuất hiện ở dòng lệnh, và chúng dường như là được

cập nhật trong mỗi giây đồng hồ. Thật vậy, nó thông thường được cập nhật nhiều hơn một lần, do đó chúng ta lưu ý là giây đồng hồ thay đổi chỉ khi giá trị xuất hiện thật sự khác nhau. Chương trình sẽ chạy mãi đến khi nào ta nhấn thoát bằng Ctrl + C.

Trong chương trình ta sử dụng kiểu dữ liệu DateTime, đây là một cấu trúc được chứa trong namespace System bên trong thư viện cơ sở. Cấu trúc này có một thuộc tính tĩnh là Now trả về thời gian hiện hành. Có nhiều dữ liệu thành viên và những phương thức được thêm vào trong cấu trúc DateTime. Chúng ta có thể tìm hiểu thêm về DateTime trong thư viện trực tuyến về các lớp cơ sở của .NET Framework.

Cách tốt nhất để hiện thị ngày giờ trên màn hình là sử dụng Timer. Một Timer cho phép một xử lý (hình thức của một delegate) được gọi tại một thời gian xác định hay sau một chu kỳ nào đó trôi qua. Framework chứa một lớp Timer bên trong namespace System.Timers. Lớp này được sử dụng trong ví dụ 12.2 theo sau:

Ví dụ 12.2: Sử dụng Timer.

```
---------------------------------------------------------------------------
// Timer02.cs: hiểu thị ngày giờ sử dụng Timer
// nhấn Ctrl+C hay 'q' và Enter để thoát
namespace Programming_CSharp
{
    using System;
    using System.Timers;
    public class Tester
    {
            public static void Main()
            {
        Timer  myTimer = new Timer();
        // khai báo hàm xử lý
        myTimer.Elapsed  += new ElapsedEventHandler( DisplayTimeEvent);
        // khoảng thời gian delay
        myTimer.Interval = 1000;
        myTimer.Start();
        // thực hiện vòng lặp để chờ thoát
        while ( Console.Read() != 'q')
        {
            ;   // không làm gì hết!
        }
            }
            public static void DisplayTimeEvent( object source, ElapsedEventArgs t)
            {
```

```
        Console.Write("\n{0}", DateTime.Now);
            }
    }
}
```

Kết quả:

```
12/24/2001   3:45:20 PM
......
......
```

Kết quả thực hiện cũng giống như ví dụ trước. Tuy nhiên, chương trình này thực hiện tốt hơn nhiều so với chương trình ban đầu. Thay vì cập nhật không ngừng ngày giờ được hiển thị, chương trình này chỉ cập nhật sau khoảng 1 giây. Chúng ta hãy xem kỹ cách mà Timer làm việc. Một đối tượng Timer mới được tạo ra, thuộc tính Interval được thiết lập. Tiếp theo phương thức sẽ được thực hiện sau khoảng thời gian interval được gắn với Timer. Trong trường hợp này là phương thức DisplayTimeEvent sẽ đươc thực thi, phương thức được định nghĩa ở bên dưới.

Khi Timer thực hiện phương thức Start thì nó sẽ bắt đầu tính interval. Một thuộc tính thành viên khác của Timer là AutoReset mà chúng ta cũng cần biết là: nếu chúng ta thay đổi giá trị mặc định của nó từ true sang false, thì sự kiện Timer chỉ thực hiện duy nhất một lần. Khi AutoReset có giá trị true hay ta thiết lập giá trị true thì Timer sẽ kích hoạt sự kiện và thực thi phương thức mỗi một thời gian được đưa ra (interval).

Trong chương trình này vẫn chứa một vòng lặp thực hiện đến khi nào người dùng nhấn ký tự 'q' và Enter. Nếu không chương trình thực hiện tiếp tục vòng lặp. Không có gì thực hiện trong vòng lặp này, nếu muốn chúng ta có thể thêm vào trong vòng lặp những xứ lý khác. Chúng ta cũng không cần thiết phải gọi phương thức DisplayTimeEvent trong vòng lặp bởi vì nó sẽ được gọi tự động vào khoảng thời gian xác định interval.

Timer trong chương trình này dùng để thể hiện ngày giờ trên màn hình. Timer và những sự kiện của Timer cũng có thể được sử dụng cho nhiều chương trình khác. Như chúng ta có thể tạo Timer để tắt một chương trình khác vào một thời điểm đưa ra. Chúng ta cũng có thể tạo chương trình backup thường xuyên để sao chép những dữ liệu quan trọng theo một định kỳ thời gian nào đó. Hay chúng ta cũng có thể tạo một chương trình tự động log off một người sử dụng hay kết thúc một chương trình sau một khoảng thời gian mà không có bất cứ hoạt động nào xảy ra. Nói chung là có rất nhiều cách sử dụng Timer này, và lớp Timer này rất hữa ích.

Lớp về thư mục và hệ thống

Đôi khi chúng ta cần biết thông tin hệ thống của máy mà chương trình đang thực hiện, điều này không khó khăn gì, .NET hỗ trợ một số lớp cơ bản để thực hiện việc này. Trong ví

dụ minh họa 12.3 bên dưới sẽ trình bày cách lấy các thông tin về máy tính và môi trường của nó. Việc thực hiện này thông qua sử dụng lớp Environment, trong lớp này chứa một số dữ liệu thành viên tĩnh và chúng ta sẽ thú vị với thông tin của chúng.

Ví dụ 12.3: Sử dụng lớp Environment.

```
---------------------------------------------------------------------------
// env01.cs: hiển thị thông tin của lớp Environment
namespace Programming_CSharp
{
    using System;
    class Tester
    {
            public static void Main()
            {
            // các thuộc tính
            Console.WriteLine("**************************");
            Console.WriteLine("Command: {0}", Environment.CommandLine);
            Console.WriteLine("Curr Dir: {0}", Environment.CurrentDirectory);
            Console.WriteLine("Sys Dir: {0}", Environment.SystemDirectory);
            Console.WriteLine("Version: {0}", Environment.Version);
            Console.WriteLine("OS Version: {0}", Environment.OSVersion);
            Console.WriteLine("Machine: {0}", Environment.MachineName);
            Console.WriteLine("Memory: {0}", Environment.WorkingSet);
            // dùng một vài các phương thức
            Console.WriteLine("**************************");
            string [] args = Environment.GetCommandLineArgs();
            for( int i = 0; i < args.Length; i++)
            {
                    Console.WriteLine("Arg {0}: {1}", i, args[i]);
            }
            Console.WriteLine("**************************");
            string [] drivers = Environment.GetLogicalDrives();
            for( int i = 0; i < drivers.Length; i++)
            {
                    Console.WriteLine("Drive {0}: {1}", i, drivers[i]);
            }
            Console.WriteLine("**************************");
            Console.WriteLine("Path: {0}",
                    Environment.GetEnvironmentVariable("Path"));
```

```
        Console.WriteLine("**************************");
        }
    }
}
```

Kết quả thực hiện với máy tính của tác giả (kết quả sẽ khác với máy tính của bạn:)

```
*************************
Command: D:\Working\ConsoleApplication1\bin\Debug\Env01.exe
Curr Dir: D:\Working\ConsoleApplication1\bin\Debug
Sys Dir: C:\WINDOWS\System32
Version: 1.0.3705.0
OS Version: Microsoft Windows NT 5.1.2600.0
Machine: MUN
Memory: 4575232
*************************
Arg 0: D:\Working\ConsoleApplication1\bin\Debug\Env01.exe
*************************
Drive 0: A:\
Drive 1: C:\
Drive 2: D:\
Drive 3: E:\
*************************
Path:
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS;C:\WINDO
WS\COMMAND;C:\NC
*************************
```

Như chúng ta thấy thì những thành viên tĩnh của lớp Environment cung cấp cho ta những thông tin hệ thống và môi trường. Đầu tiên là lệnh thực hiện được đưa ra chính là chương trình đang thực thi tức là chương trình Env01.exe, thuộc tính được dùng để lấy là Command-Line. Thư mục hiện hành chính là thư mục chứa chương trình thực thi thông qua thuộc tính CurrentDirectory. Tương tự như vậy các thuộc tính hệ thống như: thư mục hệ thống, phiên bản OS, tên máy tính, bộ nhớ cũng được lấy ra.

Tiếp theo là hai phương thức của lớp Environment trả về mảng chuỗi ký tự, bao gồm phương thức lấy đối mục dòng lệnh GetCommandLineArgs và phương thức nhận thông tin về ở đĩa logic trong máy tính GetLogicalDrives. Hai vòng lặp đơn giản là để xuất giá trị từng thành phần trong mảng ra.Cuối cùng là phương thức GetEnvironmentVariable nhận biến môi trường và những giá trị của chúng trong hệ thống hiện thời.

Lớp Math

Từ đầu tới giờ chúng ta chỉ thực hiện các phép toán cơ bản như cộng, trừ, nhân, chia, chia dư. Còn rất nhiều các phép toán mạnh hơn và cũng thường sử dụng mà chúng chưa được đề cập tới. C# cung cấp một tập hợp các phép toán toán học bên trong những lớp cơ sở. Chúng được chứa bên trong của namespace System.Math. Bảng 12.1 sau liệt kê những hàm toán học.

Lớp Math là lớp **sealed**, do đó chúng ta không thể xây dựng một lớp mới mà kế thừa từ lớp này được. Thêm vào đó tất cả những lớp và dữ liệu thành viên đều là tĩnh, do vậy chúng ta không thể tạo một đối tượng có kiểu Math. Thay vào đó chúng ta sẽ sử dụng những thành viên và phương thức với tên lớp.

Lớp Math	
Phương thức	**Mô tả**
Abs	Trả về trị tuyệt đối của một số
Ceiling	Trả về giá trị nhỏ nhất hay bằng giá trị đưa ra
Exp	Trả về giá trị e với mũ đưa ra
Floor	Trả về giá trị lớn nhất hay bằng giá trị đưa ra
IEEERemainder	Trả về phần dư của phép chia hai hai số thực. Phép chia này theo tiêu chuẩn của IEEE cho phép toán dấu chấm động nhị phân.
Log	Trả về giá trị logarit của giá trị đưa ra
Log10	Trả về giá trị logarit cơ số 10 của số đưa ra
Max	Trả về số lớn trong hai số
Min	Trả về số nhỏ trong hai số
Pow	Trả về kết quả x^y
Round	Trả về giá trị được làm tròn
Sign	Trả về giá trị dấu của một số. -1 nếu số âm và 1 nếu số dương
Sqrt	Trả về căn bậc hai của một số
Acos	Trả về giá trị một góc mà cosin bằng với giá trị đưa ra
Asin	Trả về giá trị một góc mà sin bằng với giá trị đưa ra
Atan	Trả về giá trị của một góc mà tang bằng với góc đưa ra
Atan2	Trả về giá trị của một góc mà tang bằng với tang của điểm (x,y) đưa ra
Cos	Trả về giá trị cosin của một góc đưa ra
Cosh	Trả về giá trị hyperbolic cosin của góc đưa ra
Sin	Trả về giá trị sin của góc đưa ra

Sinh	Trả về giá trị hyperbolic của góc đưa ra
Tan	Trả về giá trị tang của góc
Tanh	Trả về giá trị hyperbolic tang của góc.

Hình 12.1 : Phương thức của Math.

Ngoài ra lớp Math này cũng đưa vào hai hằng số: PI và số E, PI trả về giá trị pi trong toán học như là 3.14159265358979323846 Giá trị E trả về giá trị 2.7182818284590452354.

