SÁCH TỰ HỌC

DƯƠNG QUANG THIỆN
biên soạn

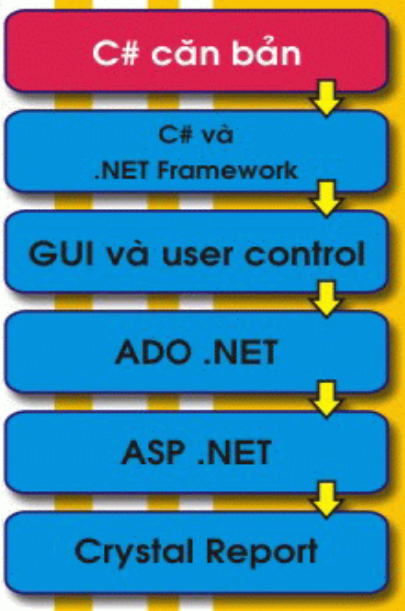
.NET Toàn tập

Tập 1

C# căn bản

và Visual Studio .NET IDE

Lập trình Visual C# thế nào?



NHÀ XUẤT BẢN TỔNG HỢP TP.HCM

C# căn bản

và Visual Studio .NET IDE

Lập trình Visual C# thế nào?

DƯƠNG QUANG THIỆN
Biên soạn

.NET Toàn tập

Tập 1

C# căn bản

và Visual Studio .NET IDE

Lập trình Visual C# thế nào?

NHÀ XUẤT BẢN TỔNG HỢP TP. HCM

Chịu trách nhiệm xuất bản:
TRẦN ĐÌNH VIỆT
Biên tập:
TRUNG HIẾU
Sửa bản in:
HỒNG HUÊ
Bìa:
HOÀNG NGỌC GIAO

NHÀ XUẤT BẢN TỔNG HỢP TP. HCM
62 Nguyễn Thị Minh Khai – Q.1
ĐT: 82225340 – 8296764 – 8220405 – 8296713 – 8223637
Fax: 8222726 – Email: nxbtphcm@bdvn.vnd.net

In 1000 cuốn, khổ 16 x 22cm, tại Xí nghiệp Cơ khí ngành in.
Giấy phép xuất bản số 399-191/XB-QLXB ký ngày 11-4-2003.
In xong và nộp lưu chiểu tháng 1-2005

Mục Lục

Chương 2: Bắt đầu từ đây ta tiến lên!

Chương 3: ***Sử dụng Debugger thế nào?***

Chương 4: *Căn bản Ngôn ngữ C#*

Chương 5: *Lớp và Đối tượng*

Chương 6: *Kế thừa và Đa hình*

Chương 7: *Nạp chồng tác tử*

Chương 8: *Cấu trúc Struct*

Chương 9: *Giao diện*

Chương 10: *Bản dãy, Indexers và Collections*

Chương 11: *Chuỗi chữ và biểu thức regular*

Chương 12: *Thụ lý các biệt lệ*

Chương 13: *Ủy thác và Tình huống*

Chương 14: *Lập trình trên môi trường .NET*

Lời mở đầu

Vào tháng 7/1998 người viết cho phát hành tập I bộ sách "Lập trình Windows sử dụng Visual C++ 6.0 và MFC". Toàn bộ gồm 8 tập, 6 nói về lý thuyết và 2 về thực hành. Các tập đi sau được phát hành lai rai mãi đến 10/2000 mới xong. Bộ sách được bạn đọc đón chào nồng nhiệt (mặc dầu chất lượng giấy và kiểu quay ronéo không được mỹ thuật cho lắm, nhưng giá rẻ vừa túi tiền bạn đọc) và được phát hành đi phát hành lại trên 10 ngàn bộ và không biết bao nhiêu đã bị photocopy và "bị luộc". Và vào thời điểm hoàn thành bộ sách lập trình Windows kể trên (tháng 10/2000) người viết cũng đã qua 67 tuổi, quá mệt mỏi, và cũng vào lúc vừa giải thể văn phòng SAMIS không kèn không trống, thế là người viết quyết định "rửa tay gác kiếm" luôn, mặc dầu trước đó vài ba tháng đã biết Microsoft manh nha cho ra đời một ngôn ngữ lập trình mới là C# trên một sàn diễn mang tên .NET. ám chỉ ngôn ngữ thời đại mạng Internet. Tuy nhiên, như đã định soạn giả vẫn ngưng viết, xem như nghỉ hưu luôn, quay về chăm sóc vườn phong lan bị bỏ bê từ lúc bắt đầu viết bộ sách lập trình Windows kể trên.

Nghỉ hưu thiếu vài tháng thì đúng 3 năm, vào tháng 5/2003, anh Nguyễn Hữu Thiện, người sáng lập ra tờ báo eChip, mời tham gia viết sách thành lập tủ sách tin học cho tờ báo. Thế là "a lê hấp" người viết đồng ý ngay, cho đặt mua một lô sách về C#, .VB.NET và .NET Framework để nghiên cứu. Càng đọc tài liệu càng thấy cái ngôn ngữ mới này nó khác với C++ đi trước khá nhiều, rõ ràng mạch lạc không rối rắm như trước và rất dễ học một cách rất ư tự nhiên. Thế là một mạch từ tháng 5/2003 đến nay, người viết đã hoàn chỉnh xong 5 trên tổng số 8 tập. Mỗi tập dài vào khoảng từ 600 đến 750 trang.

Bạn cứ thử hình dung là trong ngành điện toán, cứ vào khoảng một thập niên thì có một cuộc cách mạng nho nhỏ trong cách tiếp cận về lập trình. Vào thập niên 1960 là sự xuất hiện ngôn ngữ Cobol và Fortran (cũng như ngôn ngữ RPG của IBM) thay thế cho ngôn ngữ hợp ngữ, giữa thập niên 70 là sự xuất hiện máy vi tính với ngôn ngữ Basic, vào đầu thập niên 80 những công nghệ mới là Unix có thể chạy trên máy để bàn với ngôn ngữ cực mạnh mới là C, phát triển bởi ATT. Qua đầu thập niên 90 là sự xuất hiện của Windows và C++ (được gọi là C với lớp), đi theo sau là khái niệm về lập trình thiên đối tượng trong bước khai mào. Mỗi bước tiến triển như thế tượng trưng cho một đợt sóng thay đổi cách lập trình của bạn: từ lập trình vô tổ chức qua lập trình theo cấu trúc (structure programming hoặc procedure programming), bây giờ qua lập trình thiên đối tượng. Lập trình thiên đối tượng trên C++ vẫn còn "khó nuốt" đối với những ai đã quen cái nếp nghĩ theo kiểu lập trình thiên cấu trúc. Vả lại, lập trình thiên đối tượng vào cuối thập niên qua vẫn còn nhiều bất cập, không tự nhiên nên viết không thoải mái.

Bây giờ, với sự xuất hiện của .NET với các ngôn ngữ C#, VB.NET, J# xem ra cách suy nghĩ về việc viết chương trình của bạn sẽ thay đổi, trong chiều hướng tích cực. Nói

một cách ngắn gọn, sàn diễn .NET sẽ làm cho bạn triển khai phần mềm dễ dàng hơn trên Internet cũng như trên Windows mang tính chuyên nghiệp và thật sự thiên đối tượng. Nói một cách ngắn gọn, sàn diễn .NET được thiết kế giúp bạn triển khai dễ dàng những ứng dụng thiên đối tượng chạy trên Internet trong một môi trường phát tán (distributed). Ngôn ngữ lập trình thiên Internet được ưa thích nhất sẽ là C#, được xây dựng từ những bài học kinh nghiệm rút ra từ C (năng xuất cao), C++ (cấu trúc thiên đối tượng), Java (an toàn) và Visual Basic (triển khai nhanh, gọi là RAD - Rapid Application Development). Đây là một ngôn ngữ lý tưởng cho phép bạn triển khai những ứng dụng web phát tán được kết cấu theo kiểu ráp nối các cấu kiện (component) theo nhiều tầng nấc (n-tier).

Tập I được tổ chức thế nào?

Tập I này tập trung xoáy vào ngôn ngữ C#, phần căn bản nhất. Tập II nâng cao hơn, sẽ chỉ cho bạn cách viết các chương trình .NET trên các ứng dụng Windows và Web cũng như cách sử dụng C# với .NET Common Language Runtime. Đọc xong hai tập này, về mặt cơ bản bạn đã nắm vững phần nào ngôn ngữ Visual C#.

Chương 1: *Visual C# và .NET Framework* dẫn nhập bạn vào ngôn ngữ C# và sàn diễn .NET (.NET platform).

Chương 2: *Bắt đầu từ đây ta tiến lên!* "Xin Chào Bà Con!" cho thấy một chương trình đơn giản cung cấp một "bệ phóng" cho những gì sẽ tiếp diễn về sau, đồng thới dẫn nhập bạn vào Visual Studio .NET IDE và một số khái niệm về ngôn ngữ C#.

Chướng 3: *Sử dụng Debugger thế nào?*. Chương này chỉ cho bạn cách sử dụng bộ gỡ rối lỗi và sửa sai trong khi bạn viết chương trình. Ban sẽ thường xuyên tham khảo chương này trong suốt cuộc đời lập trình viên của bạn.

Chương 4: *Căn bản ngôn ngữ C#* Chương này trình bày những điều cơ bản về ngôn ngữ C# từ các kiểu dữ liệu "bẩm sinh" (built-in data type) đến các từ chốt (keyword). Ban sẽ hành nghề lập trình viên cũng giống như bà nội trợ nấu các món ăn. Nếu bà nội trợ phải rành rẻ các nguyên liệu mà mình sẽ chế biến thành những món ăn độc đáo thì bạn cũng phải rành các đặc tính của từng kiểu dữ liệu mà bạn sẽ dùng để "chế biến" cho ra những kết xuất mong muốn.

Chương 5: *Lớp và Đối tượng*: Vì bạn đang học lập trình thiên đối tượng nên lớp và đối tượng là hai khái niệm rất mới và quan trọng. Lớp (class) định nghĩa những kiểu dữ liệu mới (mà ta gọi là user-defined type - UDT, kiểu dữ liệu tự tạo) và cho phép nới rộng ngôn ngữ như vậy bạn mới có thể mô hình hóa vấn đề mà bạn đang giải quyết. Chương 5 này giải thích các cấu kiện (component) hình thành linh hồn của ngôn ngữ C#.

Chương 6: *Kế thừa và Đa hình (Inheritance & Polymorphisme)*: Các lớp có thể là những biểu diễn và trừu tượng hoá khá phức tạp của sự vật trong thực tại, nên chương 6 này đề

cập đến việc các lớp sẽ liên hệ với nhau thế nào cũng như tương tác thế nào để mô phỏng việc gì xảy ra thực thụ trong một thế giới thực.

Chương 7: *Nạp chồng tác tử (operator overloading)*: Chương này chỉ cho bạn cách thêm những tác tử vào trong các kiểu dữ liệu tự mình tạo ra.

Chương 8: *Cấu trúc Struct*: Struct là "anh chị em họ hàng" với lớp nhưng thuộc loại đối tượng nhẹ cân, tầm hoạt động hạn chế hơn và ít tốn hao ký ức (overhead) đối với hệ điều hành.

Chương 9: *Giao diện (interface)*: cũng là "anh chị em họ hàng" với lớp nhưng đây lại là những "hợp đồng giao dịch" mô tả cách một lớp sẽ hoạt động thế nào, như vậy các lập trình viên khác có thể tương tác với các đối tượng của bạn theo những thể thức đã được định nghĩa đúng đắn và đầy đủ.

Chương 10: *Bản dãy, Indexers và Collections*: Các chương trình thiên đối tượng thường xuyên tạo ra phần lớn những đối tượng. Những đối tượng này phải được tổ chức theo một hình thức nào đó để có thể thao tác lên chúng với nhau: đây là những bản dãy, những collection v.v.. C# cung cấp những hỗ trợ rộng rãi đối với collection. Chương này sẽ khảo sát các lớp collection mà Base Class Library cung cấp cũng như chỉ bạn thấy cách tạo những kiểu dữ liệu collection riêng cho bạn.

Chương 11: *Chuỗi chữ và biểu thức regular*: Phần lớn các chương trình Windows hoặc Web đều tương tác với người sử dụng và chuỗi chữ (string) giữ vai trò quan trọng trong giao diện người sử dụng (user interface). Chương 10 này chỉ cho bạn cách sử dụng C# trong việc thao tác các dữ liệu kiểu văn bản.

Chương 12: *Thụ lý các biệt lệ (Exception handling):* Một chương trình chạy tốt, tin tưởng được là loại chương trình không có lỗi sai. Việc tiên liệu những trường hợp biệt lệ (exception) và cách thụ lý những biệt lệ này là mấu chốt của vấn đề chất lượng của một phần mềm tin học, nên rất quan trọng không thể xem thường. Chương 12 này chỉ cho bạn cách thụ lý các biệt lệ theo một cơ chế thống nhất.

Chương 13: *Ủy thác và tình huống (Delegate & Event)*: Tất cả các chương trình Windows cũng như Web đều được vận hành theo tình huống (gọi là event driven) giống như cầu thủ đá bóng hoạt động dựa theo tình huống của trái banh. Do đó, trên C#, tình huống được xem như là thành viên trụ cột của ngôn ngữ. Chương 13 này tập trung vào việc các tình huống sẽ được quản lý thế nào, và cách các hàm ủy thác, một cơ chế callback (hàm nhắn lại) an toàn, sẽ được sử dụng thế nào để hỗ trợ việc thụ lý các tình huống.

Chương 14: *Lập trình trên môi trường .NET*: Chương này chuẩn bị cho việc qua giai đoạn viết các chương trình .NET theo C# của tập II.

Bộ sách gồm những tập nào?

Như đã nói, bộ sách này gồm 8 tập, 6 nói về lý thuyết và 2 về thí dụ thực hành.

Tập II: Visual C# và Sàn diễn .NET

Chương 1 Input/Output và Object serialization
Chương 2 Xây dựng một ứng dụng Windows
Chương 3 Tìm hiểu về Assembly và cơ chế Version
Chương 4 Tìm hiểu về Attribute và Reflection
Chương 5 Marshalling và Remoting
Chương 6 Mạch trình và Đồng bộ hoá
Chương 7 Tương tác với unmanaged code
Chương 8 Lập trình ứng dụng Web
Chương 9 Lập trình Web Service

Tập III: Giao diện người sử dụng viết theo Visual C#

Chương 1 Tạo giao diện người sử dụng dùng lại được
Chương 2 Thiết kế giao diện theo Lớp và Tam nguyên
Chương 3 Tìm hiểu đồ hoạ và GDI+
Chương 4 Tìm hiểu biểu mẫu
Chương 5 Cơ bản về lớp Control
Chương 6 Windows Forms Controls
Chương 7 Các ô control tiên tiến
Chương 8 Custom Controls
Chương 9 Hỗ trợ Custom Control vào lúc thiết kế
Chương 10 MDI Interfaces và Workspace
Chương 11 Dynamic User Interfaces
Chương 12 Data Controls
Chương 13 GDI+ Controls
Chương 14 Hỗ trợ Help

Tập IV: Lập trình Căn Cứ Dữ Liệu với Visual C# & ADO.NET

Chương 1 Sử dụng Căn cứ dữ liệu
Chương 2 Tổng quan về ADO .NET
Chương 3 Data Component trong Visual Studio .NET
Chương 4 Các lớp ADO.NET tách rời
Chương 5 ADO.NET Data Providers
Chương 6 Trình bày IDE theo quan điểm Database
Chương 7 Làm việc với XML

Tập V: Lập trình ASP.NET & Web

Tập VI: **Lập trình các báo cáo dùng Crystal Reports .NET**

Tập VII: Sổ tay kỹ thuật C# - phần A

Chưa định hình các chương

Tập VIII: Sổ tay kỹ thuật C# - phần B

Chưa định hình các chương

Bộ sách này dành cho ai?

Bộ sách này được viết dành cho những ai muốn triển khai những ứng dụng chạy trên Windows hoặc trên Web dựa trên nền .NET. Chắc chắn là có nhiều bạn đã quen viết C++, Java hoặc Visual Basic, hoặc Pascal. Cũng có thể bạn đọc khác lại quen với một ngôn ngữ khác hoặc chưa có kinh nghiệm gì về lập trình ngoài lập trình cơ bản. Bộ sách này dành cho tất cả mọi người. Vì đây là một bộ sách tự học không cần thầy, chỉ cần có một cái máy tính được cài đặt .NET. Đối với ai chưa hề có kinh nghiệm lập trình, thì hơi khó một chút nhưng "cày đi cày lại" thì cũng vượt qua nhanh những khó khăn này. Còn đối với những ai đã có kinh nghiệm lập trình, thì sẽ mê ngay ngôn ngữ này và chỉ trong một thời gian rất ngắn, 6 tháng là tối đa là có thể nắm vững những ngóc ngách của ngôn ngữ mới này, và có thể biết đâu trong một thời gian rất ngắn bạn trở thành một guru ngôn ngữ C#. Người viết cũng xin lưu ý bạn đọc là bộ sách này là sách tự học (tutorial) chứ không phải một bộ sách tham chiếu (reference) về ngôn ngữ, nên chỉ mở đường phát quang hướng dẫn bạn đi khỏi bị lạc, và đem lại 60% kiến thức về ngôn ngữ. Và khi học tới đâu, tới một chặng đường nào đó bạn có thể lên MSDN phăng lần đào sâu từng đề mục con mà bạn đang còn mơ hồ để có thể phăng lần 40% còn lại kiến thức để nắm vững vấn đề. Lấy một thí dụ. Trong bộ sách này, chúng tôi thường đề cập đến các lớp. Chúng tôi giải thích tổng quát cho biết lớp sẽ được dùng vào việc gì và sử dụng một số hàm hành sự (method) hoặc thuộc tính (property) tiêu biểu của lớp này trong những thí dụ cụ thể. Thế nhưng mỗi lớp có vô số hàm hành sự và thuộc tính cũng như tình huống. Thì lúc này bạn nên vào MSDN tham khảo từng hàm hành sự hoặc thuộc tính một của lớp này để bạn có một ý niệm sơ sơ về những công năng và đặc tính của lớp. Một số chức năng/đặc tính bạn sẽ chả bao giờ sử dụng đến, còn một số thi thoảng bạn mới cần đến. Cho nên về sau, khi bạn muốn thực hiện một chức năng gì đó, thì bạn có thể vào lại MSDN xem lớp có một hàm hoặc thuộc tính đáp ứng đúng (hoặc gần đúng) nhu cầu của bạn hay không và nếu có thì lúc này bạn mới xem kỹ cách sử dụng. Kinh nghiệm cho thấy, là trong suốt cuộc đời hành nghề lập trình viên, bạn sẽ xài đi xài lại cũng chừng nấy lệnh, hoặc một số hàm nào đó theo một mẫu dáng (pattern) nào đó, nên một khi bạn đã khám phá ra những lệnh hoặc hàm này, và áp dụng thành công thì bạn sẽ thường xuyên dùng đến một cách máy móc không cần suy nghĩ gì thêm.

Theo tập quán phát hành sách hiện thời trên thế giới, thì sách sẽ kèm theo một đĩa mềm hoặc đĩa CD chứa các bài tập thí dụ. Ở đây rất tiếc, chúng tôi không làm thế vì nhiều lý do. Thứ nhất giá thành sẽ đội lên, mà chúng tôi thì lại muốn có những tập sách giá bán đến tay bạn đọc rẻ bằng 50% giá hiện hành của các sách khác cùng dung lượng (nhưng khác chất lượng nội dung). Thứ hai, các bạn chịu khó khỏ lệnh vào máy, khỏ tới đâu bạn đọc hiểu tới đấy. Đôi khi khỏ lệnh sai, máy bắt lỗi bạn sẽ biết những thông điệp cảnh báo lỗi nói gì để về sau mà cảnh giác. Còn nếu tải chương trình xuống từ đĩa vào máy, cho thử chạy tốt rồi bạn bằng lòng rốt cuộc chả hiểu và học gì thêm. Khi khỏ một câu lệnh như thế bạn phải biết bạn đang làm gì, thực hiện một tác vụ gì, còn như nhắm mắt tải lệnh xuống thì cũng chẳng qua là học vẹt mà thôi không động não gì cả.

Chúng tôi hy vọng bộ sách sẽ giúp bạn có một nền tảng vững chắc trong việc lập trình trên .NET.

Ngoài ra, trong tương lai, nếu sức khoẻ cho phép (vì dù gì thì tuổi soạn giả cũng gần 72) chúng tôi dự kiến ra bộ sách về phân tích thiết kế các ứng dụng điện toán sử dụng UML và Pattern. Trong những buổi gặp gỡ giữa bạn bè và một đôi lần trên báo chí khi họ than phiền là kỹ sư tin học bây giờ ra trường không sử dụng được, chúng tôi thường hay phát biểu là không ngạc nhiên cho lắm khi ta chỉ cho ra toàn là "thợ lập trình" giống như bên xây dựng là thợ hồ, thợ nề thợ điện thợ mộc v..v.. chứ đâu có đào tạo những kiến trúc sư (architect) biết thiết kế những bản vẽ hệ thống. Do đó, chúng tôi dự kiến (hy vọng là như vậy) là sẽ hoàn thành một bộ sách đề cập đến vấn đề phân tích thiết kế những hệ thống sử dụng những thành tựu mới nhất trên thế giới là UML và Pattern với những phần mềm thích ứng là IBM Rational Rose XDE và Microsoft Visio for Enterprise Architect . Ngoài ra, những gì học ở trường là thuần túy về kỹ thuật lập trình, về mô phỏng, trí tuệ nhân tạo, lý thuyết rời rạc v.v.. (mà những mớ lý thuyết này không có đất dụng võ) nhưng khi ra trường vào các xí nghiệp thì mù tịt về quản lý nhân sự, về kế toán về tồn kho vật tư, về tiêu thụ v..v.. mà 80% ứng dụng tin học lại là vào các lãnh vực này. Do đó, trong bộ sách mà chúng tôi dự kiến sẽ soạn những tập đi sâu vào xây dựng những hệ thống quản lý trong các cơ quan xí nghiệp hành chánh cũng như thương mại.

Đôi lời cuối cùng

Kể từ năm 1989, năm thành lập văn phòng dịch vụ điện toán SAMIS, cho đến nay gần trên 15 năm chúng tôi cùng anh chị em trong nhóm SAMIS đã biên soạn trên 55 đầu sách, và cũng đã phát hành gần 400.000 bản, trong ấy 60% là phần của người viết. Từ những tiền lời kiếm được do tự phát hành lấy cộng thêm tiền hưu tiết kiệm của bà vợ người Thụy sĩ, hằng năm chúng tôi đã dành toàn bộ số tiền này để xây các trường cho những vùng sâu vùng xa trên 15 tỉnh thành đất nước (Sơn La, Nghệ An, Quảng Ngải, Quảng Nam, Quảng Trị, Bình Định, Ban Mê Thuột, Pleiku, Darlak, Bà Rịa Vũng Tàu, Đồng Nai, Sông Bé, TP Hồ Chí Minh, Cần Thơ, và Cà Mau), cấp học bổng cho các sinh viên nghèo tại các đại học Huế, Đà Nẵng, An Giang và TP Hồ Chí Minh, hỗ trợ vốn cho giáo viên ở An Lão (Bình Định), xây nhà cho giáo viên ở vùng sâu vùng xa (Bình Định,

Quảng Trị), và tài trợ mổ mắt cho người nghèo ở An Giang (4 năm liền). Các hoạt động xã hội này đều thông qua sự hỗ trợ của hai tờ báo Tuổi Trẻ và Sài Gòn Giải Phóng. Không ngờ những việc làm rất cá nhân này lại được Nhà Nước "theo dõi" đến nỗi không biết vị nào đã "xúi" Chủ tịch nước Trần Đức Lương ký quyết định tặng người viết Huân Chương Lao Động Hạng 3, ngày 29/8/2004. Nói ra điều này, chúng tôi muốn bạn đọc hiểu cho là tự nội lực của ta, ta cũng có thể giúp đỡ giáo dục mà khỏi nhờ viện trợ của các nước Nhật, Hàn Quốc. Nếu các bạn ý thức rằng mỗi tập sách bạn mua của chúng tôi thay vì mua sách luộc hoặc photocopy là bạn đã gián tiếp tham gia vào chương trình xây trường lớp cho vùng sâu vùng xa cũng như hỗ trợ học bổng cho sinh viên nghèo của chúng tôi.