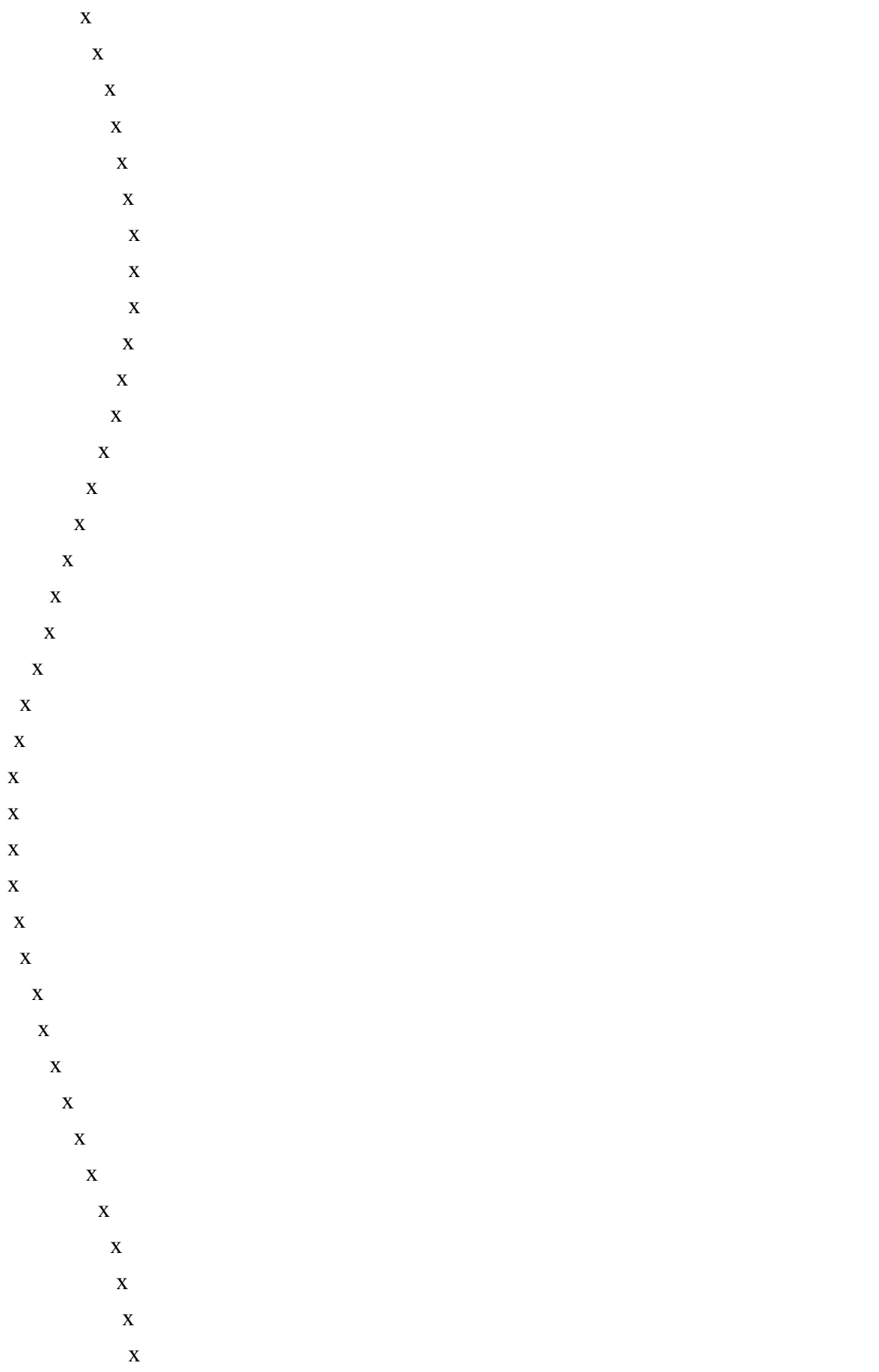
Hầu hết các hàm toán học trong bảng 12.1 trên dễ hiểu và dễ sử dụng. Ví dụ 12.4 sau minh họa việc sử dụng một số hàm toán học như sau:

Ví dụ 12.4: Sử dụng một vài hàm toán học.

```
---------------------------------------------------------------------------
using System;
namespace Programming_CSharp
{
    public class Tester
    {
        public static void Main()
        {
            int  val2;
            char ch;
            for(double ctr = 0.0; ctr <= 10; ctr += .2)
                {
                  val2 = (int) System.Math.Round((10*System.Math.Sin(ctr)));
                  for( int ctr2 = -10; ctr2 <=10; ctr2++)
                  {
                    if (ctr2 == val2)
                        ch = 'x';
                    else
                        ch = ' ';
                    Console.Write("{0}", ch);
                  }
                  Console.WriteLine(" ");
             }
            }
    }
}
---------------------------------------------------------------------------
```

Kết quả:

```
          x
```



```
            x
            x
            x
           x
          x
         x
        x
       x
      x
     x
    x
   x
```

Vòng lặp đầu tiên thực hiện thông qua một biến lặp là một giá trị double, mỗi bước lặp tăng . 2. Giá trị Sin được lấy thông qua việc sử dụng hàm Math.Sin. Do giá trị Sin từ -1 đến 1 và để cho dễ hiển thị hơn, giá trị này được chuyển từ -10 đến 10. Để chuyển thành giá trị này thì ta nhân với 10 rồi sau đó thực hiện việc làm tròn bằng cách dùng hàm Round của Math.
Kết quả của phép nhân và làm tròn là một giá trị từ -10 đến 10 được gán cho val2. Vòng lặp **for** thứ hai thực hiện việc xuất một ký tự ra màn hình.

Lớp thao tác tập tin

Khả năng để viết thông tin vào trong một tập tin hay đọc thông tin từ trong một tập tin có thể làm cho chương trình của chúng ta có nhiều hiệu quả hơn. Hơn nữa, có rất nhiều lần khi chúng ta muốn có khả năng làm việc với những tập tin hiện hữu. Phần tiếp sau chúng ta sẽ tìm hiểu những đặc tính cơ bản của việc thao tác trên tập tin. Điều này sẽ được theo sau bởi một khái niệm quan trọng của tập tin là luồng (stream).

Sao chép một tập tin

Một lớp tập tin tồn tại bên trong lớp cơ sở gọi là File, lớp này được định vị bên trong namespace System.OI. Lớp File chứa một số các phương thức tĩnh được sử dụng để thao tác trên tập tin. Thật vậy, tất cả các phương thức bên trong lớp File điều là tĩnh. Bảng 12.2 liệt kê những phương thức chính của lớp File.

Lớp File	
Phương thức	**Mô tả**
AppendText	Nối văn bản vào trong một tập tin
Copy	Tạo ra một tập tin mới từ tập tin hiện hữu

Create	Tạo ra một tập tin mới ở một vị trí xác định
CreateText	Tạo ra tập tin lưu giữ text
Delete	Xóa tập tin ở vị trí xác định. Tập tin phải hiện hữu nếu không sẽ phát sinh ra ngoại lệ.
Exists	Kiểm tra xem tập tin có thực sự hiện hữu ở vị trí nào đó.
GetAttributes	Lấy thông tin thuộc tính của tập tin. Thông tin này bao gồm: tập tin có bị nén hay không, tên thư mục, có thuộc tính ẩn, thuộc tính chỉ đọc, tập tin hệ thống...
GetCreationTime	Trả về ngày giờ tập tin được tạo ra
GetLastAccessTime	Trả về ngày giờ tập tin được truy cập lần cuối
GetLastWriteTime	Trả về ngày giờ tập tin được viết lần cuối
Move	Cho phép tập tin được di chuyển vào vị trí mới và đổi tên tập tin.
Open	Mở một tập tin tại vị trí đưa ra. Bằng việc mở tập tin này chúng ta có thể viết thông tin hay đọc thông tin từ nó.
OpenRead	Mở một tập tin hiện hữu để đọc
OpenText	Mở một tập tin để đọc dạng text
OpenWrite	Mở một tập tin xác định để viết
SetAttributes	Thiết lập thuộc tính cho tập tin
SetCreationTime	Thiết lập ngày giờ tạo tập tin
SetLastAccessTime	Thiết lập lại ngày giờ mà tập tin được truy cập lần cuối
SetLastWriteTime	Thiết lập ngày giờ mà tập tin được cập nhật lần cuối.

Bảng 12.2 : Một số phương thức chính thao tác tập tin.

Chương trình 12.5 sau minh họa việc sao chép một tập tin.

Ví dụ 12.5: Sao chép một tập tin.

```
---------------------------------------------------------------------------
// file : filecopy.cs: sao chép một tập tin
// sử dụng tham số dòng lệnh
namespace Programming_CSharp
{
    using System;
    using System.IO;
    public class Tester
    {
        public static void Main()
```

```
        {
            string[] CLA = Environment.GetCommandLineArgs();
            if ( CLA.Length < 3)
            {
                    Console.WriteLine("Format: {0} orig-file new-file", CLA[0]);
            }
            else
            {
                    string origfile = CLA[1];
                    string newfile = CLA[2];
                    Console.Write("Copy...");
                try
                {
                     File.Copy(origfile, newfile);
                }
                catch (System.IO.FileNotFoundException)
                {
                     Console.WriteLine("\n{0} does not exist!", origfile);
                     return;
                }
                catch (System.IO.IOException)
                {
                    Console.WriteLine("\n{0} already exist!", newfile);
                    return;
                }
                catch (Exception e)
                {
                    Console.WriteLine("\nAn exception was thrown trying to copy file");
                    Console.WriteLine();
                    return;
                }
                Console.WriteLine("...Done");
            }
        }
    }
}
```

Chương trình thực hiện bằng cách sau khi biên dịch ra tập tin filecopy.exe ta gọi thực hiện tập tin này với tham số dòng lệnh như sau:

```
filecopy.exe filecopy.cs filecopy.bak
```

Tập tin filecopy.cs đã hiện hữu và tập tin filecopy.bak thì chưa hiện hữu cho đến khi lệnh này thực thi. Sao khi thực thi xong chương trình tập tin filecopy.bak được tạo ra. Nếu chúng ta thực hiện chương trình lần thứ hai cũng với các tham số như vậy, thì chúng ta sẽ nhận được kết quả xuất như sau:

```
Copy...
filecopy.bak already exists!
```

Nếu chúng ta thực hiện chương trình mà không có bất cứ tham số nào, hay chỉ có một tham số, chúng ta sẽ nhận được kết quả như sau :

```
Format d:\working\filecopy\filecopy.exe orig-file new-file
```

Cuối cùng, điều tệ nhất xảy ra là chúng ta thực hiện sao chép nhưng tập tin nguồn không tồn tại:

```
Copy...
filecopy.cs does not exist!
```

Như chúng ta ta thấy tất cả các kết quả có thể có của chương trình minh họa 12.5 trên. Chương trình thực hiện việc sao chép một tập tin và nó kiểm tra tất cả các tình huống có thể có và thực hiện việc xử lý các ngoại lệ. Điều này cho thấy chương trình vừa đáp ứng được mặt logic của lập trình vừa đáp ứng được việc xử lý các ngoại lệ.

Như chúng ta biết trong lệnh sau:

```
string[] CLA = Environment.GetCommandLineArgs();
```

thực hiện việc lấy các tham số dòng lệnh được cấp cho chương trình. Lớp Environment này chúng ta đã tìm hiểu và đã sử dụng trong phần trước.

Lệnh sau kiểm tra xem chương trình có nhận được ít hơn giá trị tham số dòng lệnh hay không.

```
if ( CLA.Length < 3)
{
   Console.WriteLine("Format: {0} orig-file new-file", CLA[0]);
}
```

Nếu giá trị này nhỏ hơn 3, người sử dụng đã không cung cấp đủ thông tin. Nên nhớ rằng, sử dụng phương thức GetCommandLineArgs, thì giá trị đầu tiên mà chúng ta nhận được là tên của chương trình. Phần còn lại là các tham số dòng lệnh được theo sau. Điều này có nghĩa rằng chúng ta cần thiết 3 giá trị: tên chương trình, tên tập tin nguồn, tên tập tin mới. Nếu chúng ta nhập vào không đủ 3 tham số dòng lệnh, thì chương trình sẽ xuất ra thông báo với tên của chương trình thực hiện được đọc bởi GetCommandLineArgs.

Nếu cung cấp đầy đủ các tham số, việc xử lý sao chép sẽ được thực hiện. Để dễ theo dõi thì chương trình xuất ra thông báo việc sao chép file:

```
Console.Write("Copy...");
.....
Console.WriteLine("...Done");
```

Mặc dùng việc sao chép tập tin không có gì phức tạp, nhưng chúng ta lưu ý rằng trong chương trình trên chúng ta có thêm vào các đoạn xử lý ngoại lệ. Việc gọi hàm Copy của File được bao bọc trong khối **try**, và sau đó là ba thể hiện của **Catch** theo sau. Bởi vì có rất nhiều thứ có thể gây ra lỗi do các hoạt động trên tập tin, đề nghị rằng chúng ta nên đảm bảo cho chương trình của chúng ta thực hiện việc bắt giữ và xử lý các ngoại lệ tương ứng.

Hầu hết những phương thức File có những ngoại lệ đã được định nghĩa cho một số các lỗi quan trọng có thể xuất hiện. Khi chúng ta tra cứu tài liệu trực tuyến cho một lớp. Chúng ta sẽ tìm thấy bất cứ những ngoại lệ nào được định nghĩa cho phương thức đưa ra. Một cách thực hành lập trình tốt là thêm vào các đoạn xử lý ngoại lệ với bất cứ ngoại lệ nào có thể xuất hiện. Trong chương trình ngoại lệ đầu tiên được xử lý cho việc gọi hàm Copy() là:

```
catch (System.IO.FileNotFoundException)
{
        Console.WriteLine("\n{0} does not exist!", origfile);
        return;
}
```

Ngoại lệ này được phát sinh khi một tập tin mà chúng ta cố sao chép không tìm thấy, ngoại lệ này được đặt tên là FileNotFoundException.

Ngoại lệ tiếp theo được bắt là IOException. Ngoại lệ này được phát sinh từ một số ngoại lệ khác. Bao gồm một thư mục không tìm thấy (DirectoryNotFoundException), kết thúc của tập tin không được tìm thấy (EndOfStreamException), có lỗi đọc tập tin (FileLoadException), hay là lỗi tập tin không được tìm thấy, nhưng ngoại lệ này sẽ được bắt trong xử lý ngoại lệ bên trên. Ngoại lệ này cũng được phát sinh khi tập tin mà chúng ta cần tạo mới đã tồn tại.

Cuối cùng, chúng ta sẽ bắt giữ những ngoại lệ không mong đợi còn lại bằng sử dụng ngoại lệ chung hay ngoại lệ tổng quát. Bởi vì chúng ta không biết nguyên nhân của việc xảy ra ngoại lệ, ở đây chỉ có thể hiển thị thông điệp của chính bản thân ngoại lệ phát sinh.

✍*Ghi chú*: Nên sử dụng xử lý ngoại lệ khi thao tác trên tập tin. Phong cách lập trình tốt là không nên nghĩ rằng người sử dụng sẽ cung cấp cho chương trình mọi thứ mà chương trình cần nhất là khi sử dụng tham số dòng lệnh.

Lấy thông tin về tập tin

Ngoài lớp File được cung cấp từ thư viện cơ sở, một lớp khác cũng thường xuyên được sử dụng thể làm việc với tập tin là lớp FileInfo. Chương trình 12.6 minh họa việc sử dụng lớp này. Trong chương trình này sẽ lấy một tên tập tin và hiển thị kích thước và những ngày quan trọng liên quan đến việc tạo, bổ sung tập tin.

🗗 *Ví dụ 12.6: Sử dụng lớp FileInfo.*

```
// Filsize.cs
namespace Programming_CSharp
{
    using System;
    using System.IO;
    public class Tester
    {
        public static void Main()
        {
        string[] CLA = Environment.GetCommandLineArgs();
        FileInfo    fiExe = new FileInfo( CLA[0] );
        if ( CLA.Length < 2)
        {
                Console.WriteLine("Format: {0} filename", fiExe.Name);
        }
        else
        {
                try
                {
            FileInfo  fiFile = new FileInfo( CLA[1]);
            if (fiFile.Exists)
            {
                    Console.WriteLine("******************************");
                    Console.WriteLine("{0}   {1}",fiFile.Name, fiFile.Length);
                    Console.WriteLine("******************************");
                    Console.WriteLine("Last access: {0}",fiFile.LastAccessTime);
                    Console.WriteLine("Last write:  {0}", fiFile.LastWriteTime);
                    Console.WriteLine("Creation: {0}", fiFile.CreationTime);
                    Console.WriteLine("******************************");
            }
            else
            {
                    Console.WriteLine("{0} doesn't exist!", fiFile.Name);
            }
                }
                catch (System.IO.FileNotFoundException)
                {
```

```
            Console.WriteLine("\n{0} does not exists!", CLA[1]);
            return;
                }
                catch (Exception e)
                {
            Console.WriteLine("\n An exception was thrown trying to copy file");
            Console.WriteLine();
            return;
                }// end catch
        }// end else
        }// end Main
    }// end class
}// end namespace
```

Kết quả:

```
*****************************
filesize.cs   1360
*****************************
Last access: 12/5/2002 12:00:00 AM
Last write:  12/5/2002 5:50:50 PM
Creation: 12/5/2002 5:53:31 PM
*****************************
```

Một đối tượng FileInfo được tạo ra và gắn với một tập tin tương ứng:

```
FileInfo  fiInfo = new FileInfo( CLA[1]);
```

Tham số của bộ khởi dựng lớp FileInfo xác định tên của tập tin mà nó sẽ chứa nhận thông tin, trong trường hợp này nó sẽ lấy tham số thứ hai của tham số dòng lệnh làm tập tin mà nó sẽ thực hiện. Nếu người dùng không nhập tên tập tin thì chương trình sẽ in ra tên của chương trình.

Làm việc với tập tin dữ liệu

Phần trước chúng ta đã thực hiện công việc như lấy thông tin của tập tin và sao chép tập tin sang một tập tin mới. Việc thực hiện quan trọng của tập tin là đọc và viết những thông tin từ tập tin. Trong phần này chúng ta sẽ tìm hiểu về luồng nhập xuất và cách tạo mới một tập tin để ghi hay mở một tập tin đã tồn tại để đọc thông tin.

Luồng nhập xuất

Thuật ngữ tập tin thì nói chung là liên quan đến những thông tin lưu trữ bên trong ỗ đĩa hoặc trong bộ nhớ. Khi làm việc với tập tin, chúng ta bao hàm với việc sử dụng một luồng.

Nhiều người nhầm lẫn về sự khác nhau giữa tập tin và luồng. Một luồng đơn giản là luồng của thông tin, chứa thông tin sẽ được chuyển qua, còn tập tin thì để lưu trữ thông tin.
Một luồng được sử dụng để gởi và nhận thông tin từ bộ nhớ, từ mạng, web, từ một chuỗi,...Một luồng còn được sử dụng để đi vào và ra với một tập tin dữ liệu.

Thứ tự của việc đọc một tập tin

Khi đọc hay viết một tập tin, cần thiết phải theo một trình tự xác định. Đầu tiên là phải thực hiện công việc mở tập tin. Nếu như tạo mới tập tin, thì việc mở tập tin cùng lúc với việc tạo ra tập tin đó. Khi một tập tin đã mở, cần thiết phải tạo cho nó một luồng để đặt thông tin vào trong một tập tin hay là lấy thông tin ra từ tập tin. Khi tạo một luồng, cần thiết phải chỉ ra thông tin trực tiếp sẽ được đi qua luồng. Sau khi tạo một luồng gắn với một tập tin, thì lúc này chúng ta có thể thực hiện việc đọc ghi các dữ liệu trên tập tin. Khi thực hiện việc đọc thông tin từ một tập tin, chúng ta cần thiết phải kiểm tra xem con trỏ tập tin đã chỉ tới cuối tập tin chưa, tức là chúng ta đã đọc đến cuối tập tin hay chưa. Khi hoàn thành việc đọc ghi thông tin trên tập tin thì tập tin cần phải được đóng lại.
Tóm lại các bước cơ bản để làm việc với một tậo tin là:

- Bước 1: Mở hay tạo mới tập tin
- Bước 2: Thiết lập một luồng ghi hay đọc từ tập tin
- Bước 3: Đọc hay ghi dữ liệu lên tập tin
- Bước 4: Đóng lập tin lại

Các phương thức cho việc tạo và mở tập tin

Có nhiều kiểu luồng khác nhau. Chúng ta sẽ sử dụng những luồng khác nhau và những phương thức khác nhau phụ thuộc vào kiểu dữ liệu bên trong của tập tin. Trong phần này, việc đọc/ghi sẽ được thực hiện trên tập tin văn bản. Trong phần kế tiếp chúng ta học cách đọc và viết thông tin trên tập tin nhị phân. Thông tin nhị phân bao hàm khả năng mạnh lưu trữ giá trị số và bất cứ kiểu dữ liệu nào khác.

Để mở một tập tin trên đĩa cho việc đọc và viết tập tin văn bản, chúng ta cần phải sử dụng cả hai lớp File và FileInfo. Một vài phương thức có thể sử dụng trong những lớp này. Các phương thức này bao gồm:

AppendText	Mở một tập tin để và tập tin này có thể được thêm văn bản vào trong nó. Tạo luồng StreamWriter sử dụng để thêm vào trong văn bản.
Create	Tạo mới một tập tin
CreateText	Tạo và mở một tập tin văn bản. Tạo ra một luồng StreamWriter.
Open	Mở một tập tin để đọc/viết. Mở một FileStream
OpenRead	Mở một tập tin để đọc
OpenText	Mở một tập tin văn bản để đọc. Tạo ra StreamReader để sử dụng.
OpenWrite	Mở một tập tin cho việc đọc và ghi.

Làm thế nào để chúng ta có thể biết được khi nào sử dụng lớp File chính xác hơn là sử dụng lớp FileInfo nếu chúng cùng chứa những phương thức tương tự với nhau. Thật ra hai lớp này có nhiều sự khác biệt. Lớp File chứa tất cả các phương thức tĩnh, thêm vào đó lớp File tự động kiểm tra permission trên một tập tin. Trong khi đó nếu muốn dùng lớp FileInfo thì phải tạo thể hiện của lớp này. Nếu muốn mở một tập tin chỉ một lần thì tốt nhất là sử dụng lớp File, còn nếu chúng ta tổ chức việc sử dụng tập tin nhiều lần bên trong chương trình, tốt nhất là ta dùng lớp FileInfo. Hoặc nếu không chắc chắn cách sử dụng thì chúng ta có thể sử dụng lớp FileInfo.

Viết vào một tập tin văn bản

Cách tốt nhất để nắm vững cách thức làm việc với tập tin là chúng ta sẽ bắt tay vào tìm hiểu chương trình. Trong phần này chúng ta sẽ tìm hiểu một ví dụ minh họa việc tạo ra một tập tin văn bản rồi sau đó viết lại thông tin vào nó.

Ví dụ 12.7: Viết dữ liệu vào tập tin văn bản.

```
---------------------------------------------------------------------------
//writing.cs:: viết vào một tập tin văn bản
namespace Programming_CSharp
{
    using System;
    using System.IO;
    public class Tester
    {
            public static void Main(String[] args)
            {
                if (args.Length < 1)
                {
                    Console.WriteLine("Phai nhap ten tap tin.");
                }
                else
                {
                    StreamWriter  myFile = File.CreateText( args[0]);
                    myFile.WriteLine("Khong co viec gi kho");
                    myFile.WriteLine("Chi so long khong ben");
                    myFile.WriteLine("Dao nui va lap bien");
                    myFile.WriteLine("Quyet chi at lam nen");
                    for(int i=0; i < 10; i++)
                    {
```

```
                myFile.Write("{0} ",i);
                    }
                    myFile.Close();
                }
            }
    }
}
```

Khi chạy chương trình trên chúng ta phải cung cấp tên của tập tin được tạo mới, nếu không cung cấp thì chương trình sẽ không thực hiện việc tạo tập tin. Giả sử chúng ta có cung cấp một tham số dòng lệnh cho chương trình thì một tập tin văn bản mới được tạo ra có nội dung như sau:

```
Khong co viec gi kho
Chi so long khong ben
Dao nui va lap bien
Quyet chi at lam nen
0 1 2 3 4 5 6 7 8 9
```

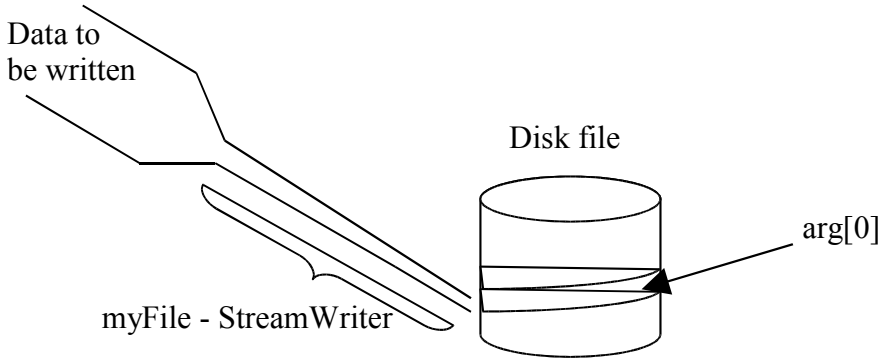
Trong chương trình trên không thực hiện việc xử lý ngoại lệ. Điều này dẫn đến chương trình có thể phát sinh ra những ngoại lệ và những ngoại lệ này không được xử lý. Đây là cách lập trình không tốt, nên yêu cầu người đọc nên thêm các xử lý ngoại lệ vào chương trình trên, ngoại lệ này cũng tương tự như ngoại lệ trong ví dụ trước.

Như chúng ta thấy hàm Main có tham số và tham số này sẽ được nhận thông qua dòng lệnh, trong các ví dụ trước, chương trình nhận tham số dòng lệnh thông qua lớp Environment, còn ở đây ta khai báo tham số dòng lệnh trực tiếp cho chương trình. Dòng lệnh đầu tiên của hàm Main() là kiểm tra số tham số nhập vào, nếu không có tham số nào chương trình sẽ xuất ra thông báo là không có tên tập tin và kết thúc chương trình.

Trong trường hợp cung cấp tham số đầy đủ chương trình sẽ thực hiện việc tạo mới tập tin. Phương thức CreateText của lớp File được gọi để tạo ra một đối tượng StreamWriter mới gọi là myFile. Tham số được truyền cho hàm là tên của tập tin sẽ được tạo. Kết quả cuối cùng của dòng lệnh này là tạo ra một tập tin văn bản mới. Dữ liệu s4 được đưa vào tập tin thông qua StreamWriter với thể của nó là myFile.

✍*Ghi chú*: Nếu một tập tin hiện hữu với cùng tên như tập tin mà chúng ta tạo ra thì tập tin cũ sẽ được viết chồng, tức là dữ liệu bên trong tập tin cũ sẽ bị xóa mất.