Cuối cùng, chúng tôi xin cảm ơn sự hỗ trợ của các anh chị Hoàng Ngọc Giao, Võ Văn Thành và Trần Thị Thanh Loan trong việc hoàn thành bộ sách này.

TP Hồ Chí Minh ngày 1/12/2004.

Dương Quang Thiện

Chương 1

Visual C# và .NET Framework

Mục tiêu của C# là đem lại cho bạn một ngôn ngữ lập trình đơn giản, an toàn, tiên tiến, mang tính thiên đối tượng, có hiệu năng cao và tập trung xoáy vào Internet đối với việc triển khai phần mềm trên nền tảng .NET. C# là một ngôn ngữ lập trình mới, nhưng nó đã rút ra những bài học kinh nghiệm được đúc kết từ 3 thập niên qua. Giống như khi bạn thấy những bạn trẻ với những hành động và nhân cách của những bậc cha mẹ và ông bà đi trước. Ban có thể thấy dễ dàng ảnh hưởng của C++ và của Java, Visual Basic (VB) và những ngôn ngữ khác trên C#.

Bộ sách này, gồm 6 tập trên 4000 trang, sẽ tập trung vào ngôn ngữ C# (thật ra là Visual C#) và cách sử dụng nó như là một công cụ lập trình trên sàn diễn .NET (.NET platform). Chúng tôi sẽ cùng bạn sử dụng C# để tạo những ứng dụng chạy đơn độc trên máy PC (gọi là desktop application), trên Windows cũng như trên Internet.

Chương này dẫn nhập bạn vào cả ngôn ngữ C# lẫn sàn diễn .NET, bao gồm .NET Framework.

1.1 Sàn diễn .NET

Khi Microsoft thông báo sáng kiến .NET với ngôn ngữ C# vào tháng 7/2000, và mãi đến tháng 4/2003 phiên bản 1.1 của toàn bộ .NET Framework mới được lưu hành, báo hiệu sự xuất hiện của sàn diễn .NET. Sàn diễn này thực chất là một khuôn giá (framework) triển khai phần mềm hoàn toàn mới, một cách tiếp cận mới làm cho cuộc sống lập trình viên dễ thở hơn. Khuôn giá này cung cấp một API (application programming interface) "tươi mát" đối với những dịch vụ và API cỗ điển của hệ điều hành Windows, đặc biệt Windows 2000, đồng thời kéo lại gần nhau những công nghệ rời rạc mà Microsoft đã phát triển trong những năm cuối thập kỷ 1990. Ta có thể kể những dịch vụ cấu kiện (component) COM+, khuông giá triển khai ASP Web, XML và thiết kế thiên đối tượng, và những hỗ trợ đối với những nghi thức mới liên quan đến Web service chẳng hạn SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) và UDDI (Universal Description, Discovery, and Integration), cũng như sự chú tâm vào Internet, tất cả đều hòa mình vào kiến trúc DNA (Distributed interNet Applications Architecture).

Phạm vi của sàn diễn .NET rất rộng lớn. Nó bao gồm 4 nhóm sản phẩm:

- ***Các công cụ triển khai phần mềm và các thư viện***.
  Một bộ ngôn ngữ lập trình, bao gồm Visual C#, J#, và Visual Basic .NET; một lô công cụ triển khai, bao gồm Visual Studio .NET; một thư viện lớp (class library) để xây dựng những dịch vụ Web, cũng như những ứng dụng Web và Windows; kể luôn cả ***Common Language Runtime*** (CLR) lo thi hành những đối tượng được xây dựng trong khuông giá .NET Framework.

- ***Các server chuyên biệt***
  Một bộ .NET Enterprise Servers, trước đó được biết dưới cái tên là SQL Server 2000, Exchange 2000, BizTalk 2000, v.v.. cung cấp những chức năng đối với việc trữ dữ liệu theo quan hệ (relational data storage), email, B2B commerce v.v..

- ***Web Services***
  Sản phẩm chào hàng liên quan đến những dịch vụ thương mại trên mạng Web, được mang tên Project Hailstorm; các nhà phát triển phần mềm có thể sử dụng những dịch vụ này để tạo ra những ứng dụng cần đến sự hiểu biết về lai lịch và sở thích của người sử dụng (trong mục đích marketing), v.v.. Chỉ cần trả một số tiền "thù lao" nho nhỏ các lập trình viên có thể sử dụng những dịch vụ này trong việc xây dựng ứng dụng cần đến chúng.

- ***Các thiết bị (devices)***
  Những thiết bị không thuộc máy tính nhưng có khả năng sử dụng .NET, từ điện thoại di động đến những trò chơi game.

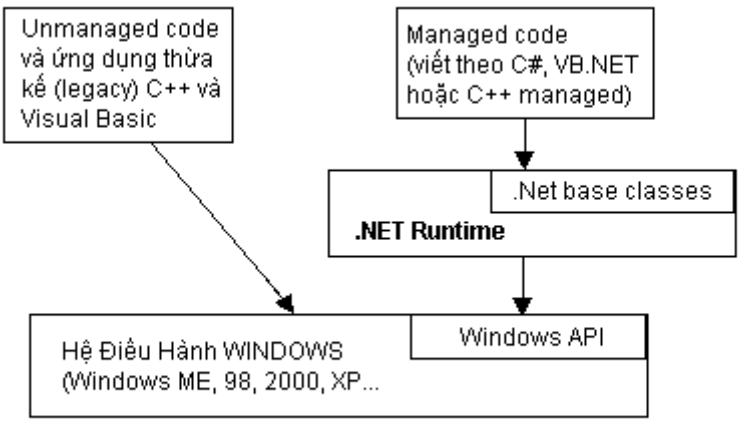
1.2 Tổng quan về .NET Framework

Cách dễ nhất để nghĩ về .NET Framework là môi trường mà đoạn mã của bạn sẽ hoạt động. Đây có nghĩa là .NET sẽ quản lý việc thi hành chương trình - khởi động chương trình, cấp phép hoạt động, cấp phát ký ức để trữ dữ liệu làm việc, hỗ trợ việc thu hồi nguồn lực (resource) và ký ức không dùng đến, v.v.. Tuy nhiên, ngoài việc tiến hành những công tác vừa kể trên, .NET còn chuẩn bị sẵn một thư viện lớp - được gọi là **.NET base classes** (lớp cơ bản .NET), cho phép thực hiện vô số tác vụ trên Windows. Nói tóm lại, .NET giữ hai vai trò: quản lý việc thi hành chương trình của bạn và cung cấp những dịch vụ mà chương trình của bạn cần đến.

Trước khi xem việc gì sẽ xảy ra khi một chương trình .NET chạy, ta thử lướt qua một vài cấu kiện hình thành .NET Framework và một số từ ngữ mà ta sẽ cần đến. Nếu danh sách sau đây chưa nói lên điều gì đối với bạn (vì có thể bạn mới tập tễnh lập trình) thì cũng không sao, về sau khi đọc hết bộ sách này bạn có thể trở về đọc lại thì bạn sẽ hiểu ngay.

- **.NET Runtime**[1], còn được gọi là **Common Language Runtime** (tắt CLR), hiện là bộ phận lo quản lý việc thi hành đoạn mã của bạn: nạp chương trình, cho chạy đoạn mã theo những mạch trình (thread) nhất định cũng như quản lý các mạch trình này và cung cấp tất cả các dịch vụ hỗ trợ ở hậu trường. CLR tạo một môi trường theo đấy chương trình được thi hành. CLR bao gồm một "cỗ máy ảo" (virtual machine) tương tự như Java virtual machine (JVM). Ở cấp cao, CLR cho hiện dịch các đối tượng, tiến hành những kiểm tra an toàn đối với các đối tượng này, bố trí chúng lên ký ức[2], cho thi hành chúng rồi xong việc cho thu hồi ký ức chúng chiếm dụng trong thời gian "tại chức". Nói tóm lại CLR được xem như là linh hồn của kiến trúc .NET.

- **Managed code** (đoạn mã được giám quản[3]): bất cứ đoạn mã nào được thiết kế để chạy trên môi trường .NET được gọi là **đoạn mã được giám quản**. Những đoạn mã khác, đơn giản chạy trên Windows, ngoài môi trường .NET, thì được gọi là **unmanaged code** (đoạn mã vô quản). Ban nhớ cho **.NET Runtime** chỉ là một lớp phần mềm nằm giữa hệ điều hành Windows và các ứng dụng. .NET không phải là một hệ điều hành. Hệ điều hành Windows vẫn còn đó. Lẽ dĩ nhiên những ứng dụng unmanaged vẫn tiếp tục làm việc với Windows cũng như trực tiếp với Windows API giống như trước kia. Ban có thể hình dung tình trạng manged code và umnamaged code như theo hình 1-1 sau đây:



Hình 1-1 : .NET Runtime, Managed code và Unmanaged code

[1] Runtime là những gì xảy ra vào lúc chạy chương trình
[2] Chúng tôi thường dịch "memory" là ký ức, chứ không dùng từ "bộ nhớ", vì memory mang ý nghĩa ký thác, cầm giữ dữ liệu chứ không có nhớ nhung gì cả.
[3] Tắt chữ "giám sát và quản lý".

- **Intermediate Language (IL)**: *Ngôn ngữ trung gian*. Các chương trình .NET không được biên dịch thành tập tin khả thi (executable, .EXE), mà lại được biên dịch thành tập tin IL. Khi bạn cho biên dịch managed code, thì trình biên dịch sẽ cho ra IL, rồi CLR sẽ lo giai đoạn biên dịch vào giờ chót ngay trước khi đoạn mã được thi hành. IL được thiết kế có khả năng biên dịch nhanh ra ngôn ngữ máy nguyên sinh (native machine code), trong khi vẫn hỗ trợ những chức năng của .NET. Tập tin IL được tạo ra đối với C# cũng tương tự như các tập tin IL được tạo ra đối với các ngôn ngữ khác J# hoặc VB.NET bởi CLR. Điểm chủ yếu ở đây là CLR hỗ trợ C# cũng như VB.NET giống như nhau, không "phân biệt đối xử".

- **Common Type System (CTS)**: *Đặc tả kiểu dữ liệu thông dụng*. Để có thể thực hiện việc liên thông[4] ngôn ngữ (language interoperability), nghĩa là các ngôn ngữ khác nhau có thể hiểu nhau, cần thiết phải đồng thuận trên một tập hợp những kiểu dữ liệu cơ bản, để từ đấy có thể chuẩn hóa tất cả các ngôn ngữ. CTS cung cấp định nghĩa trên, đồng thời cung cấp những qui tắc định nghĩa những lớp "cây nhà lá vườn" (custom class). Tất cả các cấu kiện .NET phải tuân thủ các qui tắc này. Thí dụ, trên .NET mọi thứ đều là một đối tượng của một lớp đặc trưng nào đó và lớp này được dẫn xuất từ lớp gốc mang tên `System.Object`. CTS hỗ trợ khái niệm chung về các lớp, giao diện (interface), ủy thác (delegate), các kiểu dữ liệu qui chiếu (reference type) và kiểu dữ liệu trị (value type).

- **.NET Base Classes.** *Lớp cơ bản .NET*. Đây là một thư viện lớp rất đồ sộ chứa toàn những đoạn mã viết sẵn trước cho phép thực hiện vô số công tác trên Windows, đi từ hiển thị những cửa sổ và biểu mẫu (form), triệu gọi các dịch vụ cơ bản Windows, đọc/viết các tập tin, thâm nhập vào mạng, Internet cũng như truy xuất các nguồn dữ liệu (trên căn cứ dữ liệu chẳng hạn). Khi bạn học toàn bộ tập sách này (gồm 6 cuốn), thì bạn cũng sẽ biết qua khá nhiều lớp cơ bản .NET.

- **Assembly**.[5] Một assembly là một đơn vị theo đấy đoạn mã managed được biên dịch sẽ được trữ trong assembly. Nó gần giống như một EXE hoặc DLL, nhưng điểm quan trọng của assembly là nó có khả năng tự mô tả lấy mình. Assembly có chứa một loại dữ liệu được gọi là **metadata**, cho biết những chi tiết của assembly cũng như tất cả các kiểu dữ liệu, hàm hành sự (method) v.v.. được định nghĩa trong assembly. Một assembly có thể là private (riêng tư, chỉ được truy xuất bởi ứng dụng mà thôi) hoặc shared (được chia sẻ sử dụng, được truy xuất bởi bất cứ ứng dụng nào trên Windows).

[4] Liên thông là thông thương liên lạc giữa các ngôn ngữ

[5] Assembly ở đây phải hiểu theo nghĩa "một dây chuyền lắp ráp" chứ không được nhầm lẫn với hợp ngữ assembly, vì có người dịch là "bộ hợp ngữ".

- **Assembly cache.** Đây là một vùng ký ức trên đĩa dành trữ các assembly có thể được chia sẻ sử dụng (shared assembly). Trên mỗi máy tính có cài đặt CLR sẽ có một code cache được gọi là GAC (global assembly cache) dùng để trữ những assembly đặc biệt được thiết kế chia sẻ sử dụng bởi nhiều ứng dụng. Chúng tôi muốn giải thích về từ ngữ "cache". Nếu dịch theo từ điển là "nơi cất giấu' bạn không thể hình dung ra là cái gì. Chúng tôi đành giải thích theo công dụng ám chỉ. Khi bạn nạp một đối tượng vào ký ức, thì cả một thủ tục rắc rối mất thời gian, do đó lần đầu tiên khi đối tượng được nạp xong vào ký ức, thì CLR sẽ cho ghi con trỏ (pointer) chỉ về đối tượng lên một vùng ký ức (ký ức CPU hoặc ký ức đĩa) được mệnh danh là cache. Khi một chương trình nào đó muốn sử dụng cùng đối tượng này, thì CLR đến cache xem có con trỏ chỉ về đối tượng này hay không. Nếu có, thì báo cho chương trình biết đến lấy mà sử dụng. Nếu không có, thì CLR phải làm thủ tục nạp đối tượng ban đầu. Như vậy, cache là một phương tiện giúp tăng hiệu năng hoạt động của chương trình, cũng như giảm việc chiếm dụng ký ức. Nói tóm lại, cache ở đây là vùng ký ức tạm thời dùng cất trữ cái gì đó, nằm ẩn sau hậu trường, và có thể lấy ra dùng ngay lập tức khỏi tìm kiếm lung tung.

- **Common Language Specification (CLS).** *Đặc tả ngôn ngữ thông dụng*. Đây là một tập hợp tối thiểu những chuẩn mực (qui tắc) bảo đảm đoạn mã này có thể được truy xuất bất cứ ngôn ngữ nào, cho phép hội nhập ngôn ngữ. Tất cả các ngôn ngữ nào nhắm hoạt động trên .NET phải hỗ trợ CLS. Từ ngữ *.NET-aware* (tạm dịch là "ăn ý với .NET) dùng để chỉ loại ngôn ngữ này. CLS sẽ hình thành một tập hợp con chức năng có sẵn trên .NET và trên IL, và sẽ không có vấn đề gì khi đoạn mã của bạn sử dụng thêm những chức năng không thuộc phạm vi CLS. Trong trường hợp này, các chức năng bất-CLS sẽ không được sử dụng lại bởi các ngôn ngữ khác. Nhìn chung, các trình biên dịch nào tuân theo các qui tắc CLS sẽ tạo những đối tượng có thể hoạt động liên thông với các trình biên dịch khác "ăn ý với .NET".

- **Reflection**. *Phản chiếu*. Vì assembly có thể tự mình mô tả, nên theo lý thuyết ta có khả năng thâm nhập bằng chương trình vào metadata của assembly. Trong thực tế, có một vài lớp cơ bản được thiết kế lo việc này. Công nghệ này được gọi là reflection, nghĩa là chương trình dùng tiện nghi này để tự "soi mình trong gương" quan sát nội dung của metadata của mình. Trong y khoa bây giờ có từ "nội soi" nghĩa là quan sát lục phủ ngủ tạng mà khỏi mổ. Ban có thể xem reflection như là "nội soi" chương trình.

- **Just-in-Time (JIT) compilation**: *biên dịch vừa đúng lúc*. Đây là tiến trình (thường được gọi là *JITing*) thực hiện giai đoạn chót của việc biên dịch từ IL qua mã máy nguyên sinh (native machine code). Từ này để chỉ một đoạn mã nhỏ được biên dịch theo yêu cầu vào giờ chót. Kết quả của JITing là một mã máy có thể được thi hành bởi bộ xử lý (processor) của máy. Trình biên dịch JIT chuẩn sẽ

chạy theo *yêu cầu.* Khi một hàm hành sự (method) được triệu gọi, JIT compiler sẽ phân tích IL và cho ra một đoạn mã máy rất hữu hiệu chạy rất nhanh. JIT compiler khá thông minh để có thể biết là đoạn mã nào đã được biên dịch, nên việc biên dịch chỉ xảy ra khi nào cần thiết. Do đó, khi các ứng dụng .NET chạy, thì chúng chạy ngày càng nhanh khi có những đoạn mã đã biên dịch sẵn được dùng đi dùng lại (giống như kiểu "fast food", thức ăn nhanh).

- **Manifest[6].** Đây là vùng ký ức (thường trên đĩa) của một assembly có chứa meta data, cho trữ tất cả các qui chiếu cùng những mối lệ thuộc (dependency) của assembly đối với các assembly khác.

- **Application Domain.** *Địa bàn hoạt động của ứng dụng*. Đây là cách mà CLR cho phép nhiều đoạn mã khác nhau có thể chạy trên cùng một khoảng không gian ký ức của một process (đơn vị xử lý). Việc cách ly (isolation) giữa các đơn vị mã sẽ được thực hiện bằng cách dùng kỹ thuật an toàn kiểu dữ liệu (type safety) của IL để kiểm tra trước khi thi hành mỗi đoạn mã nhỏ để chúng cư xử tốt, không "quậy phá lung tung".

- **Garbage Collection**: *Thu gom rác*. Bộ phận dịch vụ lo thu hồi ký ức không còn dùng nữa, như vậy tránh khỏi việc rò rỉ ký ức (memory leak) thường xảy ra trong các ngôn ngữ đi trước. Lúc này, với .NET ứng dụng của bạn khỏi bận tâm lo việc thu hồi ký ức này.

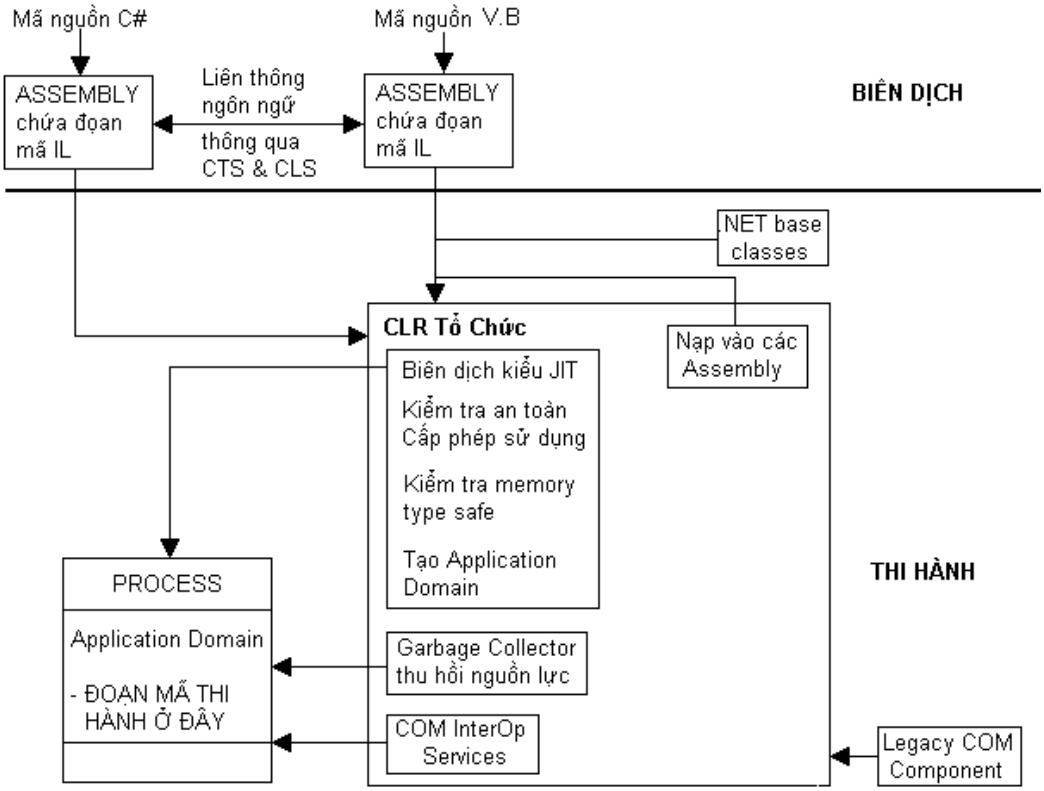
1.3 Việc gì xảy ra khi bạn biên dịch và chạy một chương trình

Trước tiên, ta sẽ có một cái nhìn bao quát xem .NET hoạt động thế nào cũng như những dịch vụ nào nó cung cấp, bằng cách xem việc gì xảy ra khi bạn cho chạy một chương trình trên nền tảng .NET. Hình 1-2 sau đây tóm lược tiến trình này. Chúng tôi giả định bạn đang lập trình một ứng dụng gồm phần chính đoạn mã viết theo C#, và một lớp thư viện viết theo Visual Basic .NET. Ứng dụng cũng đòi hỏi triệu gọi thêm một cấu kiện COM "di sản kế thừa" (legacy), và có thể nói ứng dụng sẽ dùng ít nhất vài lớp cơ bản .NET.

Trên hình 1-2, những ô hình chữ nhật tượng trưng cho những cấu kiện (component) chính tham gia vào việc biên dịch và thi hành chương trình, trong khi những mũi tên cho

[6] Từ này mượn của ngành hàng không, hàng hải. Dach sách hành khách chuyến bay được gọi là manifest (không vận đơn)

biết những công tác được thực hiện. Phần trên đỉnh hình đồ cho thấy tiến trình biên dịch riêng rẽ mỗi dự án (project) thành một assembly. Hai assembly có khả năng tương tác với nhau, nhờ vào chức năng "liên thông ngôn ngữ" (language interoperability) của .NET. Nửa phần dưới hình đồ cho thấy tiến trình biên dịch JIT (tắt chữ just in time) từ IL trên các assembly thành đoạn mã hiện hành.



Hình 1-2: Các giai đoạn biên dịch & thi hành chương trình

1.3.1 Biên dịch mã nguồn

Trước khi chương trình có thể thi hành được, nó phải qua giai đoạn biên dịch. Tuy nhiên, khác với những gì trước kia đối với các tập tin khả thi EXE hoặc DLL, đoạn mã C# sau khi biên dịch xong sẽ không chứa các chỉ thị của hợp ngữ (assembly language). Thay vào đó, đoạn mã chứa chỉ thị theo ngôn ngữ được gọi là Microsoft Intermediate Language (viết tắt MSIL hoặc IL), ngôn ngữ trung gian. IL bắt chước ý tưởng "byte code" của Java. Đây là một ngôn ngữ cấp thấp được thiết kế để có thể chuyển đổi nhanh thành mã máy nguyên sinh (native machine code) thông qua JIT.

"Gói chương trình biên dịch" (gọi là package, vì nó được "đóng gói") được phân phối cho khách hàng thường bao gồm một số assembly; mỗi assembly chứa đoạn mã IL, nhưng lại còn chứa metadata dùng mô tả các kiểu dữ liệu và hàm hành sự (method) trong assembly. Ngoài ra, metadata lại còn chứa một *đoạn mã băm dằm* (hash code) nội dung assembly, dùng kiểm tra xem assembly không bị giả mạo hoặc bị đánh tráo khi đến tay khách hàng, kể cả thông tin liên quan đến phiên bản cũng như những chi tiết liên quan đến các assembly khác mà assembly này có thể qui chiếu. Thông tin quyết định cho phép chạy đoạn mã này hay không cũng được đính kèm.

Trong quá khứ, một package phần mềm trọn vẹn thường bao gồm chương trình chính với điểm đột nhập (entry point) kèm theo một hoặc nhiều thư viện và những cấu kiện COM. Với .NET, thì package phần mềm chứa một số assembly, một trong những assembly này là chương trình khả thi chứa điểm đột nhập chính, còn các assembly khác được chỉ định là những thư viện. Trong thí dụ ở trên (hình 1–2) ta sẽ có 2 assembly - một khả thi chứa đoạn mã được biên dịch viết theo C#, và một thư viện chứa đoạn mã biên dịch viết theo VB .NET.

1.3.2 Thi hành chương trình biên dịch

Khi cho chạy chương trình, .**NET Runtime** sẽ nạp assembly đầu tiên vào – assembly có chứa điểm đột nhập. Nó sẽ dùng đoạn mã băm (hash code) kiểm tra xem có bị giả mạo hay không, đồng thời .NET Runtime sẽ dùng metadata để kiểm tra tất cả các kiểu dữ liệu bảo đảm là có thể chạy assembly, kể cả việc kiểm tra an toàn xem người sử dụng có quyền sử dụng assembly hay không.

Một điều mà CLR sẽ làm trong giai đoạn này là kiểm tra gì đó được gọi là **memory type safety** của đoạn mã, nghĩa là truy xuất ký ức cách nào mà CLR có thể kiểm soát được. Ý nghĩa của việc kiểm tra này là bảo đảm đoạn mã không được đọc hoặc viết lên bất cứ đoạn ký ức nào không thuộc quyền sử dụng của mình, nghĩa là không chạy loạng quạng qua phần ký ức dành cho ứng dụng khác. Nếu CLR không thể kiểm tra đoạn mã thuộc loại memory type safety thì nó có quyền từ chối cho chạy đoạn mã, và điều này tùy thuộc chính sách an toàn của máy tại chỗ.

Cuối cùng, sau khi thoát khỏi các cửa ải kiểm tra, CLR cho chạy thi hành đoạn mã. Nó tạo ra một "đường đua" (process[7], tiến trình xử lý) đối với đoạn mã này để cho chạy trên đường đua này, đồng thời đánh dấu một application domain (địa bàn hoạt động của ứng dụng), theo đấy nó đặt vào mạch trình[8] (thread) chính của chương trình. Trong vài trường hợp, thay vào đó chương trình yêu cầu được đặt vào cùng đường đua như với vài

[7] Từ này giờ đây rất được phổ biến trong phân tích. Nó giống như một "qui trình sản xuất" trong sản xuất công nghiệp hoặc chế biến thực phẩm.
[8] Có người dịch "thread" là tiểu trình. Chúng tôi dịch là "mach trình", với chữ mạch ở đây là "kinh mạch" trong đông y.

đoạn mã đang chạy khác, trong trường hợp này CLR chỉ sẽ tạo một application domain mới cho nó.

Kế tiếp, CLR sẽ lấy phần đầu tiên của đoạn mã cần cho chạy rồi cho biên dịch phần này từ IL qua hợp ngữ, và cho thi hành từ mạch trình thích ứng. Mỗi lần trong quá trình thi hành gặp phải một hàm hành sự mới chưa được thi hành trước đó, thì hàm hành sự này sẽ được biên dịch thành đoạn mã khả thi. Tuy nhiên, việc biên dịch này chỉ thực hiện một lần thôi, và vị chỉ[9] (address) nhập vào hàm hành sự sẽ được thay thế bởi vị chỉ của đoạn mã hàm được biên dịch. Theo thể thức này, hiệu năng thi hành chương trình sẽ được duy trì, vì chỉ một đoạn nhỏ của đoạn mã là được biên dịch mà thôi. Tiến trình biên dịch này được mệnh danh là *JIT compiling* (just-in-time compiling), nghĩa là biên dịch vừa đúng lúc. Việc làm này cũng giống như trong một nhà hàng đặc sản, khách hàng có thể chỉ vào một con rắn đang còn sống chẳng hạn, rồi yêu cầu làm món rắn xào lăn, lẩu rắn chẳng hạn thì đầu bếp sẽ giết ngay con rắn, trích máu mời bạn uống rồi bắt đầu làm các món theo yêu cầu của bạn. Nghĩa là, món ăn chỉ được làm theo yêu cầu vào lúc chót của khách hàng, chứ không làm sẵn trước.

Khi đoạn mã đang chạy, thì CLR sẽ điều khiển việc sử dụng ký ức. Vào một lúc nào đó, việc thi hành tạm ngưng trong một thời gian rất ngắn (vài millisecond) và bộ phận hốt rác (garbage collector, tắt là GC) được gọi vào; bộ phận này sẽ xem xét những biến trong chương trình để xác định xem vùng ký ức nào đang còn được sử dụng, vùng nào không, như vậy bộ phận GC sẽ thu hồi lại ký ức nào bị chiếm dụng không còn dùng nữa.

Ngoài ra, CLR sẽ lo thụ lý việc nạp vào những assembly khi được yêu cầu, kể cả những cấu kiện COM thông qua dịch vụ .NET COM Interop Services.

1.3.3 Những lợi điểm của đoạn mã được giám quản (managed code)

Qua những mô tả kể trên, bạn có thể thấy một vài lợi điểm khi cho chạy đoạn mã dưới sự giám sát và quản lý của CLR, (được gọi là managed code):

- Về mặt an toàn, thì quá hiển nhiên. Vì assembly được viết theo IL, nên .NET Runtime có cơ hội kiểm tra mọi thứ mà đoạn mã sẽ thi hành. Lợi điểm thực sự là đoạn mã chạy an toàn hơn. Đơn giản, bạn chỉ cần đề ra chính sách an toàn .NET để bảo đảm là đoạn mã không được phép thực hiện những gì khác với những gì được chờ đợi. .NET Runtime hỗ trợ cả **role-based security** (chế độ an toàn dựa trên sự nhận diện của process lo chạy đoạn mã) lẫn **code-based security** (chế độ an toàn dựa trên mức độ tin cậy của bản thân đoạn mã ít nhiều bao nhiêu).

[9] Chúng tôi không dịch là "địa chỉ" vì đây không phải là đường phố mà là vị trí ký ức.

- Sự hiện diện của bộ phận dịch vụ Garbage Collector (GC) giải phóng bạn khỏi mọi lo lắng phải viết ra những lệnh giải phóng ký ức, tránh việc rò rỉ ký ức (memory leak). Nghĩa là giờ đây, với đoạn mã được giám quản, việc rò rỉ ký ức do các biến "ngồi chơi xơi nước" sẽ triệt để được quản lý bởi GC.

- Khái niệm mới về "application domain" (địa bàn hoạt động của ứng dụng) ám chỉ những ứng dụng khác nhau cần được cô lập với nhau, nhưng cũng cần liên lạc với nhau, thì có thể nên đặt chúng với nhau trong cùng một process, đem lại những thành tích to lớn. Ban có thể hình dung nhiều sân quần vợt được lưới rào chắn, nằm trên cùng một khuôn viên câu lạc bộ. Khuôn viên là process, và sân chơi là application domain.

- Trong thí dụ trên ta có những đoạn mã nguồn được viết theo C# và theo VB .NET. Đây có nghĩa là có một sự liên thông ngôn ngữ trong hoạt động (gọi là interoperability) làm cho việc sử dụng những đoạn mã viết theo những ngôn ngữ khác nhau dễ dàng hơn. Nó giống như hội nhập kinh tế giữa các nước ASEAN vậy.

- Cấu trúc tự mô tả (thông qua metadata) của các assembly sẽ làm cho khó lòng xảy ra bug do phiên bản khác nhau cùng một chương trình gây ra, một đau đầu đối với ngôn ngữ đi trước.

- Có một điểm ta chưa nhắc đến đó là các lớp cơ bản của .NET. .NET cung cấp một mô hình đối tượng (object model) rất đơn giản, mang tính trực giác rất cao làm cho việc viết các chức năng Windows dễ dàng hơn trước nhiều, nó đơn giản hoá rất nhiều trong đoạn mã.

1.4 Intermediate Language (IL)

Chúng tôi đã bảo là Microsoft bắt chước ý niệm Java byte code (JBC) của Java để tạo ra Intermediate Language (ngôn ngữ trung gian), rồi dán cái "mác" MSIL (Microsoft Intermediate Language), rồi gọi ngắn gọn là IL. Thật ra, IL cũng như JBC đều thuộc loại ngôn ngữ cấp thấp với một cú pháp đơn giản (dựa trên mã kiểu số thay vì kiểu văn bản), có thể được dịch nhanh và dễ dàng qua mã máy "nguyên sinh" (native machine code). Ý ban đầu của JBC là cung cấp một ngôn ngữ độc lập với sàn diễn (platform indepen dence), nghĩa là đoạn mã gồm chỉ thị byte code có thể đem sử dụng trên bất cứ sàn diễn nào cũng được, cho dù là UNIX, Linux hoặc Windows, và vào lúc chạy ở giai đoạn chót của tiến trình biên dịch, có thể đưa đoạn mã chạy trên một sàn diễn đặc biệt nào đó.

Đây có nghĩa là bạn đơn giản viết một đoạn mã nguồn, cho biên dịch thành JBC rồi chắc chắn như đinh đóng cột là nó có thể chạy trên bất cứ hệ điều hành nào cũng được.

IL cũng mang ý niệm như trên, nhưng lại có tham vọng lớn hơn. IL thuộc loại biên dịch vừa đúng lúc (just-in-time compilation) trong khi JBC thì thuộc loại diễn dịch (interpreted). Đây có nghĩa là những mất mát về hiệu năng do diễn dịch JBC sẽ không áp dụng cho IL. Nói tóm lại *đoạn mã của IL sẽ được biên dịch qua mã máy nguyên sinh chứ không được diễn dịch.*

Mục tiêu của IL không chỉ đơn thuần đem lại tính độc lập về sàn diễn, mà còn là độc lập về ngôn ngữ (language independence) trong một môi trường thiên đối tượng. Ý là bạn có thể biên dịch thành IL từ bất cứ ngôn ngữ nào, rồi đoạn mã được biên dịch có thể đem sử dụng phối hợp liên hoàn với một đoạn mã nào đó được biên dịch từ một ngôn ngữ khác. Tập tin IL mà C# tạo ra cũng *tương tự* như tập tin IL do các ngôn ngữ .NET khác tạo ra. Ta gọi việc này là liên thông ngôn ngữ (language interoperability), khả năng hoạt động liên thông của ngôn ngữ. Một ý niệm tương tự như hội nhập kinh tế khu vực hoặc toàn cầu về mặt hải quan, văn hoá, du lịch v.v...

Khi thiết kế IL, Microsoft đã chọn một hướng đi là theo phương pháp lập trình thiên đối tượng cổ điển dùng đến *lớp* (class) và *tính kế thừa* (inheritance). Đây có nghĩa là lớp và tính kế thừa được định nghĩa trong lòng IL.

Trong phần kế tiếp chúng tôi sẽ duyệt qua những đặc tính của IL mà chúng tôi tóm lược sau đây:

- Lập trình thiên đối tượng[10] với thừa kế đơn (single inheritance) trong việc thiết đặt các lớp.
- Giao diện (interface).
- Các dữ liệu kiểu trị và kiểu qui chiếu (value type & reference type).
- Thụ lý các sai lầm biệt lệ thông qua các lớp cơ bản `System.Exceptions`.
- Hệ thống kiểm tra chặt chẽ kiểu dữ liệu (strong type system).

1.4.1 Lập trình thiên đối tượng cổ điển

Ý niệm của lập trình thiên đối tượng là chia đoạn mã của bạn thành những lớp (class). Từ "lớp" ở đây nên hiểu tương tự như từ "giai cấp" (giống như "giai cấp vô sản") chứ không phải "lớp đất lớp đá" hoặc "lớp học". Mỗi lớp tượng trưng cho một định nghĩa về một cái gì đó được nhận diện như là một thực thể (entity) mà bạn muốn làm gì đó với

[10] Object-oriented programming, chúng tôi dịch là "lập trình thiên đối tượng" chứ không là "lập trình hướng đối tượng".

nó. Thí dụ, trên môi trường Windows, lớp có thể là `ListBox`**,** (tượng trưng cho ô liệt kê) `TextBox` (tượng trưng cho ô văn bản) hoặc `Form` (tượng trưng cho biểu mẫu).

Trong một ý nghĩ nào đó, lớp có thể được xem như là một kiểu dữ liệu (data type), giống như số nguyên (integer) hoặc số thực (floating point number) là một kiểu dữ liệu. Khác biệt ở chỗ là lớp phức tạp hơn nhiều, bao gồm một số kiểu dữ liệu nguyên sinh tập hợp lại với nhau kèm theo một số hàm (được gọi là *hàm hành sự* [11]- method) mà ta có thể triệu gọi vào sử dụng. Ngoài ra, lớp cũng có thể là một kiểu dữ liệu "cây nhà lá vườn" (gọi là custom data type) do lập trình viên tự định nghĩa trong mã nguồn. Lập trình thiên đối tượng hoạt động dựa trên ý niệm chương trình sẽ thực chất chạy bằng cách tạo những thể hiện lớp (instance of class) mà ta gọi nôm na là *đối tượng* (object), rồi cho triệu gọi các hàm hành sự khác nhau đối với các đối tượng, để cuối cùng các đối tượng này sẽ tương tác "loạn xạ" lên với nhau (giống như trên các trận đấu bóng đá), theo ý đồ của người viết chương trình.

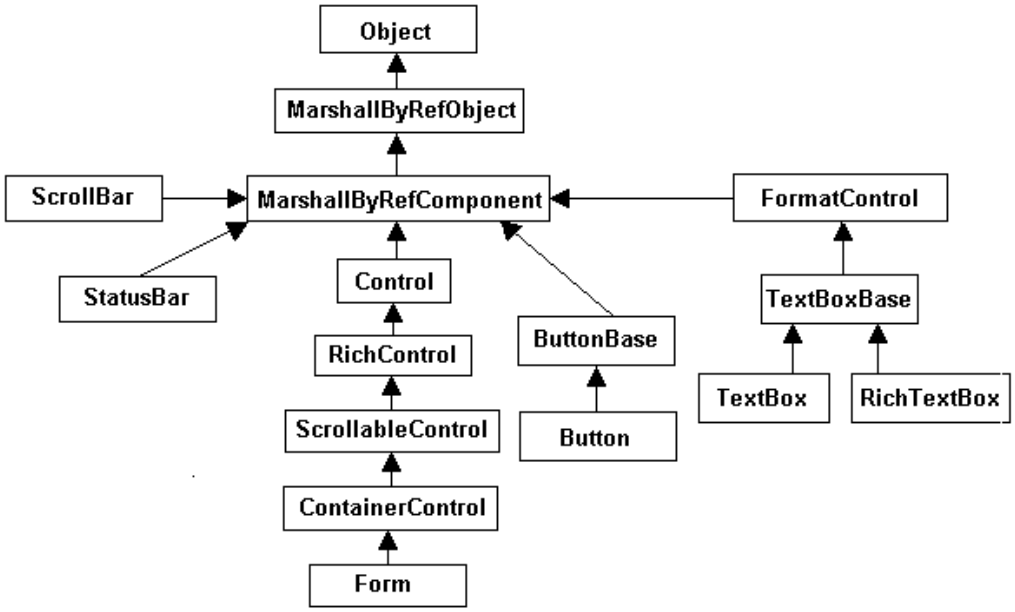
Tuy nhiên, lập trình thiên đối tượng cổ điển gồm nhiều điều hơn là chỉ thế thôi. Nó tùy thuộc nặng nề vào khái niệm *implementation inheritance*, thiết kế thi công kế thừa..

Implementation inheritance là một kỹ thuật cho phép sử dụng lại những chức năng của một lớp đối với một lớp khác. Theo thể thức này, khi định nghĩa một lớp, bạn có thể khai báo lớp này được **kế thừa** (inherited) từ một lớp khác, được biết đến như là **lớp cơ bản** (base class). Như vậy, lớp mà bạn đang định nghĩa (được gọi là **lớp dẫn xuất** – derived class) sẽ tự động thừa hưởng tất cả các hàm hành sự cũng như dữ liệu v.v.. đã được định nghĩa trong lớp cơ bản. Sau đó, bạn có thể bổ sung thêm những hàm hành sự hoặc dữ liệu coi như của riêng của lớp dẫn xuất, hoặc đồng thời bạn có thể cho biết một vài hàm hành sự của lớp cơ bản không phù hợp với lớp dẫn xuất của bạn nên có thể được chỉnh trang thiết kế lại, và ta gọi hàm cơ sở bị điều chỉnh là **hàm hành sự bị phủ quyết** (overriden). Thí dụ, bạn định nghĩa một lớp **NhanVien** trữ thông tin liên quan đến tất cả các nhân viên công ty. Trong lớp này có một hàm hành sự, **TinhLuong**, chuyên lo tính lương hằng tháng cho nhân viên. Tuy nhiên, trong công ty lại có lớp người quản lý, **QuanLy**, cách tính lương có hơi khác một chút so với nhân viên thông thường, nghĩa là phải tính thêm tiền thưởng do chức vụ đem lại. Như vậy, bạn bổ sung một lớp **QuanLy**, được dẫn xuất từ lớp **NhanVien,** nhưng hàm hành sự **TinhLuong**, ở lớp **NhanVien**, phải bị phủ quyết thay thế bởi một hàm hành sự cũng mang tên **TinhLuong** nhưng cách tính lại khác đi, phải tính thêm tiền thưởng chức vụ.

IL hỗ trợ khái niệm **single inheritance** (kế thừa đơn), nghĩa là một lớp nào đó chỉ có thể trực tiếp được dẫn xuất từ một lớp khác mà thôi (kiểu một vợ một chồng, không có kiểu đa thê đa phu). Nhưng lại không giới hạn bao nhiêu lớp sẽ được dẫn xuất từ một lớp nào đó. Điều này ta đi đến một cái cây (tree) giống như cấu trúc các lớp. Trong trường

[11] Chúng tôi dịch "method" là "hàm hành sự" theo kiểu kiếm hiệp, chứ không dùng từ "phương thức" như của một số dịch giả khác.

hợp IL, tất cả các lớp cuối cùng đều được dẫn xuất từ một lớp gốc được mang tên **Object**. Nghĩa là ông cố tổ của tất cả các lớp trên IL mang tên **Object**. Hình 1–3 cho thấy thân cây các lớp cơ bản của .NET. Đây là một cái cây nằm lộn đầu so với cây trong thiên nhiên. Các cây tin học đều lộn đầu như thế, nghĩa là gốc nằm trên cao. Bạn để ý nếu ta lần theo cây xuống dưới, thì các lớp trở thành chuyên hoá (specialized) hơn.



Hình 1-3: Cây các lớp cơ bản

Ngoài lập trình thiên đối tượng, IL còn đưa vào khái niệm về **giao diện** (interface). Thật ra khái niệm này chả có gì mới mẻ vì nó đã được thiết đặt trong COM của Windows. Tuy nhiên, giao diện .NET không giống giao diện COM: nó không cần sự hỗ trợ của bất cứ "hạ tầng cơ sở" COM nào cả, thí dụ nó không được dẫn xuất từ **IUnknown**, và nó không có những mã nhận diện GUID được gắn liền. Tuy nhiên, giao diện trên IL vẫn mượn đỡ khái niệm giao diện COM là giao diện hoạt động như là một "hợp đồng" hoặc "khế ước" (contract) và những lớp thiết đặt một giao diện nào đó phải bảo đảm thiết đặt những hàm hành sự và thuộc tính mà giao diện chỉ định theo hợp đồng.

1.4.2 Dữ liệu kiểu trị và kiểu qui chiếu

Giống với bất cứ ngôn ngữ lập trình nào, IL cung cấp một số kiểu dữ liệu gọi là "bẩm sinh"[12] đã được định nghĩa sẵn trước (mà các sách thường gọi là "built-in type" hoặc "intrinsic type"). Tuy nhiên, một đặc tính khá nổi bật của IL là phân biệt rõ ràng giữa dữ liệu kiểu trị (value type) và dữ liệu kiểu qui chiếu (reference type). Dữ liệu kiểu trị là khi nào biến trực tiếp trữ nội dung dữ liệu, trong khi dữ liệu kiểu qui chiếu thì biến chỉ trữ vị chỉ ký ức (memory address) chỉ về nơi chứa nội dung dữ liệu. Giống như trong một công ty, tiền bạc để tại quỹ của công ty, gọi là tiền mặt (cash), còn tiền để ở ngân hàng gọi là tài khoản (account). Tiền mặt thuộc "value type", còn tài khoản thuộc "reference type". Mô hình này cũng giống như trên Java.

IL phân định rõ ràng: những thể hiện (instance) của dữ liệu kiểu qui chiếu sẽ được trữ trên một vùng ký ức được gọi là heap[13], trong khi dữ liệu kiểu trị thì sẽ được trữ trên vùng ký ức khác mang tên stack. Chúng tôi sẽ giải thích heap và stack trong Chương 4, "Căn bản ngôn ngữ C#".

1.4.3 Kiểm tra chặt chẽ kiểu dữ liệu

Tất cả các biến trên IL đều được đánh dấu thuộc kiểu dữ liệu cụ thể nào đó. Khi biết kiểu dữ liệu, thì IL sẽ không cho phép thực hiện bất cứ tác vụ (operation) nào đi ngược lại đặc tính của kiểu dữ liệu này. Nghĩa là, sẽ không có sự nhập nhằng, ai làm việc nấy.

Lập trình viên C++ ai cũng biết tệ nạn sử dụng con trỏ (pointer) khi ép kiểu dữ liệu, đem lại khá nhiều rủi ro không an toàn. Còn đối với lập trình viên VB, thì việc chuyển đổi dữ liệu từ kiểu này qua kiểu kia quá dễ dãi, nên cũng có thể gây ra một số sai lầm. Do đó, IL đề ra một số biện pháp tăng cường việc kiểm tra kiểu dữ liệu:

- Trước tiên, CLR dựa trên việc có khả năng xem xét đoạn mã để biết loại tác vụ nào đoạn mã cần thực hiện, trước khi có thể cho thi hành. Điều này rất quan trọng cả trên quan điểm cấp phép an toàn lẫn quan điểm kiểm tra liệu xem đoạn mã không thể gây nguy hại cho đoạn mã khác đang thi hành trên một application domain khác, mà ngay trên cùng khoảng không gian ký ức. IL được thiết kế để có thể tiến hành dễ dàng những kiểm tra này.

- Để có thể nhận diện dễ dàng vùng ký ức cần thu hồi, bộ phận garbage collector cần có khả năng nhận diện không nhập nhằng kiểu dữ liệu được trữ trên mỗi vị trí ký ức. Nếu bộ phận này không thể nhận diện kiểu dữ liệu thì khó lòng .NET Runtime có thể hoạt động một cách đúng đắn.

---

[12] Khi bạn vừa mới lọt lòng mẹ, bạn đã có những chức năng "bẩm sinh" là bú, hét, khóc, ỉa, đái v.v.. mà khỏi học tập gì cả. Trong ngôn ngữ lập trình cũng thế, có những kiểu dữ liệu "bẩm sinh" (built-in data type) do nhà sản xuất làm sẵn cho bạn dùng.

[13] heap có nghĩa là một "đụn rơm" ở nhà quê.