Hình sau minh họa việc thực hiện tạo tập tin và đưa dữ liệu vào



Hình 12.3 : Mô tả thực hiện tạo tập tin và đưa dữ liệu vào

Khi một luồng được thiết lập và chỉ đến một tập tin, chúng ta có thể viết vào trong luồng và nó sẽ được viết vào tập tin:

```
myFile.WriteLine("Khong co viec gi kho");
```

Dòng lệnh trên viết một chuỗi vào trong tập tin, việc viết này giống như là viết ra màn hình console. Nhưng ở đây là được viết ra thiết bị khác, tức là ra tập tin. Sau khi thực hiện toàn bộ công việc, cần thiết phải đóng luồng lại bằng cách gọi phương thức Close.

Đọc tập tin văn bản

Đọc dữ liệu từ tập tin văn bản cũng tương tự như việc viết thông tin vào nó. Ví dụ minh họa tiếp sau đây thực hiện việc đọc tập tin mà chúng ta đã tạo ra từ chương trình minh họa 12.7 trước. Đây là chương trình đọc tập tin văn bản.

Ví dụ 12.8: Đọc một tập tin văn bản.

```
-------------------------------------------------------------------------
using System;
using System.IO;
namespace Programming_CSharp
{
        public class Tester
    {
        public static void Main(string[] args)
        {
        if ( args.Length < 1)
        {
            Console.WriteLine("Phai nhap ten tap tin");
        }
        else
        {
```

```
            string buffer;
            StreamReader myFile = File.OpenText( args[0]);
            while ( (buffer = myFile.ReadLine()) !=null)
            {
            Console.WriteLine(buffer);
            }
            myFile.Close();
        }
            }
    }
}
```

Nếu chúng ta nhập đúng tên của tập tin vừa tạo trong ví dụ trước thì kết quả chương trình thực hiện như sau:

Kết quả:

```
Khong co viec gi kho
Chi so long khong ben
Dao nui va lap bien
Quyet chi at lam nen
0  1  2  3  4  5  6  7  8  9
```

Nếu chương trình được cung cấp đầy đủ tên tập tin thì nó sẽ bắt đầu thực hiện việc đọc thông tin. Đầu tiên là khai báo một chuỗi dùng làm chỗ lưu thông tin đọc tử tập tin ra. Để mở tập tin văn bản thì phương thức OpenText được gọi. Phương thức này được truyền vào tên của tập tin văn bản cần mở. Một lần nữa một luồng tên myFile được tạo ra gắn với tập tin đã cho, luồng này có kiểu là StreamReader. Phương thức ReadLine() của myFile thực hiện việc đọc từng dòng trong tập tin và sau đó được xuất ra màn hình console. Việc đọc này kết hợp với việc kiểm tra là đã đọc đến cuối tập tin chưa, nếu đọc đến cuối tập tin, tức là hàm ReadLine() trả về chuỗi rỗng, lúc này chương trình sẽ kết thúc việc đọc. Cuối cùng hàm Close được gọi để đóng tập tin lại, và chương trình chấm dứt.

Trong chương trình này cũng không xử lý ngoại lệ, nếu tập tin không tồn tại chương trình sẽ phát sinh ra ngoại lệ và ngoại lệ này không được xử lý. Đề nghị người đọc nên đưa vào các đoạn xử lý ngoại lệ tương tự như ví dụ 12.6

Viết thông tin nhị phân vào tập tin

Nếu chúng ta sử dụng một tập tin văn bản, thì khi chúng ta lưu dữ liệu kiểu số thì phải thực hiện việc chuyển đổi sang dạng chuỗi ký tự để lưu vào trong tập tin văn bản và khi lấy ra

ta cũng lấy được giá trị chuỗi ký tự do đó ta phải chuyển sang dạng số. Đôi khi chúng ta muốn có cách thức nào đó tốt hơn để lưu trực tiếp giá trị vào trong tập tin và sau đó đọc trực tiếp giá trị ra từ tập tin. Ví dụ khi viết một số lượng lớn các số integer vào trong tập tin như là những số nguyên, thì khi đó ta có thể đọc các giá trị này ra như là số integer. Trường hợp nếu chúng được viết vào tập tin với dạng văn bản, thì khi đọc ra ta phải đọc ra văn bản và phải chuyển mỗi giá trị từ một chuỗi đến các số integer. Tốt hơn việc phải thực hiện thêm các bước chuyển đổi, ta có thể gắn một kiểu luồng nhị phân BinaryStream vào trong một tập tin, rồi sau đó đọc và ghi thông tin nhị phân từ luồng này.

Tiếp theo ta sẽ xem một ví dụ minh họa việc đọc viết dữ liệu nhị phân vào một tập tin. Mặc dù trong chương trình này thực hiện việc viết 100 giá trị integer vào trong một tập tin nhưng có thể dễ dàng viết bất cứ kiểu dữ liệu nào khác.

✍*Ghi chú*: Thông tin nhị phân là thông tin đã được định dạng kiểu lưu trữ dữ liệu.

🗗 *Ví dụ 12.9: Viết vào một tập tin nhị phân.*

```
//binarywriter.cs
using System;
using System.IO;
namespace Programming_CSharp
{
    public class Tester
    {
            public static void Main(string[] args)
            {
        if ( args.Length < 1)
        {
            Console.WriteLine("Phai nhap ten tap tin!");
        }
        else
        {
            FileStream  myFile = new FileStream( args[0], FileMode.CreateNew);
            BinaryWriter  bwFile = new BinaryWriter(myFile);
            for (int i=0; i < 100; i++)
            {
            bwFile.Write(i);
            }
            bwFile.Close();
            myFile.Close();
        }
```

```
        }
    }
}
```

Cũng tương tự như các ví dụ trên thì tên tập tin được đưa vào tham số dòng lệnh. Nếu chương trình được nhập các tham số đầy đủ, chương trình sẽ thực hiện việc viết thông tin nhị phân vào trong tập tin, và không có output ra màn hình console. Nếu chúng ta mở tập tin và xem thì chỉ thấy các ký tự mở rộng được thể hiện, sẽ không thấy những số đọc được.

Trong chương trình này cũng chưa thực hiện việc xử lý các ngoại lệ. Nếu thực hiện việc viết thông tin vào một tập tin đã hiện hữu, thì một ngoại lệ sẽ được phát sinh khi thực hiện lệnh:

```
FileStream  myFile = new FileStream( args[0], FileMode.CreateNew);
```

Trong chương trình 12.9 này thực hiện việc tạo và mở tập tin khác với việc mở tập tin văn bản. Lệnh trên tạo một đối tượng FileStream gọi là myFile. Luồng này được gắn với tập tin thông qua bộ khởi dựng. Tham số đầu tiên của bộ khởi dựng là tên của tập tin mà chúng ta tạo (args[0]), tham số thứ hai là mode mà chúng ta mở. Tham số này là giá trị kiểu liệt kê FileMode, trong chương trình thì do chúng ta thực hiện việc tạo mới để ghi nên sử dụng giá trị FileMode.CreateNew. Bảng 12.3 sau liệt kê những mode giá trị khác trong kiểu liệt kê FileMode.

Giá trị	**Định nghĩa**
Append	Mở một tập tin hiện hữu hoặc tạo một tập tin mới
Create	Tạo một tập tin mới. Nếu một tập tin đã hiện hữu, nó sẽ bị xóa và một tập tin mới sẽ được tạo ra với cùng tên.
CreateNew	Tạo một tập tin mới. Nếu một tập tin đã tồn tại thì một ngoại lệ sẽ được phát sinh.
Open	Mở tập tin hiện hữu.
OpenOrCreate	Mở tập tin hay tạo tập tin mới nếu tập tin chưa tồn tại
Truncate	Mở một tập tin hiện hữu và xóa nội dung của nó.

Bảng 12.4: Giá trị của FileMode.

Sau khi tạo ra FileStream, chúng ta cần thiết phải thiết lập để nó làm việc với dữ liệu nhị phân. Dòng lệnh tiếp theo:

```
BinaryWriter  bwFile = new BinaryWriter(myFile);
```

Dòng này thiết lập một kiểu viết dữ liệu nhị phân vào trong luồng bằng cách khai báo kiểu BinaryWrite. Đối tượng thể hiện của BinaryWrite là bwFile được tạo ra. myFile được truyền vào bộ khởi dựng BinaryWrite, nó sẽ gắn bwFile với myFile.

```
for (int i=0; i < 100; i++)
{
    bwFile.Write(i);
```

```
    }
```

Vòng lặp này thực hiện việc viết trực tiếp giá trị integer vào trong BinaryWrite bwFile bằng cách sử dụng phương thức Write. Dữ liệu được viết có thể là kiểu dữ liệu đặc biệt khác. Trong chương trình này thì sử dụng giá trị integer. Khi thực hiện xong các công việc viết vào tập tin, chúng ta cần thiết phải đóng luồng mà chúng tả đã mở.

Đọc thông tin nhị phân từ tập tin

Trong phần trước chúng ta đã thực hiện việc viết thông tin nhị phân vào trong tập tin, và bây giờ chúng ta mong muốn được đọc các thông tin đã ghi vào trong tập tin. Việc đọc thông tin cũng khá đơn giản như là việc viết vào. Chương trình 12.10 sau minh họa cho công việc này.

Ví dụ 12.10: Đọc thông tin nhị phân.

```
---------------------------------------------------------------------------
// BinaryRead.cs: Doc thong tin tu file nhi phan
namespace    Programming_CSharp
{
    using System;
    using System.IO;
    public class Tester
    {
            public static void Main( String[] args)
            {
        if ( args.Length < 1)
        {
                Console.WriteLine("Phai nhap ten tap tin");
        }
        else
        {
                FileStream  myFile  = new FileStream( args[0], FileMode.Open);
                BinaryReader  brFile = new BinaryReader(myFile);
                // đọc dữ liệu
                Console.WriteLine("Dang doc tap tin....");
                while (brFile.PeekChar() != -1)
                {
                    Console.Write("<{0}>", brFile.ReadInt32());
                }
                Console.WriteLine("....Doc xong");
                brFile.Close();
```

```
            myFile.Close();
        }
        }
    }
}
```

Kết quả:

```
Dang doc tap tin....
<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> <30> <31> <32>
<33>
<34> <35> <36> <37> <38> <39> <40> <41> <42> <43> <44> <45> <46> <47> <48>
<49>
<50> <51> <52> <53> <54> <55> <56> <57> <58> <59> <60> <61> <62> <63> <64>
<65>
<66> <67> <68> <69> <70> <71> <72> <73> <74> <75> <76> <77> <78> <79> <80>
<81>
<82> <83> <84> <85> <86> <87> <88> <89> <90> <91> <92> <93> <94> <95> <96>
<97>
<98> <99> ....Doc xong!
```

Với ứng dụng này, chúng ta có thể đọc dữ liệu mà chúng ta đã viết trong ví dụ trước. Trong ví dụ này chúng ta tạo ra luồng FileStream. Lúc này, mode thao tác của tập tin được sử dụng là mode FileMode.Open. Sau đó chúng ta thực hiện việc gắn luồng này với luồng BinaryReader trong dòng tiếp sau, luồng này sẽ giúp cho chúng ta đọc thông tin nhị phân:

```
FileStream  myFile  = new FileStream( args[0], FileMode.Open);
BinaryReader  brFile = new BinaryReader(myFile);
```

Sau khi tạo ra luồng giúp cho việc đọc thông tin nhị phân từ tập tin, chương trình bắt đầu đọc thông qua vòng lặp:

```
while (brFile.PeekChar() != -1)
{
    Console.Write("<{0}>", brFile.ReadInt32());
}
```

Ở đây có một vài sự khác nhỏ, phương thức PeekChar của lớp BinaryReader được sử dụng. Phương thức này sẽ lấy ký tự kế tiếp trong luồng. Nếu ký tự kế tiếp là cuối tập tin thì giá trị -1 được trả về. Ngược lại, thì ký tự kế tiếp được trả về Khi ký tự kế tiếp không phải ký tự cuối tập tin thì lệnh bên trong vòng lặp sẽ đọc một số integer từ đối tượng BinaryStream brFile.

Phương thức được sử dụng để đọc một số nguyên là ReadInt32, chúng ta sử dụng kiểu tên của Framework tốt hơn là kiểu do C# đưa ra. Nên nhớ rằng, tất cả những lớp từ Framework điều được gọi bởi ngôn ngữ C# và chúng không phải là một bộ phận của ngôn ngữ C#. Những lớp này còn được sử dụng tốt bởi những ngôn ngữ khác C#.

Ngoài ra lớp BinaryReader còn có những phương thức khác để thực hiện việc đọc các kiểu dữ liệu khác nhau. Những phương thức đọc này được sử dụng cùng với cách mà ReadInt32 được sử dụng trong chương trình. Bảng 12.4 sau liệt kê một số phương thức dùng để đọc các kiểu dữ liệu.