- Tính liên thông ngôn ngữ của .NET Framework dựa trên tập hợp những kiểu dữ liệu được định nghĩa rõ ràng và nhất quán.

Hệ thống kiểu dữ liệu mà IL dùng đến sẽ được biết dưới cái tên **Common Type System** (hoặc **Common Type Specification**, tắt là CTS) mà chúng tôi sẽ đề cập sau.

1.5 Các cấu kiện của .NET Framework

Trong phần này chúng tôi sẽ xét đến các cấu kiện[14] (component) còn lại hình thành .NET Framework.

1.5.1 Assembly

Trong phần này, chúng tôi không đi sâu vào chi tiết liên quan đến assembly. Tập II của bộ sách này sẽ đề cập đến vấn đề này. Trước tiên, bạn chớ nhầm lẫn từ assembly với ngôn ngữ assembly, mà ta thường gọi là hợp ngữ. Từ assembly trong thế giới .NET ám chỉ một "giây chuyền lắp ráp". Assembly là lắp ráp. Tạm thời chúng tôi không dịch từ này.

Assembly là một đơn vị lô gic chứa đoạn mã hoạt động theo kiểu .NET; theo định nghĩa này nó cũng giống như một DLL hoặc một tập tin .EXE, hoặc một tập tin chứa các cấu kiện COM. Assembly hoàn toàn tự mô tả được mình; tuy nhiên, nó cũng là một đơn vị lô gic thay vì vật lý, nghĩa là có thể trữ assembly trên nhiều tập tin (và các tập tin này có thể nằm rải rác đâu đó trên đĩa), và như vậy sẽ có một tập tin là trụ cột chứa điểm đột nhập, và mô tả những tập tin khác thuộc assembly.

Bạn để ý cũng cấu trúc assembly này sẽ được sử dụng trong cả đoạn mã khả thi (.EXE) lẫn đoạn mã thư viện (.DLL). Chỉ khác ở một điểm là trên assembly khả thi có một điểm đột nhập trong khi assembly thư viện lại không có.

1.5.1.1 Metadata và Manifest

Đặc tính quan trọng nhất của một assembly là nó chứa metadata (siêu dữ liệu) mô tả những kiểu dữ liệu và hàm hành sự được định nghĩa trong đoạn mã tương ứng cũng như chứa "assembly metadata" mô tả bản thân assembly. Loại thông tin "assembly metadata " này được trữ trên một vùng ký ức mang tên là **manifest**, cho phép kiểm tra phiên bản của assembly cũng như tính toàn vẹn dữ liệu (data integrity, nghĩa là không bị giả mạo). Từ

[14] Component phải suy nghĩ theo kiểu những "linh kiện" trong một bộ phận máy móc. Cấu kiện là những linh kiện cấu thành một cái gì đó.

"**manifest"** mượn của ngành hàng không, hàng hải: một manifest là danh sách các khách hàng trên chuyến bay (gọi là không vận đơn), hoặc danh sách các mặt hàng bốc xuống tàu (hải vận đơn).

Việc assembly chứa metada liên quan đến chương trình có nghĩa là những ứng dụng hoặc các assembly khác triệu gọi đoạn mã của một assembly nào đó sẽ không cần đến registry hoặc bất cứ nguồn lực dữ liệu nào khác, để có thể tìm ra cách sử dụng assembly. Đây là cách tách rời dứt khoát khỏi tập tục làm việc của COM với những mã nhận diện GUID các cấu kiện và giao diện được lấy từ registry, và trong nhiều trường hợp, những chi tiết liên quan đến các hàm hành sự và thuộc tính được trưng ra lại cần phải đọc từ một thư viện kiểu dữ liệu (type library).

Thay vì dữ liệu được rải ra nhiều nơi khác nhau như trên COM, với nguy cơ thiếu đồng bộ, thì với assembly không có rủi ro này, vì tất cả các metadata được trữ cùng với những chỉ thị của chương trình khả thi. Cho dù assembly được trữ trên nhiều tập tin, việc mất đồng bộ dữ liệu sẽ không xảy ra, vì tập tin chứa điểm đột nhập của assembly có trữ tất cả những chi tiết kể cả một đoạn mã băm (hash code) liên quan đến nội dung của các tập tin khác; có nghĩa là nếu một trong những tập tin thuộc assembly bị sửa đổi hoặc bị tráo đổi thì sẽ bị phát hiện ngay lập tức, và assembly sẽ không được nạp vào.

.

1.5.1.2 Assembly được chia sẻ sử dụng hoặc riêng tư

Assembly được phân làm hai loại: shared assembly và private assembly.

1.5.1.2.1 Private assembly

Private assembly là assembly đơn giản nhất, thường được gởi đính kèm theo phần mềm nào đó, với ý đồ chỉ đem sử dụng kèm với phần mềm này thôi. Nghĩa là bạn cung cấp một ứng dụng dưới dạng một tập tin khả thi kèm theo một số thư viện, theo đấy thư viện chứa đoạn mã chỉ được đem dùng với ứng dụng này mà thôi.

Hệ thống bảo đảm là private assembly không được dùng bởi các phần mềm khác, vì một ứng dụng chỉ có thể nạp private assembly hiện được trữ trên cùng thư mục với chương trình khả thi chính vừa được nạp vào.

Vì private assembly chỉ được dùng với phần mềm mà nó nhắm tới, nên sẽ không có việc các phần mềm khác sẽ viết chồng lên một trong những assembly của bạn với một phiên bản mới, do đó bạn không cần đến những biện pháp an toàn nghiêm ngặt. Ngoài ra, cũng sẽ không có vấn đề choảng nhau về tên, vì một ứng dụng chỉ có thể nhìn thấy một lô private assembly mà thôi.

Vì private assembly hoàn toàn "tự chứa lấy" (self-contained), nên tiến trình cài đặt cũng rất đơn giản. Ban chỉ cần đưa những tập tin thích hợp vào thư mục thích hợp trên file system. Ban sẽ không tạo mục vào Registry nào cả. Tiến trình cài đặt này được gọi là **Zero Impact Installation**.

1.5.1.2.2 Shared assembly

Shared assembly (assembly được chia sẻ sử dụng) là những assembly được thiết kế dùng làm thư viện chung cho bất cứ ứng dụng nào có thể sử dụng được.

Vì bất cứ phần mềm nào cũng có thể truy xuất shared assembly, nên phải có những biện pháp tránh những rủi ro sau đây:

- *Đụng độ choảng nhau về tên* (name collision), liên quan đến shared assembly của một công ty phần mềm khác, thiết đặt những kiểu dữ liệu cùng mang trùng tên với shared assembly của bạn. Theo nguyên tắc, ứng dụng khách hàng có thể truy xuất cùng lúc cả hai shared assembly, nên vấn đề có thể trở nên nghiêm trọng.

- Rủi ro xảy ra khi một *assembly có thể bị viết chồng bởi một phiên bản khác* của cùng assembly; và phiên bản mới lại không tương thích với đoạn mã hiện hành của khách hàng.

Để giải quyết vấn đề trên, người ta cho đặt các shared assembly vào một cây con thư mục đặc biệt trên file system, mang tên là **assembly cache**. Khác với trường hợp private assembly được cài đặt khá đơn giản là chép assembly lên thư mục thích ứng, shared assembly lại phải được cài đặt lên cache, một tiến trình được thực hiện bởi một số trình tiện ích đòi hỏi một số kiểm tra đối với assembly cũng như việc dàn dựng một đẳng cấp thư mục nhỏ trong lòng assembly cache để bảo đảm tính toàn vẹn dữ liệu của assembly.

Để tránh đụng độ về tên, người ta gán cho shared assembly một cái tên dựa trên private key cryptography (mật mã hóa mục khoá riêng). (bạn để ý là private assembly được mang cùng tên với tên tập tin chính). Tên này được biết là **strong name** (tên nặng ký), bảo đảm là duy nhất và phải được trình ra bởi ứng dụng khi muốn qui chiếu về một shared assembly.

Các vấn đề do viết chồng lên một shared assembly sẽ được giải quyết bằng cách khai báo thông tin phiên bản trong manifest, và bằng cách cho phép cài đặt cạnh nhau hai phiên bản.

1.5.2 Namespaces

Namespace ("địa bàn hoạt động của các tên") là cách mà .NET tránh né việc các tên (tên lớp, tên biến, tên hàm, v.v..) choảng nhau vì trùng tên giữa các lớp. Thí dụ, bạn ở trong trường hợp bạn định nghĩa một lớp tượng trưng cho khách hàng, bạn đặt cho lớp một tên cúng cơm là **Customer**, theo tiếng Anh cho có vẻ toàn cầu hoá, và một người khác cũng làm như thế. Giống như việc thương hiệu cà phê Trung Nguyên đã được đăng ký nhãn hiệu hàng hoá ở Hoa Kỳ và ở VN, nhưng lại thuộc hai chủ thể khác nhau.

Namespace chẳng qua cũng chỉ là một việc gộp lại những kiểu dữ liệu có liên hệ với nhau về mặt ngữ nghĩa (lớp, enumeration, giao diện, ủy thác và cấu trúc) dưới một tán dù, nhưng nó có tác dụng là tên tất cả các kiểu dữ liệu trong phạm vi namespace sẽ tự động được gán một tiền tố (prefix) mang tên namespace. Ta có thể cho các namespace nằm lồng nhau cái này trong cái kia. Thí dụ tất cả các lớp cơ bản thuộc mục đích chung (general purpose) sẽ nằm trong namespace **System**. Lớp cơ bản **Array** nằm trong **System**, như vậy tên trọn vẹn của nó là **System.Array**. Giữa System và Array có một dấu chấm (.).

.NET đòi hỏi tất cả các kiểu dữ liệu phải được định nghĩa trong một namespace. Như vậy, trong thí dụ Customer kể trên, bạn có thể đặt lớp này trong namespace **Your CompanyName** chẳng hạn, và lúc này tên trọn vẹn của lớp sẽ là **YourCompany Name.Customer.**

Microsoft khuyên là bạn nên cung cấp ít nhất hai namespace nằm lồng nhau, namespace đầu tiên là tên công ty, **YourCompanyName**, namespace thứ hai là tên công nghệ, hoặc package phần mềm, **SalesServices** chẳng hạn. Như vậy sẽ bảo vệ bạn trong đa số trường hợp, tránh việc choảng nhau vì tên đặt ra bởi những tổ chức khác.

Trong phần lớn các ngôn ngữ nguồn, namespace có thể được khai báo đơn giản trong mã nguồn. Thí dụ, trong C#, cú pháp sẽ như sau:

```
namespace YourCompanyName.SalesServices
{
    class Customer
    // v.v..
}
```

1.5.2.1 Một vòng rảo qua .NET Namespace

Phần lớn những namespace quan trọng đều xoay quanh **System.** Namespace này cung cấp phần thân cốt lõi các kiểu dữ liệu. Thật thế, bạn không tài nào xây dựng bất cứ ứng dụng C# hoạt động được mà không qui chiếu ít nhất về namespace **System**. Ban xem Bảng 1-1:

Bảng 1-1: Bảng liệt kê một số .NET namespace thông dụng nhất.

.NET Namespace	Ý nghĩa
System	Trong lòng SYSTEM bạn sẽ tìm thấy vô số lớp cốt lõi liên quan đến các kiểu dữ liệu bẩm sinh, tính toán số học, thu lượm rác, v.v..
System.Collections	Namespace này định nghĩa một số đối tượng "thùng chứa" (container object), chẳng hạn **ArrayList, Queue, SortedList.**
System.Data **System.Data.Common** **System.Data.OleDb** **System.Data.SqlClient**	Những namespace này được dùng trong việc khai thác các căn cứ dữ liệu. Ban sẽ nghiên cứu các namespace này trong tập 4 bộ sách này.
System.Diagnostics	Tại đây, bạn có thể tìm thấy vô số kiểu dữ liệu có thể được dùng bởi bất cứ ngôn ngữ "ăn ý" với .NET (.NET aware) để gỡ rối (debug) và theo dõi việc thi hành mã nguồn.
System.Drawing **System.Drawing.Drawing2D** **System.Drawing.Printing**	Tại đây bạn tìm thấy vô số những kiểu dữ liệu về vẽ đồ họa, gói ghém các kiểu dữ liệu bẩm sinh của GDI+ (chẳng hạn bitmaps, fonts, icons) cũng như hỗ trợ việc in ấn, và những lớp tô vẽ (rendering)
System.IO	Namespace này đầy những kiểu dữ liệu lo việc xuất nhập dữ liệu bao gồm file I/O, buffering, v.v..
System.Net	Namespace này (kể cả những namespace có liên hệ khác) bao gồm những kiểu dữ liệu liên quan đến lập trình mạng.
System.Reflection **System.Reflection.Emit**	Namespace này định nghĩa việc hỗ trợ phát hiện kiểu dữ liệu vào lúc chạy, cũng như tạo và triệu gọi linh động các kiểu dữ liệu "cây nhà lá vườn" (custom type).
System.Runtime.InteropServices **System.Runtime.Remoting**	Namespace này cung cấp những tiện nghi cho phép tương tác với "mã vô quản" (unmanaged code, chẳng hạn Win32 DLL, COM server, v.v..), và những kiểu dữ liệu dùng truy xuất dữ liệu từ xa.
System.Security	An toàn là một khía cạnh tổng hợp của thế giới .NET. Tại đây bạn có thể tìm thấy vô số lớp liên quan đến

	việc cấp phép, mật mã hoá (cryptography) v.v..
System.Threading	Namespace có liên hệ với mạch trình [15](thread). Tại đây, bạn sẽ tìm thấy những kiểu dữ liệu như **Mutex, Thread** và **Timeout.**
System.Web	Một số namespace hướng về việc triển khai những ứng dụng Web, kể cả ASP.NET.
System.Windows.Form	Mặc dù tên gọi, sàn diễn .NET không chứa những namespace cho phép xây dựng dễ dàng những cửa sổ chính Win32 cỗ điển, những khung đối thoại cũng như những ô control mà người ta thường gọi là những thứ "lục lăng lục chốt" (widget) kiểu "cây nhà lá vườn".
System.Xml	Namespace này chứa vô số lớp tượng trưng cho những kiểu dữ liệu cốt lõi của XML và những kiểu dữ liệu tương tác với dữ liệu XML.

Việc tìm hiểu từng kiểu dữ liệu được chứa đựng trong mỗi namespace đòi hỏi thời gian và kinh nghiệm. Sau đây là một số "ứng viên" mà bạn cần qui chiếu trong chương trình của bạn:

```
// sau đây là những namespace dùng xây dựng ứng dụng
using System;  // các kiểu dữ liệu thuộc general base class library
using System.Drawing; // kiểu dữ liệu đồ họa
using System.Widows.Forms; // các kiểu dữ liệu GUI
using System.Data;  // kiểu dữ liệu khai thác dữ liệu căn cứ dữ liệu
using System.Data.OleDb; // kiểu dữ liệu truy xuất OLE DB
```

Một khi bạn đã qui chiếu một vài namespace, bạn tự do tạo ra những thể hiện (instance) của những kiểu dữ liệu mà namespace chứa đựng. Thí dụ, bạn muốn tạo một thể hiện của lớp **Bitmap** (được định nghĩa trong **System.Drawing**), bạn có thể viết như sau:

```
// Liệt kê rõ ra tên Namespace
using System.Drawing;

class MyClass
{
   public void DoIt()
   {
```

[15] Có người dịch là "tiểu trình".

```
        // Tạo một bitmap 20 * 20 pixel
        Bitmap bm = new Bitmap(20, 20);
        …
    }
}
```

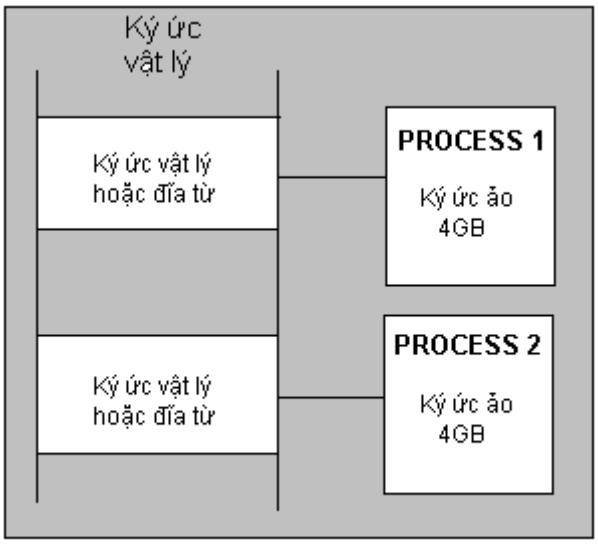
Vì ứng dụng của bạn qui chiếu namespace **System.Drawing**, nên trình biên dịch có thể giải quyết lớp **Bitmap** như là thành viên của namespace này. Nếu bạn không qui chiếu namespace trên, thì trình biên dịch sẽ phát ra thông điệp sai lầm. Tuy nhiên, bạn hoàn toàn tự do khai báo biến sử dụng *tên chính danh trọn vẹn* (fully qualified name) như sau, cũng với thí dụ kể trên:

```
// Không liệt kê rõ ra tên Namespace

class MyClass
{
    public void DoIt()
    {
        // Tạo một bitmap 20 * 20 pixel
        // Sử dụng tên chính danh trọn vẹn
        System.Drawing.Bitmap bm = new System.Drawing.Bitmap(20, 20);
        …
    }
}
```

1.5.3 Application Domain

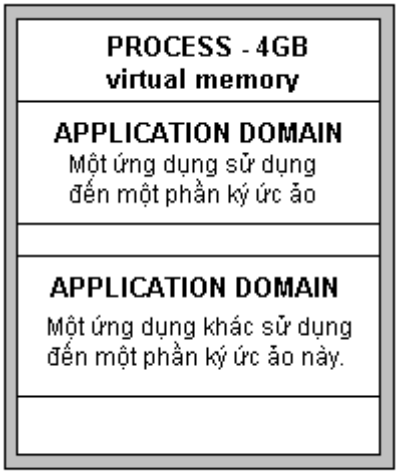
Application domain ("địa bàn hoạt động" của ứng dụng, giống như namespace là "địa bàn hoạt động" của các tên) là một cải tiến quan trọng mà .NET đem lại trước việc tiêu hao ký ức (gọi là overhead) khá nhiều để bảo đảm các ứng dụng phải được cách ly (isolation) hoàn toàn với nhau, nhưng vẫn có thể liên lạc với nhau. Ban cứ hình dung hai sân quần vợt nằm kế cận, và cả hai đều có người chơi. Vấn đề là quả bóng của bên này có thể nhảy qua bên kia, phá rối cuộc chơi. Phải làm thế nào cách ly hai sân chơi.

Mãi tới giờ, biện pháp duy nhất để cách ly đoạn mã là thông qua process. Khi bạn khởi động một ứng dụng, thì ứng dụng này chạy trong phạm vi một process (giống như một đường đua việt dã hoặc một giây chuyền sản xuất). Windows cách ly các process với nhau thông qua "khoảng không gian vị chỉ" (address space) - giống như rào lưới rất cao xung quanh sân quần vợt. Mỗi process sẽ được cấp 4 gigabytes ký ức ảo (virtual memory), để trữ dữ liệu và chương trình. Ký ức này là ảo vì thực thụ nó không nhận đủ ký ức chính vật lý hiện hành mà chỉ ánh xạ lên một phần ký ức loại này, phần còn lại ánh xạ lên đĩa từ. Việc ánh xạ này phải làm thế nào ký ức chính vật lý của các ứng dụng không được phủ chồng lên nhau. Hình 1-4 cho thấy tình trạng kể trên:



Hình 1-4: Ký ức ảo và process

Mặc dù process đem lại sự an toàn khi chương trình đang hoạt động (không cho viết đè chồng lên ký ức của process khác và cung cấp giới hạn đối với cấp phép hoạt động an toàn). Tuy nhiên, bất lợi chính của process là hiệu năng. Việc phần lớn các process làm việc với nhau là chuyện thường xuyên xảy ra, do đó chúng cần liên lạc với nhau. Lấy một thí dụ hiển nhiên: có một process triệu gọi cấu kiện COM, là một đoạn mã khả thi, và do đó phải chạy trên một process riêng của mình. Vì process không thể chia sẻ sử dụng bất cứ ký ức nào, một tiến trình phức tạp gọi là *marshalling*[16] phải được thực hiện để có thể sao chép dữ liệu giữa hai process. Điều này làm cho hiệu năng bị giảm thảm hại. Nếu bạn muốn các cấu kiện làm việc với nhau và không muốn hiệu năng giảm sút, thì đành chấp nhận sử dụng những cấu kiện kiểu DLL và cho mọi cấu kiện cùng chạy trong cùng một process, với những rủi ro tiềm ẩn do việc một cấu kiện nào đó có thể hành động theo kiểu du đảng, đưa đến kết quả là chương trình sẽ bị "sụm bà chè".



Hình 1-5 : Application Domain.

Thông thường, bất cứ process nào chỉ có thể truy xuất ký ức bằng cách khai báo một vị chỉ ký ức ảo mà thôi, process không thể truy xuất trực tiếp vào vị chỉ vật lý. Do đó, khó lòng một process truy xuất ký ức được cấp phát cho một process khác. Như vậy bảo đảm là bất cứ đoạn mã "du côn du đảng" nào cũng không thể "quậy phá" ngoài địa bàn của mình.

Ngoài ra, process còn có riêng cho mình một security token (thẻ bài an ninh), một biện pháp phòng ngừa bảo đảm an toàn, cho Windows biết một cách chính xác tác vụ nào process được phép thi hành.

Application Domain được thiết kế như là một

[16] Marshalling có thể hình dung như là việc "vô bao vô thùng' các hàng hoá phải gởi đi qua bưu điện hoặc hàng không chẳng hạn.

cách để cách ly các cấu kiện với nhau, nhưng không giảm thiểu hiệu năng hoạt động do việc trao đổi dữ liệu giữa các process. Ý niệm nằm đằng sau là chia bất cứ process nào thành một số "địa bàn hoạt động của ứng dụng" (application domain). Mỗi application domain tương ứng đối với một ứng dụng đơn lẻ, và mỗi mạch trình (thread) thi hành sẽ được chạy trong một application domain đặc biệt nào đó. Xem hình 1-5.

Nếu những đoạn mã khả thi khác nhau đều chạy trong cùng không gian ký ức của process, rõ ràng là chúng có thể chia sẻ sử dụng dữ liệu vì theo lý thuyết chúng có thể thấy dữ liệu của nhau. Mặc dù có khả năng như thế, nhưng .NET runtime bảo đảm là sẽ không xảy ra trong thực tế thông qua việc kiểm tra đoạn mã của mỗi ứng dụng đang chạy bảo đảm là đoạn mã không thể chạy ngoài vùng dữ liệu được cấp phát riêng cho mình.

Nhờ việc kiểm tra chặt chẽ về kiểu dữ liệu của IL, nên các điều kể trên có thể thực hiện được. Trong đa số trường hợp, kiểu dữ liệu được sử dụng sẽ bảo đảm là ký ức sẽ không được truy xuất một cách bất hợp lệ. Thí dụ, kiểu dữ liệu bản dãy (array) của .NET bảo đảm kiểm tra giới hạn thấp và cao (lower & upper bound) không cho phép bản dãy hoạt động quá giới hạn.

Nếu một ứng dụng đang chạy, đặc biệt cần liên lạc hoặc trao đổi dữ liệu với những ứng dụng khác đang chạy trên application domain khác, thì nó phải triệu gọi dịch vụ **.NET remoting** thông qua những lớp cơ bản .NET thuộc namespace **System.Remoting**.

Đoạn mã nào được kiểm tra là nó không thể truy xuất dữ liệu nằm ngoài application domain được gọi là **type safe** (hoặc memory type safe), an toàn về kiểu dữ liệu. Loại đoạn mã như thế có thể chạy cùng với các đoạn mã type safe khác trên các application domain khác trên cùng một process.

1.5.4 Trình biên dịch JIT

Trình biên dịch JIT (Just-In-Time compiler) là một phần chủ chốt của .NET Frame work, bảo đảm các đoạn mã được giám quản (managed code) sẽ đem lại hiệu năng cao so với những đoạn mã vô quản (unmanaged code).

Bạn nhớ cho Java byte code (JBC) được diễn dịch (interpreted) còn IL thì lại được biên dịch. Không những chỉ thế, thay vì biên dịch toàn bộ ứng dụng trong một phát, trình biên dịch JIT chỉ biên dịch một phân đoạn (do đó chữ just-in-time, vừa đúng lúc) khi được triệu gọi (một hàm chẳng hạn). Khi phân đoạn mã được biên dịch lần đầu tiên, nó sẽ được trữ cho tới khi chương trình thoát khỏi, như vậy nó sẽ không cần được biên dịch lại lần nữa khi phân đoạn này được triệu gọi vào lần nữa. Microsoft cho rằng như thế hoạt động của chương trình sẽ có hiệu năng hơn là cho biên dịch toàn bộ chương trình (vì có thể một phần lớn chương trình không bao giờ được dùng đến trong chấu làm việc).

Điều này giải thích vì sao ta có thể chờ đợi là việc thi hành đoạn mã IL sẽ hầu như nhanh bằng việc thi hành đoạn mã máy nguyên sinh. Lý do vì sao việc biên dịch chỉ vào lúc chương trình chạy, nên JIT compiler sẽ biết một cách chính xác là loại processor nào sẽ chạy, nên JIT compiler có thể lợi dụng những chức năng đặc biệt của bộ chỉ thị của processor để tối ưu hoá đoạn mã khả thi. Nghĩa là JIT compiler sẽ biết sẽ chạy trên x86 processor hoặc trên Alpha processor. Thí dụ, Visual Studio 6 tối ưu hoá đối với máy Pentium, do đó đoạn mã được kết sinh không thể hưởng những lợi điểm mà các chức năng phần cứng của processor Pentium III đem lại. Nói cách khác, JIT compiler có thể thực hiện tất cả các tối ưu mà Visual Studio 6 làm được, nhung ngoài ra, nó có thể tối ưu hoá dựa theo những đặc điểm của một processor đặc biệt mà đoạn mã đang chạy.

1.5.5 .NET Tools

Ngoài những dịch vụ runtime, .NET còn có sẵn những công cụ hỗ trợ việc triển khai các ứng dụng. Các công cụ này bao gồm:

- Visual Studio .NET, một môi trường triển khai tổng thể (IDE, Integrated Development Environment) cho phép bạn viết đoạn mã, biên dịch, gỡ rối dựa trên tất cả các ngôn ngữ của .NET, chẳng hạn C#, VB .NET, managed C++, kể cả những trang ASP.NET và unmanaged C++. Trong tập 2, bộ sách này, chúng tôi sẽ đề cập đến Visual Studio .NET IDE

- Trình biên dịch chạy theo dòng lệnh (command-line compiler) đối với các ngôn ngữ C#, VB .NET và C++.

- ILDASM, một trình tiện ích Windows có thể được đem sử dụng để quan sát nội dung của một assembly bao gồm manifest và metadata.

1.5.6 Garbage Collector

Bộ phận dịch vụ thu hồi ký ức bị chiếm dụng đã hết hạn sử dụng, gọi là Garbage Collector (thu gom rác, viết tắt GC) được xem như là cách quản lý ký ức. Mãi tới nay, trên sàn diễn Windows, có hai thể thức được dùng đến để giải phóng ký ức mà các process đã yêu cầu từ hệ thống: đó là yêu cầu đoạn mã ứng dụng giải phóng bằng tay các ký ức được cấp phát khi đã xong việc, và yêu cầu các đối tượng duy trì một cái đếm, gọi là *count reference* cho biết đối tượng có còn được qui chiếu bởi các hàm hành sự hay không.

Việc giao trách nhiệm giải phóng ký ức là kỹ thuật được sử dụng bởi ngôn ngữ cấp thấp, như C++ chẳng hạn. Nó hiệu quả, với lợi điểm là các nguồn lực (resource) không bị chiếm dụng quá lâu hơn cần thiết. Tuy nhiên, bất lợi lớn nhất là thường xuyên có bug. Đoạn mã nào yêu cầu hệ thống cấp phát ký ức buộc phải báo cho biết rõ ra là không cần

đến ký ức nữa vì đã xong công việc. (Giống như bảo công chức khi về hưu thì thông báo cho sở nhà đất thu hồi lại nhà công vụ mình đang ở). C++ có từ chốt **delete** để làm công việc này. Tuy nhiên, lập trình viên phải rất cẩn thận bảo đảm là phải giải phóng tất cả các ký ức mà y đã yêu cầu được cấp phát trước đó. Nhưng trong thực tế, thì việc quên giải phóng ký ức này khá phổ biến, kết quả là tới một lúc nào đó ký ức bị cạn kiệt do việc rò rỉ ký ức (memory leak) này. Việc rò rỉ ký ức rất khó phát hiện.

Còn việc duy trì một cái đếm qui chiếu là điều ưa thích bởi những đối tượng COM. Mỗi đối tượng COM duy trì một cái đếm để biết bao nhiêu khách hàng hiện đang qui chiếu về đối tượng, nghĩa là bao nhiêu hàm đang sử dụng đối tượng COM. Mỗi lần một khách hàng thôi qui chiếu, thì cái đếm này bị trừ đi 1. Đến khi nào cái đếm trở thành zero, thì đối tượng COM sẽ bị hủy và ký ức nó chiếm dụng trước đó sẽ bị thu hồi. Vấn đề ở chỗ là thiện chí của khách hàng báo cho đối tượng COM biết là họ đã xong việc với đối tượng (và gọi hàm hành sự **IUnknown.Release()**). Nếu chỉ một khách hàng không tuân thủ việc giải phóng ký ức, thì đối tượng nằm chơi xơi nước chiếm dụng ký ức dài dài. Giống như bên ta, cán bộ về hưu vẫn không chịu trả nhà công vụ lại cho nhà nước.

Do đó, .NET tung ra chiêu Garbage Collector (GC) để giải quyết vấn đề.

Đây là một chương trình lo thu hồi ký ức. Tất cả các yêu cầu cấp phát ký ức đều được nhận ký ức từ heap. Thỉnh thoảng, khi .NET phát hiện rằng heap cho một process nào đó trở nên đầy và như vậy sẽ có nhu cầu ký ức trống một cách bức xúc, và lúc này nó gọi GC vào cuộc. GC sẽ rảo qua các biến hiện đang trong phạm vi đoạn mã của bạn, xem xét những qui chiếu về các đối tượng được trữ trên heap để xem những đối tượng nào được truy xuất từ đoạn mã của bạn, nghĩa là những đối tượng nào có những qui chiếu chĩa về chúng. Bất cứ đối tượng nào không còn được qui chiếu bởi bất cứ đối tượng khác, thì lúc ấy nó trở thành ứng viên bị xúc đi.

Muốn biết trong thực tế, việc xảy ra thế nào, ta thử lấy một thí dụ, một đoạn mã con C#. Bạn chưa học cú pháp C#, nhưng qua phần giải thích (sau hai dấu '/') bạn cũng có thể hiểu phần nào công việc:

```
{
   TextBox UserInputArea; // khai báo một đối tượng kiểu TextBox
   UserInputArea = new TextBox(); // thể hiện một đối tượng
   TextBox txtBoxCopy = UserInputArea;  // một biến đối tượng khác
   // giả định GC được triệu gọi vào ở đây
   // xử lý gì gì đó
}
// UserInputArea và txtBoxCopy đã ra ngoài phạm vi
// giả định GC được triệu gọi vào ở đây một lần nữa
```

Đoạn mã bắt đầu khai báo một biến đối tượng kiểu **TextBox**. Biến mang tên **UserInputArea**. **TextBox** thuộc dữ liệu kiểu qui chiếu, nên **UserInputArea** chứa một vị chỉ. Lệnh thứ hai, thông qua từ chốt **new** cho hiển lộ (instantiate) một đối tượng **TextBox**

rồi cho trữ nội dung đối tượng lên trên heap, và vị chỉ qui chiếu đối tượng này sẽ được trữ trên biến đối tượng **UserInputArea**. Lệnh thứ ba kế tiếp, nội dung biến đối tượng **UserInputArea** được gán cho một biến đối tượng khác cùng kiểu **TextBox**, mang tên **txtBoxCopy**. Sau lệnh gán này thì cả hai **UserInputArea** và **txtBoxCopy** đều chỉ về cùng một đối tượng. Đối tượng **TextBox** được qui chiếu hai lần: một bởi **UserInputArea**, và một bởi **txtBoxCopy.**

Tới điểm này, nghĩa là sau lệnh gán, ta giả định GC sẽ được triệu gọi vào, GC sẽ nhận ra rằng **TextBox** được qui chiếu bởi cả hai biến **UserInputArea** và **txtBoxCopy**, và nó thấy **TextBox** vẫn đang còn được sử dụng và như vậy phải nằm yên trên heap. GC sẽ không xoá đối tượng - có thể nó dọn dẹp lại heap để tối ưu hoá hiệu năng. Do đó có thể GC sẽ di chuyển chỗ ở của đối tượng **TextBox**, và nó lặng lẽ nhật tu vị chỉ trên biến **UserInputArea** và **txtBoxCopy** phản chiếu việc thay đổi.

Về sau, các biến **UserInputArea** và **txtBoxCopy** cả hai thoát ra khỏi phạm vi (nghĩa là khi gặp dấu ngoặc nghéo đóng "}". Ta lại giả định là GC lại được triệu gọi, nó thấy một vùng ký ức trên heap được dùng để trữ **TextBox**, và nó tìm thấy giữa những biến sử dụng bởi đoạn mã của bạn không còn qui chiếu về **TextBox**. GC đi đến kết luận là **TextBox** không còn cần thiết nữa và như thế cho xoá sổ nó luôn, thu hồi lại "đất đai".

Bạn không thể biết khi nào GC sẽ được triệu gọi vào làm việc. Nó chỉ được gọi vào khi nào **.NET Runtime** thấy là cần thiết. Nếu chương trình của bạn đòi hỏi liên tục ký ức thì GC sẽ thường xuyên được gọi vào để dọn dẹp.

Bạn có thể triệu gọi GC trong đoạn mã của bạn bằng cách sử dụng lớp cơ bản của **System.GC**, ở ngay chỗ bạn vừa kết thúc sử dụng một số khá nhiều biến. Tuy nhiên, trong đa số trường hợp, bạn có thể tin tưởng vào GC.

1.5.7 Biệt lệ[17] (exceptions)

.NET được thiết kế để xử lý dễ dàng các điều kiện sai lầm, bằng cách sử dụng cùng cơ chế dựa trên biệt lệ (exception) như đã được dùng trên C++ hoặc Java.

Chúng tôi sẽ đề cập chi tiết về biệt lệ ở Chương 12, "Thụ lý các biệt lệ". Ở đây, chúng tôi chỉ lướt qua ý niệm về lĩnh vực này. Trong vài vùng trên đoạn mã, ta sẽ có những thường trình lo *thụ lý biệt lệ* (exception handling)[18], mỗi thường trình sẽ thụ lý một điều kiện sai lầm đặc biệt (thí dụ, một tập tin không tìm thấy, hoặc chia một số cho zero,

[17] Exception có người dịch là "ngoại lệ" giống như "trường hợp ngoại lệ" mang tính loại trừ. Đây không phải thế. Đây mang ý nghĩa một trường hợp đặc biệt không nằm trong thông lệ, nên chúng tôi dịch là "biệt lệ".

[18] Bạn để ý, chúng tôi dịch từ "handle" là "thụ lý" giống như ở toà án người ta thụ lý các hồ sơ phạm nhân.

v.v..). Cấu trúc biệt lệ bảo đảm là khi một điều kiện sai lầm xảy ra, việc thi hành sẽ nhảy lập tức qua bộ phận thụ lý biệt lệ, nơi chuyên xử lý điều kiện sai lầm cụ thể.

Ngoài ra, cấu trúc biệt lệ còn cung cấp một phương tiện tiện lợi theo đấy một đối tượng, gọi là **Error object**, đối tượng sai lầm thuộc kiểu dữ liệu **System.Exception**, chứa những chi tiết của điều kiện sai lầm để có thể chuyển qua cho bộ thụ lý biệt lệ xem xét. Thông tin mà đối tượng này cầm giữ có thể chứa một thông điệp thích ứng đối với người sử dụng và những chi tiết cho biết sai lầm xảy ra ở đâu trên đoạn mã.

C# thụ lý biệt lệ bằng cách dùng những khối lệnh **try {}, catch {}** và **finally {}**. .NET sẽ cung cấp những lớp cơ bản tượng trưng cho biệt lệ cho phép tung ra (throw) những đối tượng biệt lệ để thường trình thụ lý biệt lệ có thể xử lý biệt lệ. Và việc thụ lý biệt lệ này được giải quyết một cách nhất quán đối với tất cả các ngôn ngữ chịu "ăn ý" với .NET.

1.5.8 Attributes

Chúng tôi không dịch từ "Attribute" là "thuộc tính", vì trong phạm trù này nó không mang ý nghĩa nào là thuộc tính (property) cả, do đó chúng tôi để yên. Đề nghị bạn hiểu theo công năng mà chúng tôi sẽ giải thích sau đây. **Attribute** là một chức năng trở thành phổ biến trong việc nới rộng C++ trong những năm gần đây. Lập trình viên VB và Java thì chưa hề biết chức năng này. Ý niệm nằm đằng sau attribute là khởi đi nó cung cấp thông tin bổ sung liên quan đến vài mục tin trong chương trình, và thông tin này có thể đưa cho trình biên dịch dùng trong tiến trình biên dịch. Những thí dụ sử dụng attribute trên C# bao gồm việc cho biết mô hình mạch trình (thread model) nào mà mạch trình khởi động phải chạy:

```
[STAThread]
void Main()
{
```

hoặc một attribute có thể dùng đánh dấu là hàm hành sự đã lỗi thời (obsolete):

```
[Obsolete]
public int SomeObsoleteMethod()
{
```

Một hàm hành sự lỗi thời sẽ kết sinh một thông điệp cảnh báo của trình biên dịch, hoặc trong vài trường hợp là sai lầm nếu triệu gọi bởi đoạn mã khác.

Với C#, bạn có thể định nghĩa attribute "cây nhà lá vườn" riêng của bạn trong đoạn mã nguồn, và sẽ được đặt cùng với dữ liệu hàm hành sự đối với những kiểu dữ liệu hoặc hàm hành sự tương ứng. Điều này có thể hữu ích trong mục đích sưu liệu, tại chỗ mà chúng có thể dùng phối hợp với reflection (sẽ được đề cập sau đây) để thực hiện những tác vụ lập trình dựa trên attributes.

Attributes sẽ được đề cập trong Tập 2 của bộ sách này.

1.5.9 Reflection

Trong thế giới .NET, *dịch vụ phản chiếu* (reflection) là tiến trình phát hiện những kiểu dữ liệu vào lúc chương trình đang chạy. Sử dụng dịch vụ reflection, bạn có khả năng nạp một assembly vào lúc chạy, và khám phá ra những thông tin giống như với ILDasm.exe. Thí dụ, bạn có thể có một danh sách của tất cả các kiểu dữ liệu được chứa trong một module nào đó, bao gồm những hàm hành sự, thuộc tính, vùng mục tin và tình huống (event) được định nghĩa đối với một kiểu dữ liệu nào đó. Ngoài ra, bạn cũng có thể khám phá một cách linh động tập hợp những giao diện được hỗ trợ bởi một lớp (hoặc một struct) nào đó, những thông số của một hàm hành sự cũng như những chi tiết khác có liên hệ (lớp cơ bản, thông tin namespace, v.v..).

Muốn hiểu dịch vụ reflection, bạn cần tìm hiểu lớp `Type` (được định nghĩa trong namespace `System`) cũng như lớp `System.Reflection. Ban` sẽ thấy lớp `Type` có chứa một số hàm hành sự cho phép bạn trích ra những thông tin đáng giá liên quan đến kiểu dữ liệu hiện hành mà bạn đang xem xét. Còn `System.Reflection` chứa vô số kiểu dữ liệu liên đới giúp cho việc gắn kết trễ (late binding) và nạp động của assembly dễ dàng hơn.

Reflection sẽ được giải thích trong Tập 2 của bộ sách này.

1.6 .NET Framework

Microsoft .NET không những hỗ trợ tính độc lập trong ngôn ngữ, mà ngay cả trong việc "hội nhập" (integration) ngôn ngữ. Điều này có nghĩa là bạn có thể kế thừa từ các lớp, chặn bắt biệt lệ và tận dụng những lợi thế của tính đa hình (polymorphisme) xuyên qua những ngôn ngữ khác nhau. Điều này trở thành hiện thực nhờ vào một đặc tả mang tên ***Common Type Specification (CTS)*** mà tất cả các cấu kiện .NET phải tuân thủ. Thí dụ, mọi việc trên .NET đều là một đối tượng của một lớp đặc biệt nào đó, và lớp này được dẫn xuất từ một lớp gốc mang tên **System.Object**. CTS hỗ trợ khái niệm tổng quát về lớp, kế thừa, giao diện, ủy thác (delegate, hỗ trợ callback), dữ liệu kiểu qui chiếu và kiểu trị.

Ngoài ra, .NET cho bao gồm một đặc tả khác mang tên ***Common Language Specification (CLS)*** cung cấp một loạt những qui tắc cơ bản cần thiết cho việc hội nhập tất cả các ngôn ngữ. CLS xác định những đòi hỏi tối thiểu để trở thành một ngôn ngữ .NET. Các trình biên dịch tuân thủ các đặc tả của CLS sẽ tạo ra những đối tượng có thể

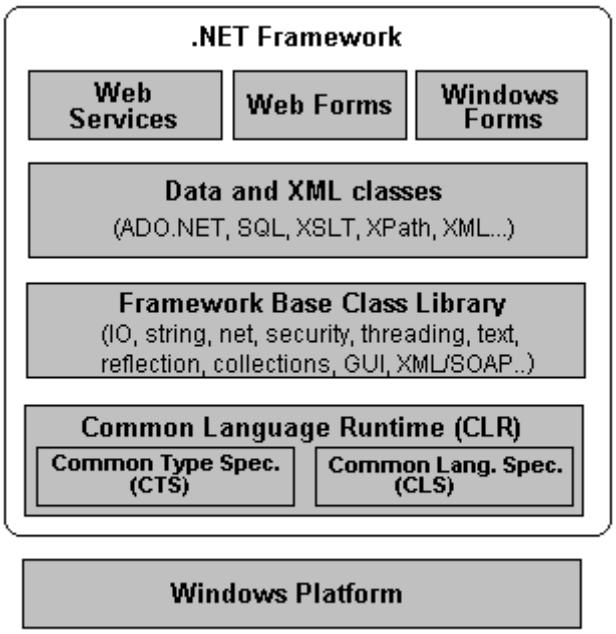
hợp tác với nhau. Toàn bộ thư viện Framework Class Library (FCL) có thể được sử dụng bởi bất cứ ngôn ngữ nào phù hợp với CLS.

.NET Framework nằm trên đầu hệ điều hành (bất cứ là "hương vị" nào của Windows, hoặc của Unix hoặc của bất cứ hệ điều hành nào) và bao gồm một số cấu kiện[19] (component). Hiện thời, .NET Framework gồm có:

- Chính thức 4 ngôn ngữ: C#, VB.NET, J#, Managed C++, và một ngôn ngữ kịch bản JScript .NET.

- **Common Language Runtime** (CLR), một sàn diễn thiên đối tượng đối với việc triển khai phần mềm trên Windows và trên Web mà tất cả các ngôn ngữ kể trên chia sẻ sử dụng.

- Một số thư viện lớp có liên hệ với nhau, nằm dưới cái tên là **Framework Class Library** (FCL).



Hình 1-6: Kiến trúc .NET Framework

Hình 1-6 cho thấy kiến trúc của .NET Framework.

Cấu kiện quan trọng nhất của .NET Framework là CLR, cung cấp môi trường hoạt động đối với chương trình. CLR bao gồm một virtual engine, tương tự như virtual machine của Java trong chừng mực nào đó. Ở cấp cao, CLR khởi động các đối tượng, tiến hành những bước kiểm tra an toàn trên các đối tượng này, cho nạp đối tượng vào ký ức, cho thi hành rồi cuối cùng cho thu hồi ký ức chiếm dụng bởi các đối tượng này nhưng đã hết sử dụng. Ngoài ra, Common Type System (CTS) cũng thuộc thành phần của CLR.

Trên hình 1-6, lớp nằm trên đầu CLR là một tập hợp những lớp cơ bản của khuông giá theo sau một lớp bổ sung liên quan đến các lớp dữ liệu và XML, cộng thêm một lớp dành cho `Web Services`, `Web Forms` và `Windows Forms`. Gộp chung 3 lớp nằm

[19] Có người dịch là "cấu thành".

trên CLR ta gọi là **Framework Class Library** (FCL). Đây là một thư viện lớp lớn nhất trong lịch sử và là thư viện cung cấp một API thiên đối tượng bao gồm tất cả các chức năng mà sàn diễn .NET có thể bao trọn. Với hơn 5000 lớp khác nhau, FCL cung cấp những tiện nghi triển khai nhanh những ứng dụng trên máy để bàn (desktop), client server và các dịch vụ web và ứng dụng khác.

Tập hợp những lớp cơ bản, lớp cấp thấp nhất trên FCL, cũng tương tự như tập hợp lớp trên Java. Những lớp này hỗ trợ những công tác xuất nhập dữ liệu thô sơ, thao tác trên chuỗi chữ, quản lý an toàn, truyền thông mạng, quản lý mạch trình, thao tác văn bản, các chức năng reflection và collection v.v..

Nằm trên lớp FCL, là lớp **Data & XML** nới rộng tầm hoạt động của các lớp cơ bản ở dưới bằng cách hỗ trợ việc quản lý dữ liệu "lưu tồn" [20] (persistent) được trữ trên đĩa cũng như thao tác XML. Data class hỗ trợ việc quản lý dữ liệu được duy trì trên các căn cứ dữ liệu (database)[21] nằm ở "hậu cứ". Những lớp này bao gồm những lớp SQL (Structured Query Language) cho phép bạn thao tác dữ liệu trên các căn cứ dữ liệu thông qua giao diện chuẩn SQL. Ngoài ra, một tập hợp lớp mang tên ADO .NET cho phép bạn thao tác trên dữ liệu được trữ trên các đĩa từ. .NET Framework còn hỗ trợ một số lớp cho phép bạn thao tác dữ liệu XML cũng như tiến hành truy tìm và dịch theo kiểu XML.

Nới rộng tầm hoạt động của các lớp cơ bản FCL và các lớp Data & XML, là các lớp xoay quanh việc xây dựng những ứng dụng dùng đến những công nghệ khác nhau: `Web Services`, `Web Forms` và `Windows Forms`. `Web Services` bao gồm một số lớp hỗ trợ việc phát triển những cấu kiện phân tán "nhẹ cân" có thể hoạt động cho dù đứng trước bức tường lữa và phần mềm Network Address Translator (NAT). Vì `Web Services` sử dụng nghi thức truyền thông chuẩn HTTP và SOAP (Simple Object Access Protocol), những cấu kiện này hỗ trợ plug-and-play xuyên qua cyberspace.