Phương thức	**Ý nghĩa**
Read	Đọc những ký tự và chuyển vị trí đọc sang vị trí tiếp theo. Phương thức này được nạp chồng gồm 3 phương thức.
ReadBoolean	Đọc một giá trị boolean từ luồng hiện thời và chuyển vị trí đọc sang một byte.
ReadByte	Đọc byte kế tiếp từ luồng hiện thời và chuyển vị trí đọc sang 1 byte.
ReadBytes	Đọc n byte từ luồng hiện thời sang một mảng byte và chuyển vị trí đọc sang n byte.
ReadChar	Đọc vị trí kế tiếp trong luồng hiện hành và chuyển vị trí đọc của luồng theo sau sử dụng mã hóa và ký tự xác định được đọc từ luồng.
ReadChars	Đọc n ký tự từ luồng hiện hành vào một mảng n ký tự. Và chuyển vị trí đọc của luồng theo sau sử dụng mã hóa và ký tự xác định được đọc từ luồng.
ReadDecimal	Đọc giá trị decimal và chuyển vị trí đọc sang 16 byte.
ReadDouble	Đọc giá trị thực 8 byte và chuyển vị trí đọc sang 8 byte.
ReadInt16	Đọc giá trị 2 byte integer có dấu và chuyển vị trí đọc sang 2 byte.
ReadInt32	Đọc giá trị 4 byte integer có dấu và chuyển vị trí đọc sang 4 byte.
ReadInt64	Đọc giá trị 8 byte integer có dấu và chuyển vị trí đọc sang 8 byte
ReadSByte	Đọc một signed byte từ luồng và chuyển vị trí đọc sang 1 byte.
ReadSingle	Đọc giá trị thực 4 byte từ luồng và chuyển vị trí đọc sang 4 byte.
ReadString	Đọc một chuỗi từ luồng. Chuỗi được cố định chiều dài trước. Và được mã hóa mỗi lần như là số nguyên 7 bit.
ReadUInt16	Đọc giá trị 2-byte unsigned integer từ luồng. Sử dụng mã hóa thứ tự nhỏ ở cuối (little endian encoding). Và chuyển vị trí hiện hành sang 2 byte.

ReadUInt64	Đọc 8-byte unsigned integer từ luồng hiện hành và chuyển sang 8 byte.

Bảng 12.4: Các phương thức đọc của BinaryReader.

Câu hỏi và trả lời

Câu hỏi 1: Các ngôn ngữ được hỗ trợ bởi .NET phải tuân thủ theo quy tắc nào không?

Trả lời 1: Như đã trình bày bên trên, các ngôn ngữ .NET phải tuân thủ theo quy định chung để có thể hoạt động trên nền của .NET. Những quy định này được gọi là Common Language Specification (CLS). CLS đưa ra những kiểu dữ liệu chung và các tập luật để thao tác trên kiểu dữ liệu này, CLS cho phép tạo ra một môi trường thực thi chung mà không cần quan tâm đến từng ngôn ngữ được sử dụng. Lợi ích của CLS là mã nguồn được viết thống nhất để quản lý, mã nguồn được viết trong ngôn ngữ này có thể được sử dụng bởi một ngôn ngữ khác.

Câu hỏi 2: Nếu muốn tìm hiểu về các lớp được cung cấp bởi .NET một cách chi tiết thì phải tìm ở đâu?

Trả lời 2: Để tìm hiểu chi tiết các lớp của .NET thì chúng ta có thể tìm trong thư viện trực tuyến của Microsoft có tên là MSDN Online, thư viện này chứa tất cả các thông tin liên quan đến .NET Framework mà người học cần quan tâm. Thư viện này thường xuyên được cập nhật và chứa những thông tin mới nhất về các phiên bản của .NET.

Câu hỏi thêm

Câu hỏi 1: Để truy xuất thời gian của đồng hồ hệ thống chúng ta phải dùng lớp nào?

Câu hỏi 2: Thông tin về máy tính có thể được truy xuất thông qua lớp nào?

Câu hỏi 3: Tham số dòng lệnh là gì? Làm thế nào để lấy được tham số dòng lệnh?

Câu hỏi 4: Lớp thao tác các phép toán học cơ bản? Chúng ta có thể tạo thể hiện của lớp này hay không?

Câu hỏi 5: Lớp thao tác tập tin File chứa trong namespace nào? Các thao tác chính được thực hiện trên tập tin?

Câu hỏi 6: Lớp nào cung cấp các thông tin về tập tin? Các phương thức chính của lớp này?

Câu hỏi 7: Luồng là gì? Phân biệt giữa tập tin và luồng?

Câu hỏi 8: Có mấy cách thức tạo tập tin? Cho biết thứ tự đọc của một tập tin?

Câu hỏi 9: Sự khác nhau giữa lớp File và FileInfo? Khi nào thì sử dụng lớp File tốt hơn là sử dụng FileInfo?

Câu hỏi 10: Khi tạo một tập tin mới trùng với tên của một tập tin cũ trong cùng một vị trí thư mục thì chuyện gì xảy ra?

Câu hỏi 11: Nếu muốn viết dữ liệu đã định dạng như là kiểu số thì dùng cách viết vào tập tin dạng nào?

Bài tập

Bài tập 1: Viết một chương trình minh họa việc truy xuất thông tin hệ thống của máy tính đang sử dụng. Thông tin này bao gồm: tên máy tính, hệ điều hành, bộ nhớ, đĩa cứng...

Bài tập 2: Viết chương trình minh họa một máy tính cá nhân cho phép thực hiện các phép toán cơ bản. Chương trình hiện ra một menu các lệnh và mỗi lệnh được gán cho một số: như công thì số 1, trừ số 2, nhân 3,... Cho phép người dùng chọn một lệnh thông qua nhập vào số tương ứng. Sau đó cho người dùng nhập vào từng toán hạng rồi thực hiện phép toán và cuối cùng in kết quả ra màn hình.

Bài tập 3: Viết chương trình cho phép xem thông tin về một tập tin. Chương trình cho người dùng nhập vào tên tập tin rồi sau đó lần lượt hiển thị các thông tin như: thuộc tính tập tin, ngày giờ tạo lập, kích thước tập tin...

Bài tập 4: Viết chương trình xem tập tin văn bản giống như lệnh type của DOS. Chương trình cho phép người dùng nhập tên tập tin thông qua tham số dòng lệnh. Nếu người dùng không nhập qua tham số dòng lệnh thì yêu cầu nhập vào.

Bài tập 5: Viết chương trình cho phép người dùng nhập vào một mảng số nguyên. Sau đó sắp xếp mảng này theo thứ tự tăng dần rồi lưu mảng vào một tập tin trên đĩa với dạng nhị phân.

Chương 13

XỬ LÝ NGOẠI LỆ

- **Phát sinh và bắt giữ ngoại lệ**
  - **Câu lệnh throw**
  - **Câu lệnh catch**
  - **Câu lệnh finally**
- **Những đối tượng ngoại lệ**
  - **Tạo riêng các ngoại lệ**
  - **Phát sinh lại ngoại lệ**
  - **Câu hỏi & bài tập**

Ngôn ngữ C# cũng giống như bất cứ ngôn ngữ hướng đối tượng khác, cho phép xử lý những lỗi và các điều kiện không bình thường với những ngoại lệ. Ngoại lệ là một đối tượng đóng gói những thông tin về sự cố của một chương trình không bình thường.

Một điều quan trọng để phân chia giữa bug, lỗi, và ngoại lệ. Một bug là một lỗi lập trình có thể được sửa chữa trước khi mã nguồn được chuyển giao. Những ngoại lệ thì không được bảo vệ và tương phản với những bug. Mặc dù một bug có thể là nguyên nhân sinh ra ngoại lệ, chúng ta cũng không dựa vào những ngoại lệ để xử lý những bug trong chương trình, tốt hơn là chúng ta nên sửa chữa những bug này.

Một lỗi có nguyên nhân là do phía hành động của người sử dụng. Ví dụ, người sử dụng nhập vào một số nhưng họ lại nhập vào ký tự chữ cái. Một lần nữa, lỗi có thể làm xuất hiện ngoại lệ, nhưng chúng ta có thể ngăn ngừa điều này bằng cách bắt giữ lỗi với mã hợp lệ. Những lỗi có thể được đoán trước và được ngăn ngừa.

Thậm chí nếu chúng ta xóa tất cả những bug và dự đoán tất cả các lỗi của người dùng, chúng ta cũng có thể gặp phải những vấn đề không mong đợi, như là xuất hiện trạng thái thiếu bộ nhớ (out of memory), thiếu tài nguyên hệ thống,... những nguyên nhân này có thể do các chương tránh khác cùng hoạt động ảnh hưởng đến. Chúng ta không thể ngăn ngừa các ngoại lệ này, nhưng chúng ta có thể xử lý chúng để chúng không thể làm tổn hại đến chương trình.

Khi một chương trình gặp một tình huống ngoại lệ, như là thiếu bộ nhớ thì nó sẽ tạo một ngoại lệ. Khi một ngoại lệ được tạo ra, việc thực thi của các chức năng hiện hành sẽ bị treo cho đến khi nào việc xử lý ngoại lệ tương ứng được tìm thấy.

Điều này có nghĩa rằng nếu chức năng hoạt động hiện hành không thực hiện việc xử lý ngoại lệ, thì chức năng này sẽ bị chấm dứt và hàm gọi sẽ nhận sự thay đổi đến việc xử lý ngoại lệ. Nếu hàm gọi này không thực hiện việc xử lý ngoại lệ, ngoại lệ sẽ được xử lý sớm bởi CLR, điều này dẫn đến chương trình của chúng ta sẽ kết thúc.

Một trình xử lý ngoại lệ là một khối lệnh chương trình được thiết kế xử lý các ngoại lệ mà chương trình phát sinh. Xử lý ngoại lệ được thực thi trong trong câu lệnh **catch**. Một cách lý tưởng thì nếu một ngoại lệ được bắt và được xử lý, thì chương trình có thể sửa chữa được vấn đề và tiếp tục thực hiện hoạt động. Thậm chí nếu chương trình không tiếp tục, bằng việc bắt giữ ngoại lệ chúng ta có cơ hội để in ra những thông điệp có ý nghĩa và kết thúc chương trình một cách rõ ràng.

Nếu đoạn chương trình của chúng ta thực hiện mà không quan tâm đến bất cứ ngoại lệ nào mà chúng ta có thể gặp (như khi giải phóng tài nguyên mà chương trình được cấp phát), chúng ta có thể đặt đoạn mã này trong khối **finally**, khi đó nó sẽ chắc chắn sẽ được thực hiện thậm chí ngay cả khi có một ngoại lệ xuất hiện.

Phát sinh và bắt giữ ngoại lệ

Trong ngôn ngữ C#, chúng ta chỉ có thể phát sinh (throw) những đối tượng các kiểu dữ liệu là System.Exception, hay những đối tượng được dẫn xuất từ kiểu dữ liệu này. Namespace System của CLR chứa một số các kiểu dữ liệu xử lý ngoại lệ mà chúng ta có thể sử dụng trong chương trình. Những kiểu dữ liệu ngoại lệ này bao gồm ArgumentNull-Exception, InValidCastException, và OverflowException, cũng như nhiều lớp khác nữa.

Câu lệnh throw

Để phát tín hiệu một sự không bình thường trong một lớp của ngôn ngữ C#, chúng ta phát sinh một ngoại lệ. Để làm được điều này, chúng ta sử dụng từ khóa **throw**. Dòng lệnh sau tạo ra một thể hiện mới của System.Exception và sau đó throw nó:

```
throw  new System.Exception();
```

Khi phát sinh ngoại lệ thì ngay tức khắc sẽ làm ngừng việc thực thi trong khi CLR sẽ tìm kiếm một trình xử lý ngoại lệ. Nếu một trình xử lý ngoại lệ không được tìm thấy trong phương thức hiện thời, thì CLR tiếp tục tìm trong phương thức gọi cho đến khi nào tìm thấy. Nếu CLR trả về lớp Main() mà không tìm thấy bất cứ trình xử lý ngoại lệ nào, thì nó sẽ kết thúc chương trình.

Ví dụ 13.1: Throw ngoại lệ.

```
---------------------------------------------------------------------------
namespace  Programming_CSharp
{
    using System;
    public  class Test
    {
```

```
        public static void Main()
        {
        Console.WriteLine("Enter Main....");
        Test  t = new Test();
        t.Func1();
        Console.WriteLine("Exit Main...");
        }
        public void Func1()
        {
        Console.WriteLine("Enter Func1...");
        Func2();
        Console.WriteLine("Exit  Func1...");
        }
        public void Func2()
        {
        Console.WriteLine("Enter Func2...");
        throw  new  System.Exception();
        Console.WriteLine("Exit Func2...");
        }
   }
}
```

Kết quả:

```
Enter  Main....
Enter  Func1...
Enter  Func2...
Exception occurred: System.Exception: An exception of type System.Exception
was throw.
   at Programming_CSharp.Test.Func2() in ... exception01.cs:line 26
   at Programming_CSharp.Test.Func1() in ... exception01.cs:line 20
   at Programming_CSharp.Test.Main() in ... exception01.cs:line 12
```

Ví dụ minh họa đơn giản này viết ra màn hình console thông tin khi nó nhập vào trong một hàm và chuẩn bị đi ra từ một hàm. Hàm Main() tạo thể hiện mới của kiểu Test và sau đó gọi hàm Func1(). Sau khi in thông điệp "*Enter Func1*", hàm Func1() này gọi hàm Func2(). Hàm Func2() in ra thông điệp đầu tiên và phát sinh một ngoại lệ kiểu System.Exception. Việc thực thi sẽ ngưng ngay tức khắc, và CLR sẽ tìm kiếm trình xử lý ngoại lệ trong hàm Func2(). Do không tìm thấy ở đây, CLR tiếp tục vào stack lấy hàm đã gọi trước tức là Func1 và tìm kiếm

trình xử lý ngoại lệ. Một lần nữa trong Func1 cũng không có đoạn xử lý ngoại lệ. Và CLR trả về hàm Main. Tại hàm Main cũng không có, nên CLR sẽ gọi trình mặc định xử lý ngoại lệ, việc này đơn giản là xuất ra một thông điệp lỗi.