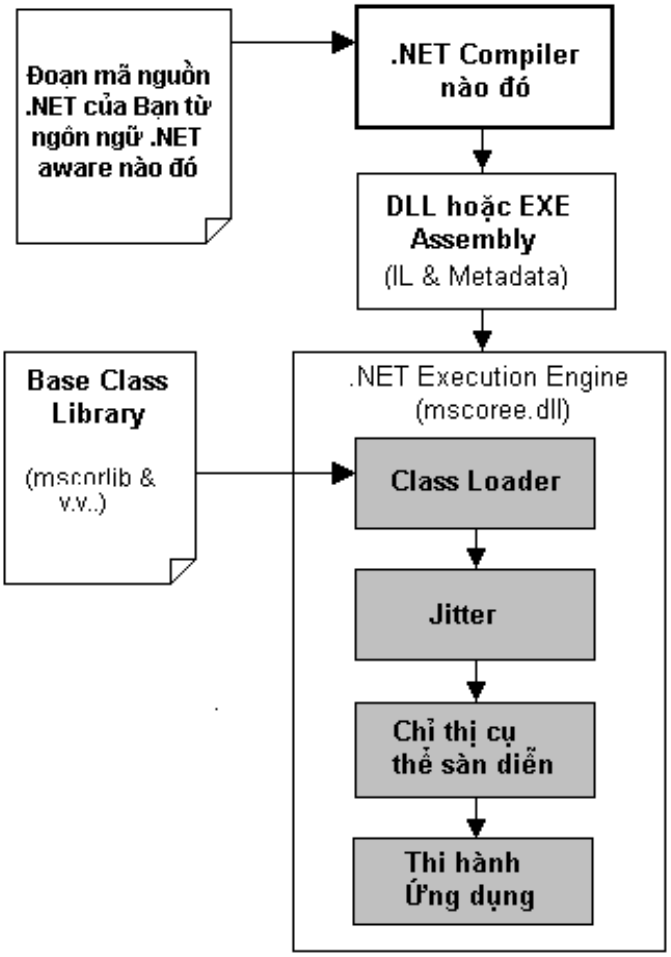
`Web Forms` và `Windows Forms` cho phép bạn áp dụng kỹ thuật RAD (Rapid Application Development) để xây dựng nhanh những ứng dụng web và windows. Đơn giản bạn chỉ cần lôi thả các ô control lên biểu mẫu, cho double-click lên ô control để cho hiện lên cửa sổ mã nguồn rồi bạn khỏ vào đoạn mã ưng ý liên quan đến tình huống gắn liền với ô control.

[20] Lưu tồn là lưu lại nơi nào đó và tồn tại lâu dài, dai dẳng.
[21] Chúng tôi không dịch là "cơ sở dữ liệu', vì từ này dịch không chính xác ý nghĩa của database

1.6.1 Common Language Runtime (CLR)

Từ ngữ runtime (vào lúc chạy) ám chỉ một tập hợp những dịch vụ cần thiết để cho thi hành một đoạn mã nào đó. Mỗi ngôn ngữ đều có riêng cho mình một runtime library hoặc runtime module. Thí dụ, MFC (Microsoft Foundation Class) hoạt động cùng với Visual C++ cần kết nối với MFC runtime library (mfc42.dll), Visual Basic 6.0 thì lại gắn chặt với một hoặc hai runtime module (msvbvm60.dll), còn Java thì lại có Java Virtual Machinee (JVM) như là module runtime.

Với sàn diễn .NET, bộ phận runtime được gọi là **Common Language Runtime (CLR)** có hơi khác so với những runtime vừa kể trên: CLR cung cấp một tầng lớp (layer) runtime duy nhất được định nghĩa rõ ràng và được chia sẻ sử dụng bởi ***tất cả*** các ngôn ngữ .NET aware ("ăn ý" với .NET).



Hình 1-7 : mscoree.dll đang hoạt động

CLR gồm hai bộ phận chủ chốt. Phần thứ nhất, là "cỗ máy" thi hành vào lúc chạy (runtime execution engine), mang tên **mscoree.dll** (tắt chữ Microsoft Core Execution Engine). Khi một assembly được gọi vào thi hành, thì **mscoree.dll** được tự động nạp vào, và đến phiên nó nạp assembly cần thiết vào ký ức. **Runtime Execution Engine** chịu trách nhiệm thực hiện một số công tác. Trước tiên, nó phải lo việc giải quyết "vi trí đóng quân" của assembly và tìm ra kiểu dữ liệu yêu cầu (nghĩa là lớp, giao diện, cấu trúc, v.v..) thông qua metadata trong assembly. Execution Engine biên dịch IL được gắn liền dựa theo chỉ thị cụ thể sàn diễn, tiến hành một số kiểm tra an toàn cũng như một số công việc liên hệ.

Phần thứ hai của CLR là

thư viện lớp cơ bản, mang tên **mscorlib.dll** (tắt chữ Microsoft Core Library), chứa vô số công tác lập trình thông dụng. Khi bạn xây dựng .NET Solutions (giải pháp .NET) bạn sẽ cần đến nhiều phần thư viện này.

Hình 1-7 minh họa cho thấy qui trình làm việc của đoạn mã của bạn (sử dụng đến các kiểu dữ liệu của thư viện lớp cơ bản), trình biên dịch .NET và .NET Execution Engine.

1.6.2 Common Type System (CTS)

Common Type System, (còn gọi là **Common Type Specification**, đặc tả kiểu dữ liệu thông dụng), có nghĩa là ngôn ngữ trung gian IL (Intermediate Language) xuất hiện với một lô kiểu dữ liệu bẩm sinh (predefined, hoặc built-in) được định nghĩa rõ ràng. Trong thực tế, những kiểu dữ liệu này được tổ chức theo một đẳng cấp kiểu (type hierarchy), điển hình trong một môi trường thiên đối tượng.

1.6.2.1 Ý nghĩa của CTS đối với sự hợp tác liên ngôn ngữ (language interoperability)

Lý do CTS rất quan trọng là, nếu lớp được dẫn xuất từ một lớp nào đó hoặc chứa những thể hiện (instance) của những lớp khác, nó cần biết kiểu dữ liệu mà các lớp khác sử dụng đến. Trong quá khứ, chính việc thiếu một cơ chế khai báo loại thông tin này là một cản trở cho việc kế thừa xuyên ngôn ngữ. Loại thông tin này đơn giản không có trong tập tin .EXE chuẩn hoặc DLL.

Một phần nào đó, vấn đề lấy thông tin về kiểu dữ liệu đã được giải quyết thông qua metadata trong assembly. Thí dụ, giả sử bạn đang viết một lớp theo C#, và bạn muốn lớp này được dẫn xuất từ một lớp được viết theo VB .NET. Muốn làm được điều này, bạn sẽ yêu cầu trình biên dịch qui chiếu về assembly mà lớp VB .NET đã được khai báo. Lúc này, trình biên dịch sẽ dùng metadata trên assembly này để lần ra tất cả các hàm hành sự, thuộc tính và vùng mục tin (field), v.v.. liên quan đến lớp VB.NET. Rõ ràng là trình biên dịch cần đến thông tin này để có thể biên dịch đoạn mã của bạn.

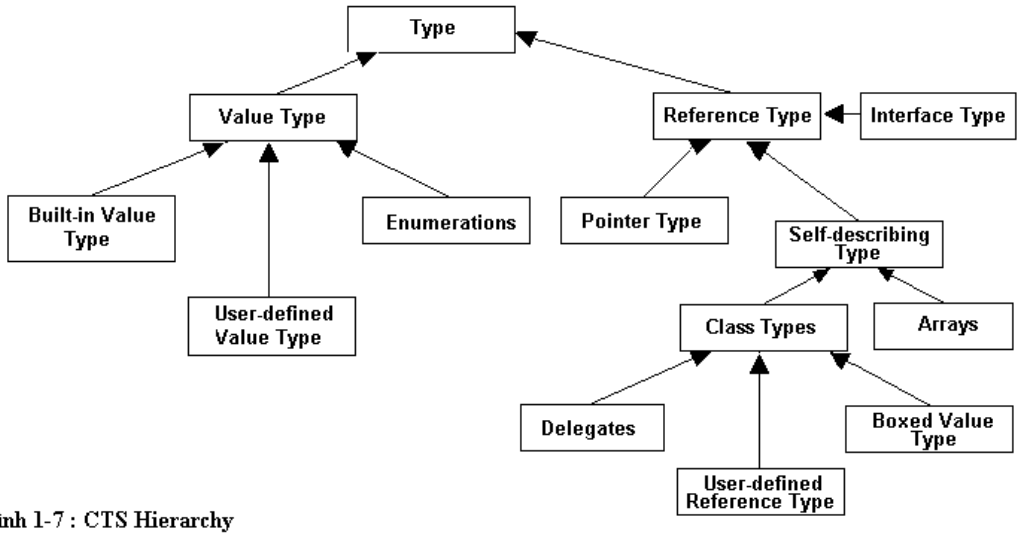
Tuy nhiên, trình biên dịch lại cần đến nhiều thông tin hơn những gì metadata cung cấp. Thí dụ, giả sử một trong những hàm hành sự trên lớp VB .NET được định nghĩa trả về một Integer – là một trong những kiểu dữ liệu bẩm sinh trên VB .NET. Tuy nhiên, C# lại không có kiểu dữ liệu mang tên Integer này. Rõ ràng là chúng ta chỉ có thể dẫn xuất từ lớp và sử dụng hàm hành sự này và sử dụng kiểu dữ liệu trả về từ đoạn mã C# nếu trình biên dịch biết làm thế nào ánh xạ (map) kiểu dữ liệu Integer của VB .NET lên một kiểu dữ liệu nào đó được định nghĩa trên C#.

Điều này có thể thực hiện được vì CTS đã định nghĩa những kiểu dữ liệu bẩm sinh có sẵn trên Intermediate Language (IL), như vậy tất cả các ngôn ngữ tuân thủ .NET Framework sẽ kết sinh ra đoạn mã dựa cuối cùng vào những kiểu dữ liệu này. Thí dụ, ta thử xét lại **`Integer`** của VB .NET, hiện là một số nguyên có dấu 32–bit, được ánh xạ lên kiểu dữ liệu **`Int32`** của IL. Như vậy đây sẽ là kiểu dữ liệu được khai báo trong đoạn mã IL. Vì trình biên dịch C# đã biết qua kiểu dữ liệu này, do đó sẽ không có vấn đề gì. Ở cấp mã nguồn, C# dùng từ chốt **`int`** để ám chỉ **`Int32`**, do đó trình biên dịch sẽ cư xử với hàm hành sự VB .NET xem như hàm sẽ trả về một **`int`**.

Nhìn chung, CTS là một đặc tả hình thức có nhiệm vụ mô tả một kiểu dữ liệu nào đó (lớp, cấu trúc, giao diện, kiểu dữ liệu bẩm sinh, v.v..) phải được định nghĩa thế nào để CLR có thể chấp nhận làm việc. Ngoài ra, CTS cũng định nghĩa một số cấu trúc cú pháp (chẳng hạn nạp chồng các tác tử - overloading operator) mà một số ngôn ngữ "ăn ý với .NET" (.NET aware language) có thể chấp nhận hỗ trợ hoặc không. Khi bạn muốn xây dựng những đoạn mã có thể đem sử dụng bởi tất cả các ngôn ngữ .NET aware, bạn cần tuân thủ các qui tắc của CLS (ta sẽ xem sau trong chốc lát) khi trưng ra những kiểu dữ liệu.

1.6.2.2 Cây đẳng cấp CTS (CTS Hierarchy)

CTS định nghĩa một đẳng cấp rất phong phú về kiểu dữ liệu, phản chiếu một phương pháp luận thiên đối tượng về kế thừa đơn (single heritance) của IL. Hình 1-8 cho thấy kiến trúc đẳng cấp như sau:



Hình 1-7 : CTS Hierarchy

Ta thử xem chi tiết các kiểu dữ liệu được liệt kê ở trên của CTS.

1.6.2.2.1 CTS Class Types

Mọi ngôn ngữ .NET aware đều hỗ trợ khái niệm về một "class type" vì đây là "hòn đá tảng" của lập trình thiên đối tượng. Một lớp thường được cấu thành bởi một số vùng mục tin (field), thuộc tính (property), hàm hành sự (method) và tình huống (event). Như bạn có thể chờ đợi, CTS cho phép một lớp nào đó hỗ trợ những thành viên trừu tượng để cung cấp một giao diện đa hình (polymorphic interface) cho bất cứ lớp nào được dẫn xuất. Các lớp CTS-compliant ("ăn ý" với CTS) chỉ có thể dẫn xuất từ một lớp cơ bản đơn chiếc (single base class), vì C# không cho phép kế thừa từ nhiều lớp (multiple inheritance). Bảng 1-2 liệt kê những đặc tính khá lý thú của Class Types.

Các kiểu dữ liệu trên cây đẳng cấp sẽ được giải thích theo Bảng 1-1 dưới đây:

Bảng 1-1: Bảng liệt kê các kiểu dữ liệu

Kiểu dữ liệu	**Ý nghĩa**
Arrays	*Bản dãy*. Bất cứ kiểu dữ liệu nào chứa một bản dãy đối tượng.
Boxed Value Types	*Dữ liệu kiểu trị bị đóng hộp*. Một dữ liệu kiểu trị tạm thời được bao trọn trong một qui chiếu để có thể được trữ trên ký ức heap.
Built-in Value Types	*Kiểu dữ liệu bẩm sinh kiểu trị* bao gồm phần lớn những kiểu dữ liệu chuẩn "nguyên sinh" (primitive) tượng trưng cho số, trị Bool hoặc ký tự.
Class Types	*Kiểu dữ liệu lớp*, tự mô tả nhưng không phải là bản dãy.
Delegates	*Kiểu dữ liệu ủy thác.* Một loại cấu trúc dữ liệu được thiết kế để cầm giữ những qui chiếu chỉ về các hàm hành sự.
Enumerations	*Cấu trúc liệt kê*. Loạt trị được liệt kê theo đấy mỗi trị được tượng trưng bởi một cái nhãn (label) nhưng lại được trữ dưới dạng số. Thí dụ, nếu bạn muốn tượng trưng màu sắc theo kiểu này, Enumeration cho phép bạn viết ra {Red, Green,Yellow} thay vì {0,1,2}
Interface Types	*Giao diện*. Kiểu dữ liệu lớp dùng làm giao diện.
Pointer Types	*Các con trỏ*.
Reference Types	*Dữ liệu kiểu qui chiếu*. Bất cứ kiểu dữ liệu nào có thể được truy xuất thông qua một qui chiếu và được trữ trên ký ức heap.
Self-describing Types	*Kiểu dữ liệu tự mô tả mình*. Đây là kiểu dữ liệu cung cấp những thông tin liên quan đến dữ liệu dành cho dịch vụ thu gom những vùng ký ức bị chiếm dụng

	nhưng không còn dùng nữa (Garbage Collector).
Type	*Lớp cơ bản* (base class) tượng trưng cho bất cứ kiểu dữ liệu nào.
User-defined Value Types	*Dữ liệu kiểu trị tự tạo.* Kiểu dữ liệu được định nghĩa trong mã nguồn, và được trữ như là dữ liệu kiểu trị. Theo từ ngữ C#, có nghĩa là bất cứ cấu trúc (struct) nào.
User-defined Reference Types	*Dữ liệu kiểu qui chiếu tự tạo*. Kiểu dữ liệu được định nghĩa trong mã nguồn, và được trữ như là dữ liệu kiểu qui chiếu. Theo từ ngữ C#, có nghĩa là bất cứ lớp nào.
Value Types	*Dữ liệu kiểu trị.* Lớp cơ bản tượng trưng cho bất cứ dữ liệu kiểu trị nào.

Bảng 1-2: Đặc tính của NET Class Type

Đặc tính Class Type	**Ý nghĩa**
Lớp có bị "vô sinh" (sealed) hay không ?	Các lớp "vô sinh" (sealed) là kiểu dữ liệu không thể hoạt động như là lớp cơ bản đối với các lớp khác, nghĩa là không thể dẫn xuất từ lớp "vô sinh".
Lớp có thiết đặt bất cứ giao diện nào hay không ?	Một giao diện là một collection gồm những thành viên trừu tượng cung cấp một "khế ước" giữa đối tượng và người sử dụng đối tượng. CTS cho phép một lớp thiết đặt bất cứ bao nhiêu giao diện cũng được.
Lớp thuộc loại trừu tượng (abstract) hoặc loại cụ thể (concrete)?	Các lớp trừu tượng không thể trực tiếp tạo ra đối tượng, nhưng lại được dùng làm khuôn mẫu định nghĩa những hành xử thông dụng đối với những lớp được dẫn xuất. Các lớp cụ thể có thể được dùng tạo ra đối tượng.
Độ tầm nhìn (visibility) của lớp thế nào?	Mỗi lớp sẽ được cấu hình với một thuộc tính tầm nhìn (visibility attribute). Cơ bản mà nói, thuộc tính này cho biết liệu xem lớp có thể được dùng bởi những assembly nằm ngoài hoặc chỉ trong lòng assembly (nghĩa là một lớp hỗ trợ private).

1.6.2.2.2 CTS User-defined Value Types hoặc Structure Types

Kiểu dữ liệu kiểu trị do người sử dụng định nghĩa, còn gọi là kiểu **struct**. Khái niệm về một cấu trúc cũng được định nghĩa bởi CTS. Nếu bạn đã quen C, bạn sẽ vui khi biết kiểu dữ liệu này vẫn còn chỗ đứng trong thế giới .NET. Nhìn chung, một **struct** là một kiểu dữ liệu "nhẹ cân" với khá nhiều biệt lệ (xem Chương 8, "Struct"). CTS-compliant **struct** có thể có bất cứ bao nhiêu hàm constructor có thông số (parametized constructor). Theo cách này, bạn có thể thiết lập trị của mỗi vùng mục tin trong thời gian xây dựng lớp. Thí dụ:

```
// Tạo một C# struct (mô tả một bé sơ sinh)
struct Baby
{   // struct có thể có những vùng mục tin (field)
    public string name;

    // struct có thể có hàm constructor có thông số
    public Baby(string name)
    { this.name = name; }

    // struct có thể có hàm hành sự
    public void Cry()
    {   Console.WriteLine("OaOaOaOaOa!!!");
    }
    public Bool IsSleeping() { return false;}   // hàm hỏi ngủ không
    public Bool IsChanged() { return false; }   // hàm hỏi thay tả
không
}
```

Sau đây struct Baby đi vào hoạt động:

```
Baby barnaBaby = new Baby("Titi");   // đẻ một con cho mang tên Titi
Console.WriteLine("Thay tã chưa?: {0}",
                    barnaBaby.IsChanged.ToString());
Console.WriteLine("Ngủ chưa?: {0}",
                    barnaBaby.IsSleeping.ToString());

// Thử xem bé khóc thét thế nào?
for (int = 0; i < 1000; i++)
    barnaBaby.Cry();
```

Tất cả các kiểu dữ liệu struct "chiều lòng CTS" (CTS-compliant) đều được dẫn xuất từ một lớp cơ bản **System.ValueType**. Lớp cơ bản này cấu hình một struct hoạt động như là một kiểu dữ liệu kiểu trị (nghĩa là trữ dữ liệu trên ký ức stack) thay ví kiểu qui chiếu (trữ dữ liệu trên ký ức heap). Ban cũng nên nhớ cho, CTS cho phép **struct** thiết đặt bao nhiêu giao diện cũng được. Tuy nhiên, **struct** được xem như là một lớp "vô sinh", nghĩa là không "đẻ" ra những lớp dẫn xuất khác.

1.6.2.2.3 CTS Interface Types

Giao diện chẳng qua cũng chỉ là một tập hợp (collection) những định nghĩa hàm hành sự trừu tượng, thuộc tính và tình huống. Khác với COM cổ điển, giao diện .NET không dẫn xuất từ một giao diện cơ bản, chẳng hạn **IUnknown**. Tự bản thân, giao diện không hữu dụng chi mấy. Tuy nhiên, khi một lớp hoặc struct thiết đặt một giao diện nào đó theo cách riêng của mình, thì bạn có thể đòi hỏi truy xuất chức năng được cung cấp bằng cách sử dụng một qui chiếu giao diện (interface reference). Khi bạn xây dựng một giao diện tự tạo (custom interface) sử dụng một ngôn ngữ .NET aware, thì CTS cho phép một giao

diện nào đó được dẫn xuất từ nhiều giao diện cơ bản (base interface). Chúng tôi sẽ giải thích rõ giao diện ở Chương 9, "Giao diện".

1.6.2.2.4 CTS Enumeration Types

Enumeration, kiểu dữ liệu liệt kê, là một kiến trúc lập trình rất tiện lợi cho phép bạn gộp cặp tên/trị (name/value) dưới một cái tên rất đặc biệt. Thí dụ, giả sử bạn đang tạo một ứng dụng video game cho phép người sử dụng chọn một trong 3 kiểu chơi (Wizard, Fighter, hoặc Thief). Thay vì theo dõi một cách thô thiển trị số để tượng trưng cho mỗi khả năng**,** bạn có thể xây dựng một custom enumeration như sau:

```
// Một C# enumeration
enum PlayerType
{   Wizard = 100,
    Fighter = 200,
    Thief = 300
}
```

CTS yêu cầu kiểu dữ liệu liệt kê dẫn xuất từ một lớp cơ bản thông dụng mang tên **System.Enum**. Như bạn có thể thấy lớp cơ bản này định nghĩa một số thành viên cho phép bạn trích (và thao tác) những cặp tên/trị nằm đằng sau.

Chúng tôi sẽ đề cập đến Enumerations ở chương 4, "Thử xem căn bản ngôn ngữ C#", mục 4.2.3.

1.6.2.2.4 CTS Delegate Type

Trên C#, delegate (ủy thác), tương đương với con trỏ hàm (function pointer) kiểu C, nhưng an toàn hơn về mặt kiểu dữ liệu. Khác biệt chủ yếu là việc .NET delegate tượng trưng cho một lớp được dẫn xuất từ **System.MulticastDelegate**, thay vì từ một vị chỉ ký ức (memory address) thô thiển. Kiểu dữ liệu delegate, hoạt động giống như một hộp thư giao liên, chỉ hữu ích khi bạn muốn cung cấp một thể thức cho một đối tượng chuyển một triệu gọi hàm cho một đối tượng khác. Trong Chương 13, "Ủy thác và Tình huống (Delegate & Events)", bạn sẽ làm quen với delegate trong việc xây dựng nghi thức về tình huống (events).

1.6.2.2.5 CTS Self-describing Types

Như bạn có thể thấy, lớp và struct có thể có bất cứ bao nhiêu thành viên (member) cũng được. Về mặt hình thức, một thành viên là một hàm hành sự, hoặc thuộc tính, hoặc vùng mục tin, hoặc tình huống. Đối với mỗi thành viên, CTS định nghĩa một số "đồ trang sức" cho biết thêm thông tin về thành viên. Thí dụ, mỗi thành viên sẽ được gắn liền một mức độ nhìn thấy (visibility), nghĩa là private, public, protected, v.v.. Một thành viên

cũng có thể được khai báo như là "abstract" để kiểm tra mức độ đa hình trên các kiểu dữ liệu được dẫn xuất. Các thành viên có thể là "static" (bị gắn chặt ở cấp lớp) hoặc "instance" (bị gắn chặt ở cấp đối tượng).

Dịch vụ Garbage Collector (GC) lo thu hồi lại những vùng ký ức bị chiếm dụng hết sử dụng sẽ dựa trên những thông tin cung cấp bởi kiểu dữ liệu này để làm tròn nhiệm vụ.

1.6.2.2.6 CTS Boxed Value Types

Dữ liệu kiểu trị bị "đóng hộp". Về sau, khi bạn bắt đầu học C#, bạn sẽ thấy trong nhiều trường hợp, có những dữ liệu kiểu trị (value type) tạm thời cần được chuyển đổi thành dữ liệu kiểu qui chiếu và được trữ trên ký ức heap. Tiến trình chuyển đổi này được gọi là "boxing" [22], cho đóng hộp. Boxing đòi hỏi mỗi dữ liệu kiểu trị phải có một kiểu qui chiếu tương ứng tượng trưng cho một phiên bản bị đóng hộp trên heap của kiểu dữ liệu này.Thông thương, bất cứ lúc nào một dữ liệu kiểu trị được định nghĩa trong đoạn mã C# của bạn, thì lặng lẽ .NET ở hậu trường cũng sẽ định nghĩa một dữ liệu kiểu đóng hộp tương ứng, xem như là dữ liệu kiểu qui chiếu tượng trưng cho những biến kiểu dữ liệu này khi chúng muốn được đối xử như là kiểu qui chiếu.

Trong chương 6, "Kế thừa và Đa hình", mục 6.6 chúng tôi sẽ đề cập đến kiểu dữ liệu này.

1.6.2.2.7 CTS Intrinsic Value Types

CTS đã định nghĩa một bộ kiểu dữ liệu "bẩm sinh" (intrinsic hoặc built-in) kiểu trị như Boolean, int, float, char, v.v.. Bảng 1-3 sau đây liệt kê những kiểu dữ liệu bẩm sinh mà .NET cung cấp cho bạn:

Bảng 1-3: Kiểu dữ liệu bẩm sinh kiểu trị định nghĩa bởi CTS

Kiểu dữ liệu .NET	Biểu diễn theo Visual Basic .NET	Biểu diễn theo C#	Biểu diễn theo C++ với nới rộng được managed
System.Byte	Byte	byte	char
System.SByte	(không hỗ trợ)	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int hoặc long
System.Int64	Long	long	_int64
System.UInt16	(không hỗ trợ)	ushort	unsigned short
System.UInt32	(không hỗ trợ)	uint	unsigned int hoặc unsigned long

[22] Không phải đấu quyền Anh đâu.

System.UInt64	(không hỗ trợ)	ulong	unsigned _int64
System.Single	Single	float	float
System.Double	Double	double	double
System.Object	Object	object	Object*
System.Char	Char	char	_wchart_t
System.String	String	string	String*
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	Bool	Bool

Như bạn có thể thấy, không phải tất cả các ngôn ngữ đều có khả năng tượng trưng cùng các thành viên dữ liệu của CTS. Như bạn có thể tưởng tượng, có thể là có ích khi cho tạo một subset của CTS định nghĩa một bộ kiến trúc lập trình (và kiểu dữ liệu) thông dụng được chia sẻ sử dụng bởi tất cả các ngôn ngữ .NET aware. Chính lúc này, Commom Language Specification (CLS) nhập cuộc.

1.6.3 Common Language Specification (CLS)

Common Language Specification (CLS, đặc tả ngôn ngữ thông dụng) làm việc "ăn ý" với CTS để bảo đảm suôn sẻ việc hợp tác liên ngôn ngữ. CLS đề ra một số chuẩn mực mà tất cả các trình biên dịch nhắm vào .NET Framework phải chấp nhận hỗ trợ. Có thể có một số người viết trình biên dịch muốn hạn chế khả năng của một trình biên dịch nào đó bằng cách chỉ hỗ trợ một phần nhỏ những tiện nghi mà IL và CTS cung cấp. Việc này không hề gì miễn là trình biên dịch hỗ trợ mọi thứ đã được định nghĩa trong CLS.

Để lấy một thí dụ, CTS định nghĩa một kiểu dữ liệu số nguyên có dấu 32-bit, **`Int32`** và không dấu, **`UInt32`**. C# nhận biết hai kiểu dữ liệu này là **int** và **uint**. Nhưng VB .NET thì chỉ công nhận độc nhất **`Int32`**, mang từ chốt **Integer** mà thôi.

Ngoài ra, CLS hoạt động theo hai cách. Trước tiên, có nghĩa là những trình biên dịch riêng rẽ không nhất thiết phải mạnh, chịu hỗ trợ tất cả các chức năng của .NET Framework, đây là một cách khuyến khích việc phát triển những trình biên dịch đối với những ngôn ngữ khác sử dụng sàn diễn .NET. Thứ đến, CLS đua ra một bảo đảm là nếu bạn chỉ giới hạn các lớp của bạn vào việc trưng ra những chức năng chiều ý CLS (CLS-compliant), thì CLS bảo đảm là đoạn mã viết theo các ngôn ngữ khác có thể sử dụng các lớp của bạn. Thí dụ, nếu bạn muốn đoạn mã của bạn là CLS-compliant, bạn sẽ không có bất cứ hàm hành sự nào trả về **`UInt32`**, vì kiểu dữ liệu này không phải là thành phần của CLS. Nếu bạn muốn, bạn cũng có thể trả về một UInt32, những lúc ấy không bảo đảm đoạn mã của bạn hoạt động xuyên ngôn ngữ. *Nói cách khác, hoàn toàn chấp nhận khi bạn viết một đoạn mã không CLS-compliant. Nhưng nếu bạn làm thế, thì không chắc gì đoạn mã biên dịch IL của bạn hoàn toàn độc lập về mặt ngôn ngữ (language independent*).

CLS là một tập hợp những qui tắc hướng dẫn mô tả một cách chi tiết sống động những tính năng (feature) tối thiểu và trọn vẹn mà một trình biên dịch .NET aware nào đó phải chấp nhận hỗ trợ để có thể tạo ra đoạn mã chiều ý CLR, và đồng thời có thể được sử dụng theo một cách đồng nhất giữa các ngôn ngữ theo đuôi sàn diễn .NET. Trong chừng mực nào đó, CLS có thể được xem như là một tập hợp con (*subset*) của chức năng trọn vẹn được định nghĩa bởi CTS.

CLS bao gồm một tập hợp những qui tắc mà các nhà tạo công cụ phải tuân thủ nếu muốn sản phẩm mình làm ra có thể hoạt động không trục trặc trong thế giới .NET. Mỗi qui tắc sẽ mang một tên (chẳng hạn "CLS Rule 6"), và mô tả qui tắc này sẽ ảnh hưởng thế nào đối với người thiết kế công cụ cũng như đối với người tương tác với công cụ.

Ngoài ra, CLS còn định nghĩa vô số qui tắc khác; chẳng hạn CLS mô tả một ngôn ngữ nào đó phải biểu diễn thế nào các chuỗi chữ, liệt kê kiểu enumeration phải được biểu diễn thế nào về mặt nội tại, v.v...Nếu bạn muốn biết chi tiết về CLS, thì mời tham khảo MSDN trên máy của bạn.

1.6.4 .NET Base Class Library

Thư viện .NET Base Class Library là một tập hợp khổng lồ các đoạn mã lớp được giám quản (managed code) mà Microsoft đã bỏ công viết ra, cho phép bạn làm bất cứ công việc gì mà trước kia chỉ có ở Windows API. Các lớp trên thư viện này theo cùng mô hình đối tượng mà IL dùng đến, dựa trên single inheritance. Đây có nghĩa là bạn có thể hoặc hiển lộ (instantiate) đối tượng từ bất cứ lớp cơ bản .NET thích hợp nào đó hoặc bạn có thể dẫn xuất lớp riêng của bạn từ các lớp cơ bản này.

Điểm lý thú của các lớp cơ bản .NET là chúng được thiết kế cho dễ sử dụng và hoàn toàn tự sưu liệu. Thí dụ, muốn bắt đầu một mạch trình (thread) bạn triệu gọi hàm **Start()** của lớp **Thread. M**uốn mở một tập tin bạn triệu gọi hàm **Open()** của lớp **File.** Muốn vô hiệu hoá một **TextBox**, bạn cho đặt để thuộc tính **Enabled** của đối tượng **TextBox** về false. Ban thấy là "dễ ợt" phải không ?

Đối với những lập trình viên C++ họ sẽ nhẹ nhõm khi biết sẽ không còn vật lộn với những hàm **GetDIBits(), RegisterWndClassEx()** và **IsEqualIID()**và vô số hàm khác đòi hỏi các mục quản (handle) Windows phải chạy rong. Mặt khác, lập trình viên C++ bao giờ cũng có thể truy xuất dễ dàng toàn bộ Windows API.

Các lĩnh vực mà .NET Class Library bao gồm là:

- Các chức năng cốt lõi cung cấp bởi IL, chẳng hạn những kiểu dữ liệu bẩm sinh trong Common Type System (CTS).

- Hỗ trợ giao diện GUI (Graphical User Interface), các ô control v.v..
- Web Forms (ASP.NET)
- Data Access (ADO.NET) ; truy xuất dữ liệu trên các căn cứ dữ liệu
- Directory Access: truy xuất thư mục.
- File System và Registry Access: hệ thống xuất nhập dữ liệu trên các tập tin và truy xuất Registry.
- Networking & Web browsing: mạng và trình duyệt trên Web.
- .NET Attributes và Reflection.
- Thâm nhập vào các khía cạnh khác nhau của hệ điều hành Windows, môi trường các biến, v.v..
- COM interoperability
- Graphics GDI+

Phần lớn các lớp trên .NET Class Library hiện được viết bằng C#. Đó là theo lời của Microsoft. Tin hay không là tùy bạn.

1.7 Ngôn ngữ C#

.NET hỗ trợ chính thức 4 ngôn ngữ: C#, VB.NET, J# và C++ managed. Trong phần này chúng tôi chỉ đề cập đến C#, là mục tiêu của bộ sách 6 tập này.

Với cuộc "cách mạng" .NET này, C# là một ngôn ngữ rất đỗi đơn giản, trong sáng, và chỉ gồm vào khoảng 80 từ chốt (keyword), và một tá kiểu dữ liệu bẩm sinh, nhưng C# diễn đạt rất tốt khi đi vào việc thiết đặt thi công những khái niệm lập trình tiên tiến. C# bao gồm những hỗ trợ cho việc lập trình thiên đối tượng, có cấu trúc cũng như thiên về "lắp ráp" các ứng dụng dựa trên cấu kiện (component-based).

C# là một ngôn ngữ lập trình thiên đối tượng nên nó hỗ trợ việc định nghĩa cũng như làm việc với các lớp. Lớp định nghĩa những kiểu dữ liệu mới cho phép bạn nới rộng ngôn ngữ để mô hình hoá vấn đề bạn đang bận tâm giải quyết. C# có chứa những từ chốt cho phép khai báo những lớp mới, thuộc tính và hàm hành sự cũng như để thiết đặt khái niệm

gói ghém (encapsulation), kế thừa và đa hình (polymorphisme), cái kiềng ba chân của lập trình thiên đối tượng..

Trên C#, mọi việc liên quan đến khai báo một lớp nằm ngay bản thân trong phần khai báo lớp. Khai báo lớp trên C# không đòi hỏi những tập tin tiêu đề (header file) hoặc ngôn ngữ IDL (Interface Definition Language) như với C++. Ngoài ra, C# hỗ trợ một kiểu XML mới cho phép sưu liệu ngay tại chỗ (inline documentation) đơn giản hóa việc sưu liệu đối với ứng dụng.

C# cũng hỗ trợ giao diện, một kiểu khế ước với một lớp liên quan đến những dịch vụ mà giao diện đề ra. Trên C#, một lớp chỉ có thể kế thừa từ một lớp khác nhưng có thể thiết đặt vô số giao diện. Khi thiết đặt một giao diện, một lớp C# hứa là sẽ cung cấp những chức năng như giao diện đã đề xuất.

C# cũng cung cấp hỗ trợ đối với struct, một khái niệm mà ý nghĩa đã thay đổi khá nhiều từ C++. Trên C#, struct là một lớp "nhẹ cân", hạn chế, và khi được hiển lộ thì đòi hỏi rất ít "công chăm sóc" đối với hệ điều hành cũng như ít chiếm dụng ký ức so với một lớp cỗ điển. Một struct không thể được kế thừa từ một lớp khác hoặc cho kế thừa, nhưng một struct có thể thiết đặt thi công giao diện.

C# chịu hỗ trợ những chức năng thiên về cấu kiện chẳng hạn thuộc tính, tình huống và cấu trúc khai báo (declarative construct) được gọi là attributes. Lập trình thiên cấu kiện được hỗ trợ bởi CLR trong việc trữ metadata kèm theo đoạn mã đối với một lớp. Metadata mô tả lớp bao gồm thuộc tính, hàm hành sự cũng như thông tin về an toàn và các attributes khác, chẳng hạn liệu xem lớp có được "sản sinh hàng loạt" (serialized) hay không. Khái niệm về serialization sẽ được giải thích ở tập 2, chương 1, "Input/Output và Object serialization". Đoạn mã chứa phần lô gic cần thiết để thực hiện các chức năng của lớp. Do đó, một lớp được biên dịch là một đơn vị "tự cung tự cấp"(self contained), nên một môi trường "chứa chấp" (host environment) biết cách đọc thế nào metadata và đoạn mã sẽ không cần thêm thông tin để có thể sử dụng lớp. Bằng cách sử dụng C# và CLR bạn có thể thêm custom metadata lên một lớp bằng cách tạo những custom attributes. Cũng tương tự như thế, ta có thể đọc metadata của lớp bằng cách sử dụng lớp CLR chịu hỗ trợ reflection.

Một assembly là một tập hợp nhiều tập tin mà lập trình viên có thể xem như là một DLL hoặc EXE động. Trên .NET, một assembly là một đơn vị cơ bản dùng lại được, giải quyết vấn đề phiên bản, an toàn và triển khai sử dụng dễ dàng. CLR cung cấp một số lớp cho phép thao tác trên assembly.

Chương 2

Bắt đầu từ đây ta tiến lên!

Theo truyền thống, các sách lập trình ở Mỹ bao giờ cũng bắt đầu bởi một chương trình con "Hello World!", với ý thức muốn toàn cầu hóa ngành tin học của Mỹ. Thế thì ta cũng làm như thế, nhưng thay vì "Hello, World!" ta bắt đầu bởi "Xin chào bà con!". Trong chương này, bạn sẽ học tạo một chương trình viết theo C#, rồi cho biên dịch và chạy thử nghiệm. Việc phân tích chương trình con này sẽ dẫn dắt bạn vào những chức năng chủ chốt của ngôn ngữ C#.

2.1 Chương trình C#, hình thù ra sao?

Thí dụ 2-1 minh họa một chương trình C# ở dạng đơn giản nhất; đây là một ứng dụng đơn giản chạy trên console[23] hiển thị một thông điệp lên màn hình:

Thí dụ 2-1: Một chương trình C# đơn giản "XinChao"'

```
class XinChao
{
   public static void Main()
   {
      // Dùng đến đối tượng console của System
      System.Console.WriteLine("Xin Chào Bà Con!");
   }
}
```

Bạn có thể biên dịch chương trình này bằng cách khỏ vào một trình soạn thảo văn bản đơn giản, Notepad chẳng hạn, rồi cho cất trữ dưới dạng một tập tin (thí dụ XinChao.cs) mang phần đuôi là **.cs** (tắt chữ C Sharp), rồi cho chạy trình biên dịch C# command line (csc.exe) trên tập tin XinChao.cs:

```
csc XinChao.cs
```

csc là tắt chữ C Sharp Compiler.

[23] Console là một từ ngữ tin học rất xưa để chỉ "màn hình + bàn phím". Ở đây các ứng dụng console không chạy trên Windows.

Một chương trình khả thi (executable) mang tên **XinChao.exe** sẽ được tạo ra, và bạn có thể cho chạy chương trình này từ command line giống như với DOS hoặc từ Windows Explorer như bất cứ chương trình khả thi nào. Bây giờ ta thử xem qua chương trình đơn giản trên.

Nhưng trước tiên, bạn nên biết cho là trên C#, cũng như trên các ngôn ngữ C khác, chương trình được cấu thành bởi những *câu lệnh* (statement[24]) giống như bài văn được hình thành bởi những câu văn. Nếu mỗi câu văn được kết thúc bởi một dấu chấm (.), thì câu lệnh C# lại được kết thúc bởi một dấu chấm phẩy (;). Và nhiều câu lệnh có thể gộp thành một khối (block) được bao ở hai đầu bởi cặp dấu ngoặc nghéo {} (curly brace). Câu lệnh nếu dài có thể tiếp tục xuống hàng dưới, không cần đến một ký tự (gọi là ký tự tiếp tục, continuation character, như dấu gạch dưới – underscore – trên Visual Basic) báo cho biết câu lệnh tiếp tục hàng dưới.

2.2 Lớp, Đối tượng và Kiểu dữ liệu

Vấn đề cốt lõi trong lập trình thiên đối tượng (object-oriented programming) là tạo ra những kiểu dữ liệu (*type*) mới. Một *kiểu dữ liệu* tượng trưng cho một vật hoặc một sự việc. Đôi lúc vật hoặc sự việc mang tính trừu tượng (abstract), chẳng hạn một bảng dữ liệu (data table) hoặc một mạch trình (thread); thỉnh thoảng nó lại xác thực rõ ràng thấy được, chẳng hạn một nút ấn (button) hoặc một cửa sổ nho nhỏ trên màn hình. Một kiểu dữ liệu thường mô tả những thuộc tính (property) và cách hành xử (behavior) tổng quát của vật hoặc sự việc.

Nếu chương trình của bạn dùng đến 3 thể hiện (instance) của một kiểu dữ liệu nút ấn (button) trên một cửa sổ - chẳng hạn nút ấn **OK, Cancel** và **Help** - thì mỗi thể hiện sẽ chia sẻ sử dụng chung một vài thuộc tính cũng như những cách hành xử. Thí dụ, mỗi thể hiện sẽ có một kích thước, một vị trí trên màn hình và một dòng văn bản (text, gọi là nhãn, như "OK", "Cancel" hoặc "Help"). Tương tự như thế, tất cả 3 nút đều có những hành xử giống nhau như có thể vẽ ra lên màn hình được, có thể bị ấn xuống v.v.. Như vậy, chi tiết có thể khác nhau giữa các nút riêng rẽ, nhưng nhìn chung chúng cùng thuộc một loại, mà ở đây chúng tôi gọi là thuộc cùng một kiểu dữ liệu.

Giống như mọi ngôn ngữ lập trình thiên đối tượng, trên C#, kiểu dữ liệu thường được định nghĩa bởi một ***lớp*** (class), trong khi những thể hiện riêng rẽ của lớp được gọi là ***đối tượng*** (object). Một **lớp** được xem như là định nghĩa của một kiểu dữ liệu tự tạo (user-defined type – UDT) và thường được xem như là một "bản vẽ" (blueprint) đối với những biến kiểu dữ liệu này, còn một *đối tượng* là từ ngữ mô tả một thể hiện nào đó của một lớp

[24] Statement ở đây không nên dịch là "phát biểu" như có một người đã dịch.

đặc biệt vào lúc chạy. Trên C#, từ chốt **new** là cách duy nhất để tạo một thể hiện đối tượng từ một lớp.

Về sau, trong chương này chúng tôi sẽ đề cập đến những kiểu dữ liệu khác ngoài lớp, bao gồm **enum** (liệt kê) , **struct** (cấu trúc) và **delegate** (ủy thác); nhưng tạm thời chúng ta tập trung xét đến lớp.

Chương trình "XinChao" chỉ khai báo một lớp mà thôi, đó là lớp **XinChao.** Muốn khai báo một lớp bạn chỉ cần sử dụng từ chốt **class,** rồi đặt cho lớp một cái tên "cúng cơm" - chẳng hạn **XinChao** - rồi sau đó định nghĩa tiếp những thuộc tính và những cách hành xử. Định nghĩa những thuộc tính và những cách hành xử của một lớp C# phải được bao trong một cặp dấu ngoặc nghéo, **{}**.

2.2.1 Các hàm hành sự (method)

Một lớp thường có những thuộc tính lẫn những cách hành xử. Hành xử được định nghĩa bởi những hàm mà chúng tôi gọi là ***hàm hành sự***[25] (method). Đây là những thành viên (member) của lớp. Các thuộc tính sẽ được đề cập ở Chương 5, "Lớp và Đối tượng".

Một *hàm hành sự* là một hàm (function) mà lớp sở hữu. Đôi khi người ta còn gọi hàm hành sự là *hàm thành viên* (member function). (Người Mỹ khoái đặt tên mới cho những khái niệm cũ rích, mà ta thường nói là "bình mới rượu cũ". Method chẳng qua cũng chỉ là một function, thể hiện một lối hành xử. Nói cách khác, hàm hành sự định nghĩa những gì lớp có thể làm được hoặc cách lớp hành xử. Điển hình, người ta thường gán cho hàm hành sự một cái tên "cúng cơm" mang tính hành động, chẳng hạn `WriteLine()`**,** viết ra một hàng, hoặc `AddNumbers()`, cộng các số lại. Tên cúng cơm này thường được ghép bắt đầu từ một động từ (verb). Thí dụ, hàm `WriteLine()` do động từ Write và ghép với danh từ Line có nghĩa là "viết một hàng". Tuy nhiên, trong trường hợp đặc biệt của chúng ta ở đây, hàm hành sự mang tên `Main()`, không mang một hành động nào cả, nhưng chỉ cho CLR biết là hàm chủ chốt đối với lớp của bạn, một điểm đột nhập vào chương trình.

Khác với C++, **Main()** ở đây viết hoa, và có thể trả về một số nguyên (**int)** hoặc chĩa về hư vô (**void**).

CLR sẽ triệu gọi **Main()** khi chương trình bạn bắt đầu chạy. Xem như **Main()** là điểm đột nhập xông vào (entry point) chương trình của bạn, và mỗi chương trình khả thi C# phải có ít nhất một hàm hành sự **Main()**. Bạn để ý **Main()** phải viết hoa chữ M.

---

[25] Đây không phải là một "phương thức" như có người đã dịch. Đây là một hàm thi hành một việc gì đó, giống như Triển Chiêu đi thi hành công vụ.

Muốn khai báo một hàm hành sự, bạn theo cú pháp sau đây:

```
[modifiers] return-type MethodName ([parameters])
{
   // Phần thân của hàm
}
```

Những gì nằm trong cặp dấu ngoặc vuông **[]** (square bracket) là tùy chọn (optional), nghĩa là không bắt buộc phải có. *Modifier* được dùng chỉ định một vài chức năng của hàm hành sự mà ta đang định nghĩa, chẳng hạn hàm hành sự được triệu gọi từ đâu. Trong trường hợp của ta, ta có hai modifier: **public** và **static**. Từ chốt **public** (công cộng) cho biết hàm hành sự có thể được triệu gọi bất cứ nơi nào, nghĩa là có thể được triệu gọi từ ngoài lớp của chúng ta. Từ chốt **static** sẽ được giải thích sau (xem mục 2.2.8, "Từ chốt Static"), *return-type* cho biết kiểu dữ liệu của trị trả về, còn MethodName theo sau là tên "cúng cơm" của hàm. Ngoài ra, khai báo hàm cũng cần đến cặp dấu ngoặc tròn () cho dù hàm có nhận thông số (*parameter*) hay không. Thí dụ hàm sau đây:

```
int myMethod(int size);
```

khai báo một hàm hành sự mang tên **myMethod** chịu nhận một thông số mang tên size là một số nguyên **int** và sẽ được qui chiếu trong lòng hàm hành sự như là **size**. Hàm trên trả về một trị số nguyên. Chính kiểu dữ liệu trị trả về cho người sử dụng hàm biết là loại dữ liệu nào sẽ được hàm trả về khi nó hoàn thành nhiệm vụ.

Một vài hàm hành sự sẽ không trả về một trị nào cả; trong trường hợp này ta bảo hàm trả về một **void** (hư vô), sử dụng đến từ chốt **void**. Thí dụ:

```
void myVoidMethod();
```

khai báo một hàm hành sự trả về một void và không nhận thông số nào cả. Trên C#, bao giờ bạn cũng phải khai báo một kiểu dữ liệu trả về hoặc **void**.

2.2.2 Các dòng chú giải (comments)

Một chương trình C# cũng có thể chứa những *dòng chú giải* (comment). Bạn xem lại trên chương trình XinChao, hàng đầu tiên sau dấu ngoặc nghéo mở {:

```
// Dùng đến đối tượng console của System
```

Dòng văn bản bắt đầu bởi hai dấu gạch chéo "//", đây có nghĩa là một hàng *chú giải*. Hàng chú giải chỉ là một ghi chú hoặc nhắc nhở đối với lập trình viên, không ảnh hưởng gì đến hoạt động của chương trình. C# hỗ trợ 3 loại chú giải:

Loại thứ nhất được gọi là *C++ style comment* (chú giải kiểu C++), cho biết tất cả các văn bản nằm ở tay phải của cặp dấu (//) được xem như là chú giải, cho tới cuối dòng. Nếu dòng chú giải trải dài qua hàng sau, thì phải lặp lại cặp dấu (//) ở đầu hàng sau.