Câu lệnh catch

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

Trong ví dụ 13.2 sau, câu lệnh **throw** được thực thi bên trong khối **try**, và một khối **catch** được sử dụng để công bố rằng một lỗi đã được xử lý.

Ví dụ 13.2: bắt giữ ngoại lệ.

```
---------------------------------------------------------------------------
namespace  Programming_CSharp
{
    using  System;
    public class Test
    {
            public static void Main()
            {
            Console.WriteLine("Enter Main...");
            Test  t = new Test();
            t.Func1();
            Console.WriteLine("Exit Main...");
            }
            public void Func1()
            {
            Console.WriteLine("Enter  Func1...");
            Func2();
            Console.WriteLine("Exit Func1...");
            }
            public void Func2()
            {
             Console.WriteLine("Enter  Func2...");
             try
             {
                Console.WriteLine("Entering  try block...");
                throw  new System.Exception();
                Console.WriteLine("Exiting try block...");
             }
```

```
            catch
            {
                Console.WriteLine("Exception caught and handled.");
            }
            Console.WriteLine("Exit Func2...");
           }
    }
}
```

Kết quả:

```
Enter Main...
Enter Func1...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...
```

Ví dụ 13.2 cũng tương tự như ví dụ minh họa trong 13.1 ngoại trừ chương trình thêm vào trong một khối **try**/**catch**. Thông thường chúng ta cũng co thể đặt khối **try** bao quanh những đoạn chương trình tiềm ẩn gây ra sự nguy hiểm, như là việc truy cập file, cấp phát bộ nhớ...

Theo sau khối **try** là câu lệnh **catch** tổng quát. Câu lệnh **catch** trong ví dụ này là tổng quát bởi vì chúng ta không xác định loại ngoại lệ nào mà chúng ta bắt giữ. Trong trường hợp tổng quát này thì khối **catch** này sẽ bắt giữ bất cứ ngoại lệ nào được phát sinh. Sử dụng câu lệnh **catch** để bắt giữ ngoại lệ xác định sẽ được thảo luận trong phần sau của chương.

Trong ví dụ 13.2 này, khối **catch** đơn giản là thông báo ra một ngoại lệ được bắt giữ và được xử lý. Trong ví dụ của thế giới thực, chúng ta có thể đưa hành động đúng để sửa chữa vấn đề mà gây ra sự ngoại lệ. Ví dụ, nếu người sử dụng đang cố mở một tập tin có thuộc tính chỉ đọc, chúng ta có thể gọi một phương thức cho phép người dùng thay đổi thuộc tính của tập tin. Nếu chương trình thực hiện thiếu bộ nhớ, chúng ta có thể phát sinh cho người dùng cơ hội để đóng bớt các ứng dụng khác lại. Thậm chí trong trường hợp xấu nhất ta không khắc phục được thì khối **catch** này có thể in ra thông điệp lỗi để người dùng biết.

Thử kiểm tra kỹ lại chương trình 13.2 trên, chúng ta sẽ thấy xuất hiện đoạn mã đi vào từng hàm như Main(), Func1(), Func2(), và cả khối **try**. Chúng ta không bao giờ thấy nó thoát khỏi lệnh **try** (tức là in ra thông báo "*Exiting try block...*", hay thực hiện lệnh này), mặc dù nó vẫn thoát ra theo thứ tự Func2(), Func1(), và Main(). Chuyện gì xảy ra?

Khi một ngoại lệ được phát sinh, việc thi hành ngay lập tức sẽ bị tạm dừng và việc thi hành sẽ được chuyển qua khối lệnh **catch**. Nó không bao giờ trả về luồng thực hiện ban đầu, tức là các lệnh sau khi phát ra ngoại lệ trong khối **try** không được thực hiện. Trong trường hợp này chúng ta sẽ không bao giờ nhận được thông báo "*Exiting try block....*". Khối lệnh **catch** xử lý lỗi và sau đó chuyển việc thực thi chương trình đến các lệnh tiếp sau khối **catch**.

Ở đây không có việc quay lại cuộc gọi hàm trước trong stack. Ngoại lệ bây giờ được xử lý, không có vấn đề gì xảy ra, và chương trình tiếp tục hoạt động bình thường. Điều này trở nên rõ ràng hơn nếu chúng ta di chuyển khối **try**/**catch** lên hàm Func1 như trong ví dụ minh họa 13.3 bên dưới.

Ví dụ 13.3: Catch trong hàm gọi.

```
---------------------------------------------------------------------------
namespace  Programming_CSharp
{
    using  System;
    public class Test
    {
            public static void Main()
            {
            Console.WriteLine("Enter Main...");
            Test  t = new Test();
            t.Func1();
            Console.WriteLine("Exit Main...");
            }
            public void Func1()
            {
            Console.WriteLine("Enter  Func1...");
             try
             {
                Console.WriteLine("Entering  try block...");
                Func2();
                Console.WriteLine("Exiting try block...");
             }
             catch
             {
                Console.WriteLine("Exception caught and handled.");
             }

            Console.WriteLine("Exit Func1...");
```

```
        }
        public void Func2()
        {
         Console.WriteLine("Enter  Func2...");
         throw new System.Exception();
         Console.WriteLine("Exit Func2...");
        }
   }
}
```

Kết quả:

```
Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Exception caught and handled.
Exit Func1...
Exit Main...
```

Lúc này ngoại lệ không được xử lý bên trong hàm Func2(), mà nó được xử lý bên trong hàm Func1(). Khi hàm Func2() được gọi, nó in câu lệnh thông báo vào hàm rồi phát sinh một ngoại lệ. Việc thực hiện chương trình bị ngưng, CLR tìm kiếm phần xử lý ngoại lệ, nhưng trong hàm này không có và CLR vào stack lấy hàm gọi trong trường hợp này là Func1(). Câu lệnh **catch** sẽ được gọi, và việc thực thi tiếp tục thực hiện bình thường sau câu lệnh **catch**.

Hãy chắc chắn rằng chúng ta đã hiểu rõ tại sao câu lệnh "*Exiting try block*" và "*Exit Func2*" không được in ra. Chúng ta có thể dùng cách cũ để kiểm tra việc này bằng cách dùng chương trình debug cho chương trình chạy từng bước để tìm hiểu rõ hơn.

❖ Tạo một khối **catch** xác định:

Cho đến bây giờ chúng ta chỉ dùng khối **catch** tổng quát, tức là với bất cứ ngoại lệ nào cũng được. Tuy nhiên chúng ta có thể tạo ra khối **catch** xác định để xử lý chỉ một vài các ngoại lệ chứ không phải toàn bộ ngoại lệ, dựa trên kiểu của ngoại lệ phát sinh. Ví dụ 13.4 minh họa cách xác định loại ngoại lệ mà chúng ta xử lý.

Ví dụ13.4: Xác định ngoại lệ để bắt.

```
namespace  Programming_CSharp
{
        using System;
        public  class  Test
```

```
{
        public  static  void  Main()
        {
    Test  t = new Test();
    t.TestFunc();
        }
        // ta thử chia hai phần xử lý ngoại lệ riêng
        public  void  TestFunc()
        {
    try
    {
            double  a = 5;
            double  b = 0;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
    }
    catch (System.DivideByZeroException)
    {
            Console.WriteLine("DivideByZeroException caught!");
    }
    catch (System.ArithmeticException)
    {
            Console.WriteLine("ArithmeticException caught!");
    }
    catch
    {
            Console.WriteLine("Unknown exception caught");
    }
        }
        // thực hiện phép chia hợp lệ
        public double  DoDivide(double a, double b)
        {
    if ( b == 0)
        throw  new  System.DivideByZeroException();
    if ( a == 0)
        throw  new  System.ArithmeticException();
    return  a/b;
        }
}
```

.....

}

Kết quả:

```
DivideByZeroException caught!
```

Trong ví dụ này, phương thức DoDivide() sẽ không cho phép chúng ta chia cho zero bởi một số khác, và cũng không cho phép chia số zero. Nó sẽ phát sinh một đối tượng của DivideByzeroException nếu chúng ta thực hiện chia với zero. Trong toán học việc lấy zero chia cho một số khác là được phép, nhưng trong ví dụ minh họa của chúng ta không cho phép thực hiện việc này, nếu thực hiện sẽ phát sinh ra một ngoại lệ ArithmeticException.

Khi một ngoại lệ được phát sinh, CLR sẽ kiểm tra mỗi khối xử lý ngoại lệ theo thứ tự và sẽ lấy khối đầu tiên thích hợp. Khi chúng ta thực hiện với a=5 và b=7 thì kết quả như sau:

```
5 / 7 = 0.7142857142857143
```

Như chúng ta mong muốn, không có ngoại lệ được phát sinh. Tuy nhiên, khi chúng ta thay đổi giá trị của a là 0, thì kết quả là:

```
ArithmeticException caught!
```

Ngoại lệ được phát sinh, và CLR sẽ kiểm tra ngoại lệ đầu tiên: DivideByZeroException. Bởi vì không phù hợp, nên nó sẽ tiếp tục đi tìm và khối xử lý ArithmeticException được chọn.

Cuối cùng, giả sử chúng ta thay đổi giá trị của b là 0. Khi thực hiện điều này sẽ dẫn đến ngoại lệ DivideByZeroException.

Ghi chú: Chúng ta phải cẩn thận thứ tự của câu lệnh **catch**, bởi vì DivideByZeroException được dẫn xuất từ ArithmeticException. Nếu chúng ta đảo thứ tự của câu lệnh **catch**, thì ngoại lệ DivideByZeroException sẽ được phù hợp với khối xử lý ngoại lệ ArithmeticException. Và việc xử lý ngoại lệ sẽ không bao giờ được giao cho khối xử lý DivideByZeroException. Thật vậy, nếu thứ tự này được đảo, nó sẽ không cho phép bất cứ ngoại lệ nào được xử lý bởi khối xử lý ngoại lệ DivideByZeroException. Trình biên dịch sẽ nhận ra rằng DivideByZeroException không được thực hiện bất cứ khi nào và nó sẽ thông báo một lỗi biên dịch.

Chúng ta có thể phân phối câu lệnh **try**/ **catch**, bằng cách bắt giữ những ngoại lệ xác định trong một hàm và nhiều ngoại lệ tổng quát trong nhiều hàm. Mục đích của thực hiện này là đưa ra các thiết kế đúng. Giả sử chúng ta có phương thức A, phương thức này gọi một phương thức khác tên là phương thức B, đến lượt mình phương thức B gọi phương thức C. Và phương thức C tiếp tục gọi phương thức D, cuối cùng phương thức D gọi phương thức E. Phương thức E ở mức độ sâu nhất trong chương trình của chúng ta, phương thức A, B ở mức độ cao hơn. Nếu chúng ta đoán trước phương thức E có thể phát sinh ra ngoại lệ, chúng ta có thể tạo ra khối **try**/**catch** để bắt giữ những ngoại lệ này ở chỗ gần nơi phát sinh ra ngoại lệ nhất. Chúng ta cũng có thể tạo ra nhiều khối xử lý ngoại lệ chung ở trong đoạn chương trình ở mức cao trong trường hợp những ngoại lệ không đoán trước được.

Câu lệnh finally

Trong một số tình huống, việc phát sinh ngoại lệ và unwind stack có thể tạo ra một số vấn đề. Ví dụ như nếu chúng ta mở một tập tin hay trường hợp khác là xác nhận một tài nguyên, chúng ta có thể cần thiết một cơ hội để đóng một tập tin hay là giải phóng bộ nhớ đệm mà chương trình đã chiếm giữ trước đó.

✍*Ghi chú*: Trong ngôn ngữ C#, vấn đề này ít xảy ra hơn do cơ chế thu dọn tự động của C# ngăn ngừa những ngoại lệ phát sinh từ việc thiếu bộ nhớ.

Tuy nhiên, có một số hành động mà chúng ta cần phải quan tâm bất cứ khi nào một ngoại lệ được phát sinh ra, như việc đóng một tập tin, chúng ta có hai chiến lược để lựa chọn thực hiện. Một hướng tiếp cận là đưa hành động nguy hiểm vào trong khối **try** và sau đó thực hiện việc đóng tập tin trong cả hai khối **catch** và **try**. Tuy nhiên, điều này gây ra đoạn chương trình không được đẹp do sử dụng trùng lắp lệnh. Ngôn ngữ C# cung cấp một sự thay thế tốt hơn trong khối **finally**.

Đoạn chương trình bên trong khối **catch** được đảm bảo thực thi mà không quan tâm đến việc khi nào thì một ngoại lệ được phát sinh. Phương thức TestFunc() trong ví dụ 13.5 minh họa việc mở một tập tin như là hành động đầu tiên của nó, sau đó phương thức thực hiện một vài các phép toán toán học, và sau đó là tập tin được đóng. Có thể trong quá trình mở tập tin cho đến khi đóng tập tin chương trình phát sinh ra một ngoại lệ. Nếu xuất hiện ngoại lệ, và khi đó tập tin vẫn còn mở. Người phát triển biết rằng không có chuyện gì xảy ra, và cuối của phương thức này thì tập tin sẽ được đóng. Do chức năng đóng tập tin được di chuyển vào trong khối **finally**, ở đây nó sẽ được thực thi mà không cần quan tâm đến việc có phát sinh hay không một ngoại lệ trong chương trình.

🗗 *Ví dụ 13.5: Sử dụng khối finally.*

```
---------------------------------------------------------------------------
namespace  Programming_CSharp
{
    using System;
    public class Test
    {
            public static void Main()
            {
            Test  t = new  Test();
            t.TestFunc();
            }
            // chia hai số và xử lý ngoại lệ nếu có
            public void TestFunc()
            {
            try
```

```
        {
                Console.WriteLine("Open file here");
                double  a = 5;
                double  b = 0;
                Console.WriteLine("{0} /{1} = {2}", a, b, DoDivide(a,b));
                Console.WriteLine("This line may or not print");
        }
        catch (System.DivideByZeroException)
        {
                Console.WriteLine("DivideByZeroException caught!");
        }
        catch
        {
                Console.WriteLine("Unknown exception caught");
        }
        finally
        {
                Console.WriteLine("Close file here.");
        }
        }
        // thực hiện chia nếu hợp lệ
        public double DoDivide(double a, double b)
        {
         if ( b == 0)
         {
            throw new System.DivideByZeroException();
         }
         if ( a == 0)
         {
            throw new System.ArithmeticException();
         }
         return a/b;
        }
    }
}
```

Kết quả:

```
Open file here
```

```
DivideByZeroException caught!
Close file here.
Kết quả trong trường hợp b = 12
Open file here
5/ 12 = 0.416666666666
Close file here
```

Trong ví dụ này một khối **catch** được loại bỏ và thêm vào khối **finally**. Bất cứ khi một ngoại lệ có được phát sinh ra hay không thì khối lệnh bên trong **finally** cũng được thực thi. Do vậy nên trong cả hai trường hợp ta cũng thấy xuất hiện thông điệp "*Close file here*".

Những đối tượng ngoại lệ

Cho đến lúc này thì chúng ta có thể sử dụng tốt các ngoại lệ cũng như cách xử lý khắc phục các ngoại lệ này. Trong phần này chúng ta sẽ tiến hành việc tìm hiểu các đối tượng được xây dựng cho việc xử lý ngoại lệ. Đối tượng System.Exception cung cấp một số các phương thức và thuộc tính hữu dụng. Thuộc tính Message cung cấp thông tin về ngoại lệ, như là lý do tại sao ngoại lệ được phát sinh. Thuộc tính Message là thuộc tính chỉ đọc, đoạn chương trình phát sinh ngoại lệ có thể thiết lập thuộc tính Message như là một đối mục cho bộ khởi dựng của ngoại lệ. Thuộc tính HelpLink cung cấp một liên kết để trợ giúp cho các tập tin liên quan đến các ngoại lệ. Đây là thuộc tính chỉ đọc. Thuộc tính StackTrace cũng là thuộc tính chỉ đọc và được thiết lập bởi CLR. Trong ví dụ 13.6 thuộc tính Exception.HelpLink được thiết lập và truy cập để cung cấp thông tin cho người sử dụng về ngoại lệ DivideBy-ZeroException. Thuộc tính StackTrace của ngoại lệ được sử dụng để cung cấp thông tin stack cho câu lệnh lỗi. Một thông tin stack cung cấp hàng loạt các cuộc gọi stack của phương thức gọi mà dẫn đến những ngoại lệ được phát sinh.

Ví dụ 13.6: Làm việc với đối tượng ngoại lệ.

```
namespace Programming_CSharp
   {
      using System;
      public class Test
      {
         public static void Main()
         {
            Test  t = new Test();
            t.TestFunc();
         }
         // chia hai số và xử lý ngoại lệ
```

```
        public void TestFunc()
        {
            try
            {
                Console.WriteLine("Open file here");
                double a = 12;
                double b = 0;
                Console.WriteLine("{0} /{1} = {2}", a, b, DoDivide(a,b));
                Console.WriteLine("This line may or not print");
            }
            catch  (System.DivideByZeroException e)
            {
                Console.WriteLine("\nDivideByZeroException! Msg: {0}", e.Message);
                Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
                Console.WriteLine("\nHere's a stack trace: {0}\n", e.StackTrace);
            }
            catch
            {
                Console.WriteLine("Unknown exception caught");
            }
        }
        // thực hiện phép chia hợp lệ
        public double DoDivide( double a, double b)
        {
         if ( b == 0)
         {
            DivideByZeroException e = new DivideByZeroException();
            e.HelpLink = "http://www.hcmunc.edu.vn";
            throw  e;
        }
        if ( a == 0)
        {
            throw new ArithmeticException();
        }
        return a/b;
        }
        }
}
```

Kết quả:

```
Open file here
DivideByZeroException Msg: Attempted to divide by zero
HelpLink: http://www.hcmuns.edu.vn
Here's a stack trace:
at Programming_CSharp.Test.DoDivide(Double c, Double b)
in c:\...exception06.cs: line 56
at Programming_CSharp.Test.TestFunc() in ...exception06.cs: line 22.
Close file here
```

Trong đoạn kết quả trên, danh sách trace của stack được hiển thị theo thứ tự ngược lại thứ tự gọi. Nó hiển thị một lỗi trong phương thức DoDivde(), phương thức này được gọi từ phương thức TestFunc(). Khi các phương thức gọi lồng nhau nhiều cấp, thông tin stack có thể giúp chúng ta hiểu thứ tự của các phương thức được gọi.

Trong ví dụ này, hơn là việc đơn giản phát sinh một DidiveByZeroException, chúng ta tạo một thể hện mới của ngoại lệ:

```
DivideByZeroException  e = new DivideByZeroException();
```

Chúng ta không truyền vào thông điệp của chúng ta, nên thông điệp mặc định sẽ được in ra:

```
DivideByZeroException! Msg: Attemped to divide by zero.
```

Ở đây chúng ta có thể bổ sung như dòng lệnh bên dưới để truyền vào thông điệp của chúng ta tùy chọn như sau:

```
new DivideByZeroException("You tried to divide by zero which is not meaningful");
```

Trước khi phát sinh ra ngoại lệ, chúng ta thiết lập thuộc tính HelpLink như sau:

```
e.HelpLink = "http://www.hcmunc.edu.vn";
```

Khi ngoại lệ được bắt giữ, chương trình sẽ in thông điệp và HelpLink ra màn hình:

```
    catch (System.DivideByZeroException e)
    {
        Console.WriteLine("\nDivideByZeroException! Msg: {0}", e.Message);
        Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
}
```

Việc làm này cho phép chúng ta cung cấp những thông tin hữu ích cho người sử dụng. Thêm vào đó thông tin stack cũng được đưa ra bằng cách sử dụng thuộc tính StackTrace của đối tượng ngoại lệ:

```
Console.WriteLine("\n Here's a stack trace: {0}\n", e.StackTrace);
```

Kết quả là các vết trong stack sẽ được xuất ra:

```
Here's a stack trace:
at Programming_CSharp.Test.DoDivide(Double c, Double b)
```

```
in c:\...exception06.cs: line 56
at Programming_CSharp.Test.TestFunc() in ...exception06.cs: line 22.
```

Lưu ý rằng, phần đường dẫn được viết tắt, do đó kết quả của bạn có thể hơi khác một tí.

Bảng 13.1 sau mô tả một số các lớp ngoại lệ chung được khai báo bên trong namespace System.

CÁC LỚP NGOẠI LỆ	
Tên ngoại lệ	**Mô tả**
MethodAccessException	Lỗi truy cập, do truy cập đến thành viên hay phương thức không được truy cập
ArgumentException	Lỗi tham số đối mục
ArgumentNullException	Đối mục Null, phương thức được truyền đối mục null không được chấp nhận
ArithmeticException	Lỗi liên quan đến các phép toán
ArrayTypeMismatchException	Kiểu mảng không hợp, khi cố lưu trữ kiểu không thích hợp vào mảng
DivideByZeroException	Lỗi chia zero
FormatException	Định dạng không chính xác một đối mục nào đó
IndexOutOfRangeException	Chỉ số truy cập mảng không hợp lệ, dùng nhỏ hơn chỉ số nhỏ nhất hay lớn hơn chỉ số lớn nhất của mảng
InvalidCastException	Phép gán không hợp lệ
MulticastNotSupportedException	Multicast không được hỗ trợ, do việc kết hợp hai delegate không đúng
NotFiniteNumberException	Không phải số hữu hạn, số không hợp lệ
NotSupportedException	Phương thức không hỗ trợ, khi gọi một phương thức không tồn tại bên trong lớp.
NullReferenceException	Tham chiếu null không hợp lệ.
OutOfMemoryException	Out of memory
OverflowException	Lỗi tràn phép toán
StackOverflowException	Tràn stack
TypeInitializationException	Kiểu khởi tạo sai, khi bộ khởi dựng tĩnh có lỗi.

Bảng 13.1 : Các ngoại lệ thường xuất hiện.

Tạo riêng các ngoại lệ

CLR cung cấp những kiểu dữ liệu ngoại lệ cơ bản, trong ví dụ trước chúng ta đã tạo một vài các kiểu ngoại lệ riêng. Thông thường chúng ta cần thiết phải cung cấp các thông tin mở rộng cho khối **catch** khi một ngoại lệ được phát sinh. Tuy nhiên, có những lúc chúng ta

muốn cung cấp nhiều thông tin mở rộng hay là các khả năng đặc biệt cần thiết trong ngoại lệ mà chúng ta tạo ra. Chúng ta dễ dàng tạo ra các ngoại lệ riêng, hay còn gọi là các ngoại lệ tùy chọn (custom exception), điều bắt buộc với các ngoại lệ này là chúng phải được dẫn xuất từ System.ApplicationException. Ví dụ 13.7 sau minh họa việc tạo một ngoại lệ riêng.

Ví dụ: Tạo một ngoại lệ riêng.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
{
    using System;
    // tạo ngoại lệ riêng
    public class MyCustomException : System.ApplicationException
    {
            public MyCustomException( string message): base(message)
            {

            }
    }
    public class Test
    {
            public static void Main()
            {
            Test  t = new Test();
            t.TestFunc();
            }
            // chia hai số và xử lý ngoại lệ
            public void TestFunc()
            {
                try
                {
                    Console.WriteLine("Open file here");
                    double a = 0;
                    double b = 5;
                    Console.WriteLine("{0} /{1} = {2}", a, b, DoDivide(a,b));
                    Console.WriteLine("This line may or not print");
                }
                catch  (System.DivideByZeroException e)
                {
                    Console.WriteLine("\nDivideByZeroException! Msg: {0}", e.Message);
```

```
                Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
            }
            catch (MyCustomException  e)
            {
                Console.WriteLine("\nMyCustomException! Msg: {0}", e.Message);
                Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
            }
            catch
            {
                Console.WriteLine("Unknown excepiton caught");
            }
            finally
            {
                Console.WriteLine("Close file here.");
            }
        }
        // thực hiện phép chia hợp lệ
        public double DoDivide( double a, double b)
        {
         if ( b == 0)
        {
            DivideByZeroException e = new DivideByZeroException();
            e.HelpLink = "http://www.hcmunc.edu.vn";
            throw  e;
        }
        if ( a == 0)
        {
            MyCustomException    e  =  new  MyCustomException("Can't  have  zero
divisor");
            e.HelpLink = "http://www.hcmuns.edu.vn";
            throw e;
        }
        return a/b;
        }
    }
}
```

Lớp MyCustomException được dẫn xuất từ System.ApplicationException và lớp này không có thực thi hay khai báo gì ngoài một hàm khởi dựng. Hàm khởi dựng này lấy tham số là một chuỗi và truyền cho lớp cơ sở. Trong trường hợp này, lợi ích của việc tạo ra ngoại lệ là làm nổi bật điều mà chương trình muốn minh họa, tức là không cho phép số chia là zero. Sử dụng ngoại lệ ArithmeticException thì tốt hơn là ngoại lệ chúng ta tạo ra. Nhưng nó có thể làm nhầm lẫn cho những người lập trình khác vì phép chia với số chia là zero không phải là lỗi số học.

Phát sinh lại ngoại lệ

Giả sử chúng ta muốn khối **catch** thực hiện một vài hành động đúng nào đó rồi sau đó phát sinh lại ngoại lệ ra bên ngoài khối **catch** (trong một hàm gọi). Chúng ta được phép phát sinh lại cùng một ngoại lệ hay phát sinh lại các ngoại lệ khác. Nếu phát sinh ra ngoại lệ khác, chúng ta có thể phải nhúng ngoại lệ ban đầu vào bên trong ngoại lệ mới để phương thức gọi có thể hiểu được lai lịch và nguồn gốc của ngoại lệ. Thuộc tính InnerException của ngoại lệ mới cho phép truy cập ngoại lệ ban đầu.

Bởi vì InnerException cũng là một ngoại lệ, nên nó cũng có một ngoại lệ bên trong. Do vậy, toàn bộ dây chuyền ngoại lệ là một sự đóng tổ (nest) của một ngoại lệ này với một ngoại lệ khác. Giống như là con lật đật, mỗi con chứa trong một con và đến lượt con bên trong lại chứa...

Ví dụ 13.8: Phát sinh lại ngoại lệ & ngoại lệ inner.

```
---------------------------------------------------------------------------
namespace Programming_CSharp
{
    using System;
    // tạo ngoại lệ riêng
    public class MyCustomException : System.Exception
    {
            public MyCustomException( string message, Exception inner):
                base(message, inner)
            {

            }
    }
    public class Test
    {
            public static void Main()
            {
                Test  t = new Test();
```

```
            t.TestFunc();
        }
        // chia hai số và xử lý ngoại lệ
        public void TestFunc()
        {
            try
            {
                DangerousFunc1();
            }
            catch  (MyCustomException e)
            {
                Console.WriteLine("\n{0}", e.Message);
                Console.WriteLine("Retrieving exception history...");
                Exception  inner = e.InnerException;
                while ( inner != null)
                {
                    Console.WriteLine("{0}", inner.Message);
                    inner = inner.InnerException;
                }
            }
    }
        public void DangerousFunc1()
        {
            try
            {
                DangerousFunc2();
            }
            catch (System.Exception e)
            {
                MyCustomException ex = new
                MyCustomException("E3 – Custom Exception Situation", e);
                throw ex;
            }
        }
        public void DangerousFunc2()
        {
            try
            {
```

```
                DangerousFunc3();
            }
            catch (System.DivideByZeroException  e)
            {
                Exception  ex = new Exception("E2 - Func2 caught divide by zero", e);
                throw ex;
            }
        }
        public void DangerousFunc3()
        {
            try
            {
                DangerousFunc4();
            }
            catch (System.ArithmeticException)
            {
                throw;
            }
            catch (System.Exception)
            {
                Console.WriteLine("Exception handled here.");
            }
        }
        public void DangerousFunc4()
        {
            throw new DivideByZeroException("E1 – DivideByZero Exception");
        }
    }
}
```

Kết quả:

```
E3 – Custom Exception Situation!
Retrieving exception history...
E2 - Func2 caught divide by zero
E1 – DivideByZeroException
```

Để hiểu rõ hơn ta có thể dùng trình debugger để chạy từng bước chương trình khi đó ta sẽ hiểu rõ từng bước thực thi cũng như việc phát sinh các ngoại lệ.

Chương trình bắt đầu với việc gọi hàm DangerousFunc1() trong khối **try**:

```
try
{
        DangerousFunc1();
}
```

DangerousFunc1() gọi DangerousFunc2(), DangerousFunc2() lại gọi DangerousFunc3(), và cuối cùng DangerousFunc3() gọi DangerousFunc4(). Tất cả việc gọi này điều nằm trong khối **try**. Cuối cùng, DangerousFunc4() phát sinh ra ngoại lệ DivideByzeroException. Ngoại lệ này bình thường có chứa thông điệp bên trong nó, nhưng ở đây chúng ta tự do dùng thông điệp mới. Để dễ theo dõi chúng ta đưa vào các chuỗi xác nhận tuần tự các sự kiện diễn ra.

Ngoại lệ được phát sinh trong DangerousFunc4() và nó được bắt trong khối **catch** trong hàm DangerousFunc3(). Khối **catch** trong DangerousFunc3() sẽ bắt các ngoại lệ Arithmetic-Exception (như là DivideByZeroException), nó không thực hiện hành động nào mà chỉ đơn giản là phát sinh lại ngoại lệ:

```
catch ( System.ArithmeticException)
{
        throw;
}
```

Cú pháp để thực hiện phát sinh lại cùng một ngoại lệ mà không có bất cứ bổ sung hay hiệu chỉnh nào là : **throw**.

Do vậy ngoại lệ được phát sinh cho DangerousFunc2(), khối **catch** trong DangerousFunc2() thực hiện một vài hành động và tiếp tục phát sinh một ngoại lệ có kiểu mới. Trong hàm khởi dựng của ngoại lệ mới, DangerousFunc2() truyền một chuỗi thông điệp mới ("*E2 - Func2 caught divide by zero*") và ngoại lệ ban đầu. Do vậy ngoại lệ ban đầu (E1) trở thành ngoại lệ bên trong của ngoại lệ mới (E2). Sau đó hàm DangerousFunc2() phát sinh ngoại lệ này (E2) cho hàm DangerousFunc1().

DangerousFunc1() bắt giữ ngoại lệ này, làm một số công việc và tạo ra một ngoại lệ mới có kiểu là MyCustomException, truyền vào hàm khởi dựng của ngoại lệ mới một chuỗi mới ("*E3 – Custom Exception Situation*!") và ngoại lệ được bắt giữ (E2). Chúng ta nên nhớ rằng ngoại lệ được bắt giữ là ngoại lệ có chứa ngoại lệ DivideByZeroException (E1) bên trong nó. Tại thời điểm này, chúng ta có một ngoại lệ kiểu MyCustomException (E3), ngoại lệ này chứa bên trong một ngoại lệ kiểu Exception (E2), và đến lượt nó chứa một ngoại lệ kiểu DivideByZeroException (E1) bên trong. Sau cùng ngoại lệ được phát sinh cho hàm TestFunc;

Khi khối **catch** của TestFunc thực hiện nó sẽ in ra thông điệp của ngoại lệ :

```
E3 – Custom Exception Situation!
```

sau đó từng ngoại lệ bên trong sẽ được lấy ra thông qua vòng lặp **while**:

```
while ( inner != null)
{
```

```
        Console.WriteLine("{0}", inner.Message);
        inner = inner.InnerException;
    }
```

Kết quả là chuỗi các ngoại lệ được phát sinh và được bắt giữ:

```
Retrieving exception history...
E2 - Func2 caught divide by zero
E1 – DivideByZero Exception
```

Câu hỏi và trả lời

Câu hỏi 1: Việc sử dụng catch không có tham số có vẻ như có nhiều sức mạnh do chúng bắt giữa tất cả các ngoại lệ. Tại sao chúng ta không luôn luôn sử dụng câu lệnh catch không có tham số để bắt các lỗi?

Trả lời 1: Mặc dù sử dụng catch duy nhất có rất nhiều sức mạnh, nhưng nó cũng làm mất rất nhiều thông tin quan trọng về ngoại lệ được phát sinh. Khi đó chúng ta sẽ không biết chính xác loại ngoại lệ xảy ra và khó có thể bảo trì cũng như khắc phục những ngoại lệ sau này. Về phía người dùng cũng vậy. Nếu chương trình gặp ngoại lệ mà không có thông báo rõ ràng cho người dùng thì có thể làm cho họ hoang mang, và có thể đổ lỗi cho chương trình của chúng ta không tốt ngay cả những lỗi không phải do ta. Ví dụ như lỗi hết tài nguyên bộ nhớ do người dùng sử dụng quá nhiều chương trình hoạt động cùng lúc. Tóm lại là chúng ta nên sử dụng catch với những tham số chi tiết để thực hiện tốt việc quản lý các ngoại lệ được phát sinh.

Câu hỏi 2: Có phải tất cả những ngoại lệ được đối xử một cách bình đẳng?

Trả lời 2: Không phải, có hai loại ngoại lệ, ngoại lệ hệ thống và ngoại lệ của chương trình ứng dụng. Ngoại lệ của chương trình ứng dụng thì sẽ không kết thúc chương trình. Còn ngoại lệ hệ thống thì sẽ kết thúc chương trình. Nói chung đó là những ngoại lệ xuất hiện trước đây. Hiện nay thì người ta chia ra nhiều mức độ ngoại lệ và tùy theo từng mức độ của ngoại lệ mà chương trình của chúng ta sẽ được nhận những ứng xử khác nhau. Để biết thêm chi tiết chúng ta có thể đọc thêm trong tài liệu .NET Framework về xử lý ngoại lệ.

Câu hỏi 3: Như câu trả lời bên trên tại sao tôi phải tìm hiểu nhiều về các ngoại lệ và cách thức xử lý các ngoại lệ khi chúng được phát sinh?

Trả lời 3: Việc xây dựng một chương trình ứng dụng là hết sức phức tạp, chương trình luôn tiếm ẩn những yếu tố không ổn định và có thể phát sinh các ngoại lệ dẫn đến những lỗi không mong muốn. Việc thực hiện bắt giữ các ngoại lệ là hết sức cần thiết trong chương trình, nó cho phép chúng ta xây dựng được chương trình hoàn thiện hơn và xử lý các thông điệp ngoại lệ tốt hơn. Tìm hiểu những ngoại lệ đem đến cho chúng ta nhiều kinh nghiệm trong việc xây dựng các chương trình phức tạp hơn.

Câu hỏi thêm

Câu hỏi 1: Hãy cho biết các từ khóa được sử dụng để xử lý ngoại lệ?

Câu hỏi 2: Phân biệt giữa lỗi và ngoại lệ?

Câu hỏi 3: Khi thực hiện việc bắt giữ các ngoại lệ. Nếu có nhiều mức bắt giữ ngoại lệ thì chúng ta sẽ thực hiện mức nào. Từ chi tiết đến tổng quát, hay từ tổng quát đến chi tiết?

Câu hỏi 4: Ý nghĩa của từ khóa finally trong việc xử lý ngoại lệ?

Câu hỏi 5: Câu lệnh nào được dùng để phát sinh ngoại lệ?

Câu hỏi 6: Loại nào sau đây nên được xử lý theo ngoại lệ và loại nào thì nên được xử lý bởi các mã lệnh thông thường?

a. Giá trị nhập vào của người dùng không nằm trong mức cho phép.

b. Tập tin không được viết mà thực hiện viết.

c. Đối mục truyền vào cho phương thức chứa giá trị không hợp lệ.

d. Đối mục truyền vào cho phương thức chứa kiểu không hợp lệ.

Câu hỏi 7: Nguyên nhân nào dẫn đến phát sinh ngoại lệ?

Câu hỏi 8: Khi nào thì ngoại lệ xuất hiện?

a. Trong khi tạo mã nguồn

b. Trong khi biên dịch

c. Trong khi thực thi chương trình

d. Khi yêu cầu được đưa ta bởi người dùng cuối.

Câu hỏi 9: Khi nào thì khối lệnh trong finally được thực hiện?

Câu hỏi 10: Trong namespace nào chức các lớp liên quan đến việc xử lý các ngoại lệ? Hãy cho biết một số lớp xử lý ngoại lệ quan trọng trong namespace này?

Bài tập

Bài tập 1: Hãy viết đoạn lệnh để thực hiện việc bắt giữa ngoại lệ liên quan đến câu lệnh sau đây:

```
Ketqua = Sothu1 / Sothu2;
```

Bài tập 2: Chương trình sau đây có vấn đề. Hãy xác định vấn đề có thể phát sinh ngoại lệ khi chạy chương trình. Và viết lại chương trình hoàn chỉnh gồm các lệnh xử lý ngoại lệ:

```
-------------------------------------------------------------------------
using System;
public class Tester
{
    public static void Main()
    {
        uint so1=0;
        int so2, so3;
        so2 = -10;
        so3 = 0;
        // tính giá trị lại
```

```
        so1 -= 5;
        so2 = 5/so3;
        // xuất kết quả
        Console.WriteLine("So 1: {0}, So 2:{1}", so1, so2);
    }
}
```

Bài tập 3: Chương trình sau đây có thể dẫn đến ngoại lệ hay không? Nếu có thì hãy cho biết ngoại lệ có thể được phát sinh. Hãy viết lại chương trình hoàn chỉnh có xử lý các ngoại lệ bằng cách đưa ra thông điệp về ngoại lệ được phát sinh.

```
using System;
using System.IO;
public class Tester
{
    public static void Main()
    {
        string fname = "test3.txt";
        string buffer;
        StreamReader sReader = File.OpenText(fname);
        while ( (buffer = sReader.ReadLine()) !=null)
        {
            Console.WriteLine(buffer);
        }
    }
}
```

Bài tập 4: Hãy xem lại các ví dụ trong các chương trước, ví dụ nào có thể phái sinh ngoại lệ thì hãy thêm các đoạn xử lý ngoại lệ cho ví dụ đó.