Loại thứ hai được gọi là *C-style comment* (chú giải kiểu C), bắt đầu là một cặp dấu (/*) và kết thúc là một cặp dấu (*/). Kiểu này cho phép bạn kéo dài lời chú giải lên nhiều hàng, thay vì nhiều hàng chú giải với cặp dấu (//) ở đầu hàng. Thí dụ:

Thí dụ 2-2: Minh họa sử dụng chú giải trên nhiều hàng

```
using System;

class XinChao
{
    static void Main()
    {
    /* Dùng đến đối tượng console của System
    như sẽ được giải thích trong chương 2 */
    Console.WriteLine("Xin Chào Bà Con!");
    }
}
```

Loại chú giải thứ ba mà C# hỗ trợ gồm 3 dấu gạch chéo (///) được dùng trong sưu liệu (documentation) đoạn mã của bạn dựa trên XML, mà chúng tôi sẽ đề cập vào cuối chương này.

2.2.3 Các ứng dụng trên Console

"XinChao" là một thí dụ về một chương trình chạy trên console. Thuật ngữ "console" ám chỉ cặp "bàn phím + màn hình". Một ứng dụng console sẽ không có giao diện đồ họa người sử dụng (graphical user interface – GUI) nghĩa là không có ô liệt kê (listbox), các nút ấn, cửa sổ v.v.. Dữ liệu nhập xuất đều thông qua console chuẩn (điển hình là một command hoặc DOS window trên máy vi tính). Trong tập I bộ sách này, chúng tôi chỉ dùng những ứng dụng console để giúp đơn giản hoá các thí dụ, tập trung vào bản thân ngôn ngữ. Trong tập 2 đến tập 5 của bộ sách này, chúng tôi sẽ chuyển qua các ứng dụng trên Windows và trên Web, và vào lúc ấy chúng tôi sẽ nhấn mạnh đến những công cụ (tool) thiết kế giao diện người sử dụng Visual Studio .NET.

Trên thí dụ 2-1, hàm hành sự **Main()** chỉ làm mỗi một việc là cho in dòng văn bản "Xin Chào Bà Con!" lên màn hình. Màn hình được quản lý bởi một đối tượng mang tên `Console`. Đối tượng `Console` có một hàm hành sự mang tên **`WriteLine()`**, viết ra một hàng, chịu nhận một *string* (chuỗi chữ, một loạt ký tự) rồi cho viết lên đối tượng kết xuất chuẩn (màn hình). Khi bạn cho chạy chương trình này, một lệnh (command) hoặc

màn hình DOS sẽ phụt lên trên màn hình và cho hiển thị dòng chữ "Xin Chào Bà Con!" (bạn không kịp thấy thì dòng chữ biến mất tiêu)..

Thường bạn triệu gọi (invoke) một hàm hành sự thông qua một tác tử dấu chấm (dot operator) – (.). Do đó, muốn triệu gọi hàm hành sự `WriteLine()` của đối tượng `Console`, bạn phải viết: `Console.WriteLine(…);` và điền vào trong cặp dấu ngoặc tròn chuỗi chữ bạn muốn in ra. Có thể bạn đã biết chuỗi chữ phải được đóng khung trong cặp dấu nháy (" ")

2.2.4 Namespaces

Trong Chương 1, "C# và sàn diễn .NET", chúng tôi đã đề cập qua về namespace. Chúng tôi không dịch từ namespace. Đây là một khoảng không gian dành chứa các tên "cúng cơm" khác nhau được dùng trong chương trình. Mục tiêu của namespace là giúp tránh đụng độ về các tên (tên lớp, tên biến, tên hàm, tên thuộc tính v.v..) trong chương trình. Chúng tôi tạm gọi namespace là "địa bàn hoạt động của các tên". `Console` chỉ là một trong vô số kiểu dữ liệu hữu ích thuộc thành phần thư viện của .NET Framework Class Library (FCL). Mỗi lớp đều mang một cái tên, và như thế FCL sẽ chứa hàng ngàn cái tên, chẳng hạn `ArrayList, Dictionary, FileSelector, DataError, Event`, v.v.. Và với thời gian thư viện của FCL sẽ tích tụ lên hàng chục ngàn cái tên, dày đặc như sao trên trời.

Thế là có vấn đề. Không lập trình viên nào có thể nhớ nổi tất cả các tên mà .NET Framework sử dụng, và không chóng thì chầy bạn cũng tạo ra một cái tên mà trước đó Microsoft hoặc một người nào đó đã dùng rồi. Bạn thử nghĩ xem: bạn tự "nghiễn" ra một lớp **Dictionary**, rồi sau đó vỡ lẽ ra là .NET Framework đã đi trước một bước tạo ra một lớp **Dictionary**, thế là hai bên choảng nhau vì một cái tên. Bạn nên nhớ là trên C#, các tên (tên biến, tên hàm, tên thuộc tính v.v..) phải là duy nhất. Có thể thấy thế là bạn cho "sửa lại cái tên", gọi nó chẳng hạn **mySpecialDictionary**; nhưng xem ra bạn thua cuộc. Nội chuyện nơm nớp lo sợ tên mình đặt không biết có trùng với ai hay không, nghĩ cũng đủ thấy mệt.

Để giải quyết vấn đề, người ta đề xuất khái niệm về ***namespace***. Một namespace sẽ giới hạn phạm vi ý nghĩa của một cái tên, nghĩa là tên chỉ có ý nghĩa trong phạm vi được định nghĩa bởi namespace. Giả sử, trong giới anh chị "đá cá lăn dưa" ở chợ Cầu Muối có một trự tên là Xuân tóc đỏ. Nhưng ở chợ Cầu Ông Lãnh cũng lại xuất hiện một tay anh chị khác với cái tên trùng là Xuân tóc đỏ. Trong thực tế, gặp phải trường hợp như thế, để gọi nhau trong ngôn ngữ Việt Nam, ta thường cho kèm theo tên là địa danh, do đó ta gọi Xuân tóc đỏ Cầu Muối hoặc Xuân tóc đỏ Cầu Ông Lãnh. Không lầm vào đâu được. namespace chính là chợ Cầu Muối hoặc chợ Cầu Ông Lãnh.

Cũng tương tự như thế, nếu bạn biết .NET Framework có một lớp `Dictionary` trong lòng namespace `System.Collections`, và bạn cũng đã tạo ra một lớp `Dictionary` trong lòng namespace `ProgCSharp.DataStructures`. Trong thí dụ 2-1, tên của đối tượng `Console` được hạn chế trong namespace `System` bằng cách dùng dòng lệnh sau đây:

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

2.2.5 Tác tử dấu chấm (dot operator) "."

Trên thí dụ 2-1, tác tử dấu chấm (.) được dùng truy xuất một hàm hành sự (và dữ liệu) trong một lớp (trong trường hợp này là hàm hành sự `WriteLine()`), và để giới hạn tên lớp vào một namespace riêng biệt (trong trường hợp này, để phân biệt `Console` trong lòng namespace `System`). Cấp cao nhất là namespace `System` (chứa tất cả các đối tượng `System` mà .NET Framework cung cấp); kiểu dữ liệu `Console` hiện hữu trong lòng namespace này, còn hàm hành sự `WriteLine()` là một hàm thành viên của lớp `Console`.

Trong nhiều trường hợp, namespace thường được chia nhỏ thành những subnamespace. Thí dụ namespace `System` bao gồm một số subnamespace như `Configuration`, `Collections`, `Data` v.v.. chẳng hạn, trong khi namespace `Collections` cũng lại được chia thành nhiều subnamespace khác.

Nói tóm lại, namespace có thể giúp bạn tổ chức các kiểu dữ liệu của bạn, chia nhỏ chúng thành những "khoảng không gian" nhỏ về tên có thể kiểm soát được. Khi bạn viết một chương trình C# phức tạp, bạn nên tạo riêng cho mình một cây đẳng cấp namespace, xuống sâu bao nhiêu cũng được; mục đích của namespace là giúp bạn chia để trị sự phức tạp của đẳng cấp đối tượng của bạn.

Trong Chương 1, "Visual C# và .NET Framework", mục 1.5.2.1 "Một vòng rảo qua .NET Namespace", chúng tôi đã liệt kê ra những namespace thông dụng của C# .NET mà bạn sẽ gặp phải. Đề nghị bạn xem lại.

2.2.6 Từ chốt using

Thay vì viết từ **System** trước **Console**, bạn có thể khai báo bạn sử dụng những kiểu dữ liệu từ namespace **System** bằng cách viết câu lệnh như sau:

```
using System;
```

ngay ở đầu bảng liệt kê (listing) như theo thí dụ 2-3:

Thí dụ 2-3: Sử dụng từ chốt using

```
using System;

class XinChao
{
    static void Main()
    {
        // Dùng đến đối tượng console của namespace System
        Console.WriteLine("Xin Chào Bà Con!");
    }
}
```

Bạn để ý câu lệnh **using System;** được đặt trước phần khai báo lớp XinChao.

Mặc dù bạn có thể chỉ định bạn đang sử dụng namespace `System`, khác với một số ngôn ngữ khác bạn không thể chỉ định bạn đang sử dụng đối tượng `System.Console`. Thí dụ 2-4 sau đây sẽ không chịu biên dịch:

Thí dụ 2-4: Đoạn mã không chịu biên dịch (C# bất hợp lệ)

```
using System.Console;   // sai

class XinChao
{
    static void Main()
    {
        // Dùng đến đối tượng console của namespace System
        WriteLine("Xin Chào Bà Con!");
    }
}
```

Trình biên dịch sẽ cho phát sinh thông điệp sai lầm như sau:

```
error CS0138: A using namespace directive can only be applied to
namespaces;
'System.Console' is a class not a namespace
```

Từ chốt **using** giúp bạn giảm thiểu việc phải gõ những namespace trước các hàm hành sự hoặc thuộc tính; bạn nên sử dụng **using** với những namespace "bẩm sinh" (built-in) cũng như với namespace riêng của công ty bạn đang làm việc, nhưng không nên dùng với những phần mềm cấu kiện (component) mua trôi nổi ở ngoài thị trường phần mềm.

2.2.7 Phân biệt chữ hoa chữ thường (case sensivity)

C# là một ngôn ngữ phân biệt chữ hoa chữ thường, ta gọi là case-sensitive, có nghĩa là **writeline** khác với **WriteLine** cũng như khác với **WRITELINE**. Rất tiếc là khác với Visual Basic (VB), môi trường triển khai IDE của C# sẽ không sửa chữa các lỗi về chữ hoa chữ thường của bạn; nếu bạn viết hai lần cùng từ nhưng với chữ hoa chữ thường khác đi, thì có thể bạn gây ra trong chương trình một bug khó lòng phát hiện.

Để ngăn ngừa việc mất thời giờ và công sức do những lỗi không đâu vào đâu, bạn nên đề ra những qui ước trong việc đặt tên cho các biến, hàm, hằng v.v.. Trong bộ sách này chúng tôi dùng hai kiểu ký hiệu đặt tên:

- ***Ký hiệu "lưng gù lạc đà"*** (Camel notation) theo đấy mẫu tự (letter) đầu tiên của từ (word) đầu tiên viết theo chữ thường. Thí dụ: **someVariableName**. Nếu ta chỉ cứ nhìn vào mẫu tự đầu tiên của mỗi từ thì bạn thấy cái lưng gù của con lạc đà. *Kiểu ký hiệu này dành viết các tên biến*, như **luongCanBan, maSoNhanVien, confirmationDialog**, v.v..

- ***Ký hiệu Pascal***: theo theo đấy mẫu tự đầu tiên của mỗi từ được viết theo chữ hoa. Thí dụ: **SomeFunction.** *Kiểu ký hiệu Pascal này dành để viết các tên hàm, hằng và thuộc tính, namespace, hoặc tên lớp*.

2.2.8 Từ chốt static

Hàm hành sự **Main()** trên thí dụ 2-1, có một từ chốt **static**, nằm trước **void**, mà chúng tôi chưa giải thích:

```
static void Main()
```

Từ chốt `static` cho biết bạn có thể triệu gọi hàm **Main()** mà *khỏi phải tạo ra một đối tượng* kiểu **XinChao.** Theo thường lệ, bạn phải tạo một thể hiện đối tượng của một lớp nào đó trước khi triệu gọi hàm thành viên của đối tượng này. Với từ chốt **static** thì bạn miễn làm chuyện này; nghĩa là **Main()** không hoạt động trên một thể hiện riêng biệt của lớp **XinChao** và do đó có thể được triệu gọi mà khỏi phải trước tiên thể hiện lớp. Có vẻ hơi khó hiểu, nhưng tạm thời bạn chấp nhận hiểu như thế. Những thắc mắc sẽ lần lượt được giải thích về sau. (Xem Chương 5, "Lớp và Đối tượng", mục 5.9, "Cuộc sống bên trong của các đối tượng"). Một trong những vấn đề khi học một ngôn ngữ mới, là bạn phải học những chức năng cao cấp trước khi bạn hiểu thấu đáo về chúng. Tạm thời, bạn chấp nhận những gì chúng tôi trình bày.

2.3 Các "biến tấu" khác nhau của hàm Main()

Ở đầu chương, bạn đã làm quen với một phiên bản C# về chương trình "kinh điển" "Xin Chào Bà Con!" cho hiển thị lên màn hình chuỗi chữ Xin Chào Bà Con!.

```
// Một chương trình "Xin Chào Bà Con!" viết theo C#
class XinChao
{
   public static void Main()
   {
      // Dùng đến đối tượng console của System
      System.Console.WriteLine("Xin Chào Bà Con!");
   }
}
```

Xin nhắc lại: (1) chương trình C# phải chứa một hàm hành sự **Main()**, được xem như là điểm đột nhập vào chương trình, là nơi bạn tạo những đối tượng rồi cho thi hành các hàm hành sự khác. (2) Hàm hành sự **Main()** là một hàm static nằm trong lòng một lớp (hoặc trong một struct). Trong trường hợp của ta, lớp này mang tên **XinChao**. (3) Có 3 cách để khai báo hàm **Main()** mà ta đã làm quen qua một cách:

- Cách thứ nhất: không trả về kiểu dữ liệu, và không có đối mục (argument) dạng như sau:

```
// không trả về kiểu dữ liệu, và không có đối mục
public static void Main()
{
   // tạo một vài đối tượng
}
```

- Cách thứ hai: trả về kiểu dữ liệu **int**, không nhận thông số

```
// trả về kiểu dữ liệu int, không nhận thông số
public static int Main()
{
   // tạo một vài đối tượng
   return 0; // trả về một trị cho hệ thống
}
```

- Cách thứ ba: trả về kiểu dữ liệu int, và nhận thông số

```
// trả về kiểu dữ liệu int, và nhận nhận thông số
public static int Main(string [] args)
{
   // Xử lý các đối mục trên command line
```

```
            return 0; // trả về một trị cho hệ thống
        }
```

Thông số trên hàm **Main()** của cách thứ ba, là một bản dãy chuỗi chữ (string array) tượng trưng các đối mục trên "dòng lệnh" (command line). Bạn để ý, khác với C++, bản dãy này không bao gồm tên của tập tin khả thi (exe).

Rõ ràng, việc chọn một trong 3 dạng hàm **Main()** tùy thuộc vào hai câu hỏi: Trước tiên, bạn có cần xử lý bất cứ thông số nào trên command line hay không. Nếu có, thì các thông số này được trữ trên một bản dãy chuỗi chữ. Kế đến, bạn có muốn trả về một trị hay không cho hệ thống một khi hàm **Main()** hoàn thành nhiệm vụ; nếu có, bạn cần trả về một kiểu dữ liệu số nguyên thay vì void.

2.3.1 Xử lý các thông số của Command Line

Gỉa sử bạn muốn nhật tu chương trình XinChao cho xử lý những thông số trên command line. Bạn viết chương trình lại như sau, thí dụ 2-5:

Thí dụ 2-5: Hàm Main() có thông số

```
using System;

class XinChao
{
    public static int Main(string[] args)
    {
        // In ra những thông số
        for (int x = 0; x < args.Length; x++)
        {
            Console.WriteLine("Thông số: {0}", args[x]);
        }

        Console.WriteLine("Xin Chào Bà Con!");
        return 0;
    }
}
```

Ở đây `args` là một bản dãy chuỗi chữ, có chiều dài do thuộc tính `Length` cho biết. Thuộc tính `Length` thuộc lớp `System.Array` **(**xem Chương 10, "Bản dãy, Indexer và Collection). Câu lệnh `for` là một vòng lặp cho phép bạn rảo qua bản dãy `args` để in ra từng thông số một. Vì bản dãy là một tập hợp (collection) nên thay vì dùng `for` bạn có thể sử dụng câu lệnh `foreach` như sau, thí dụ 2-6:

Thí dụ 2-6: Hàm Main() có thông số, dùng vòng lặp foreach

```
using System;

class XinChao
{
   public static int Main(string[] args)
   {
      // In ra những thông số
      foreach (string s in args)
      {
         Console.WriteLine("Thông số: {0}", s);
      }

      Console.WriteLine("Xin Chào Bà Con!");
      return 0;
   }
}
```

2.3.2 Có nhiều hàm Main()

Khi một ứng dụng console hoặc Widows viết theo C# được biên dịch, theo mặc nhiên trình biên dịch sẽ đi tìm đúng một hàm **Main()** trên bất cứ lớp nào để làm cho lớp này là điểm đột nhập vào chương trình. Nếu có nhiều hơn một hàm **Main()**, thì trình biên dịch phát ra một sai lầm. Thí dụ, ta thử xem đoạn mã sau đây:

```
// MainExample.cs
using System;

namespace Prog_CSharp
{
   class Client
   {
      public static int Main()
      {
         MathExample.Main();
         return 0;
      }
   }

   class MathExample
   {
      static int Add(int x, int y)
      {
         return x + y;
      }

      public static int Main()
      {
```

```
            int i = Add(5,10);
            Console.WriteLine(i);
            return 0;
        }
    }
}
```

Thí dụ trên chứa hai lớp, **Client** và **MathExemple**, cả hai đều có một hàm hành sự **Main()**. Nếu ta cho biên dịch thí dụ trên theo thể thức thông thường (sử dụng csc MainExample.cs), thì ta sẽ nhận được thông điệp sai lầm như sau:

```
   MainExample.cs(8,25): error CS0017: Program 'MainExample.exe' has
more than one entry point defined: 'Prog_CSharp.Client.Main()'
   MainExample.cs(22,25): error CS0017: Program 'MainExample.exe' has
more than one entry point defined: 'Prog_CSharp.MathExample.Main()'
```

Tuy nhiên, bạn có thể cho trình biên dịch biết rõ ra hàm hành sự Main() nào sẽ được dùng làm điểm đột nhập chương trình bằng cách sử dụng switch "**/main"**, kết hợp với tên trọn vẹn (bao gồm cả namespace) của lớp mà hàm Main() hiện "đóng quân":

```
   csc MainExample.cs /main:Prog_CSharp.MathExample
```

2.4 Giới thiệu cách định dạng chuỗi chữ C#

Trong thí dụ 2-5 và 2-6, bạn thấy xuất hiện trong câu lệnh **Console.WriteLine();** nhóm ký tự **{0}**. Nhóm này được gọi là token[26]. .NET đưa vào một kiểu định dạng (formatting) chuỗi chữ mới, thay thế kiểu định dạng hơi kỳ bí của hàm printf() trên C, với những flag khó hiểu "%d", "%s", "%c"v.v.. Ta thử lấy thí dụ 2-7:

Thí dụ 2-7: Định dạng đơn giản

```
using Systems;

class BasicIO
{   public static void Main(string[] args)
    {   …
        int theInt = 90;    // số nguyên
        float theFloat = 9.99; // số thực
        BasicIO myIO = new BasicIO(); // tạo mới một đối tượng
                                      // kiểu BasicIO

        // Định dạng một chuỗi…
        Console.WriteLine("Int là: {0}\nFloat là: {1}\nBạn là: {2}",
```

[26] Trong các sòng bài, token là mấy miếng thẻ tròn tượng trưng cho một số tiền nào đó. Ở đây tượng trưng cho một phần tử gì đó khi phân tích ngữ nghĩa.

```
            theInt, theFloat, myIO.ToString());
    }
}
```

Kết xuất

```
Int là: 90
Float là: 9.99
Bạn là: BasicIO
```

Thông số đầu tiên của hàm `WriteLine()` tượng trưng cho một chuỗi định dạng (format string) chứa những token {0}, {1}, {2} được gọi là token "giữ chỗ sẵn" (placehoder). Phần còn lại của `WriteLine()` sau dấu phẩy đầu tiên là những trị cần được lần lượt (theo thứ tự 0, 1, 2…) chen vào những placeholder. Trong trường hợp này là một số nguyên, một số thực, một chuỗi chữ. Ngoài ra, bạn cũng có thể nhận biết là `WriteLine()` bị "nạp chồng" (overloaded) [27] cho phép bạn khai báo những trị của placeholder như là một bản dãy các đối tượng (array of objects). Do đó, bạn có thể cho tượng trưng bao nhiêu mục tin cũng được cần được "dộng" vào chuỗi định dạng như sau:

```
// Cho điền placeholder sử dụng một bản dãy các đối tượng
object[] doLucLangLucChot = {"Hello", 20.9, 1,
            "Hi hi", "83", 99.99933};
Console.WriteLine("Đồ Lục Lăng Lục Chốt: {0}, {1}, {2},
                {3}, {4}, {5}", doLucLangLucChot );
```

Mỗi token giữ chỗ có thể tùy chọn chứa những ký tự định dạng khác nhau (hoặc chữ hoa hoặc chữ thường) như theo Bảng 2-1 sau đây:

Bảng 2-1: Các ký tự định dạng C#

Ký tự định dạng C#	Ý nghĩa
C hoặc c	Dùng định dạng tiền tệ (currency). Theo mặc nhiên, flag sẽ có một tiền tố là dấu $ nằm trước trị. Tuy nhiên, điều này có thể bị thay đổi bởi việc sử dụng một đối tượng **NumberFormatInfo.**
D hoặc d	Định dạng số thập phân (decimal number). Flag này cũng có thể khai báo một số tối thiểu digit dùng điền (pad) zero lên trị số. Chuyển đổi một số nguyên thành cơ số 10, và điền zero vào nếu có khai báo độ tinh xác (precision).
E hoặc e	Ký hiệu số mũ (exponential notation) khoa học. Độ tinh xác cho đặt để số lẻ thập phân (6 theo mặc nhiên).

[27] Nếu bạn chưa hiểu từ này thì cũng không sao. Chương 5 sẽ giải thích rõ.

F hoặc f	Định dạng số dấu chấm cố định (fixed point). Độ tinh xác điều khiển số số lẻ thập phân.
G hoặc g	General. Dùng định dạng một con số về dạng thức cố định F hoặc lũy thừa E.
N hoặc n	Định dạng số cơ bản (với dấu phẩy như là phân thành ngàn). thí dụ 32.757.56
P hoặc p	Định dạng tỉ lệ phần trăm (percent format).
X hoặc x	Định dạng số thập lục (hexa).

==========

Những ký tự định dạng kể trên được đặt trong lòng một placeholder nào đó sử dụng một dấu hai chấm (:), thí dụ , {0:C}, {1:d}, {2:X}, v.v.. Bạn để ý là các token được nằm trong cặp dấu ngoặc nghéo {}. Để minh họa, giả sử bạn nhật tu hàm Main() với lô gic sau đây:

```
// Ta thử dùng các ký tự định dạng kể trên
public static void Main(string[] args)
{
   …
   Console.WriteLine("C Format: {0:C}", 99989.987);
   Console.WriteLine("D9 Format: {0:D9}", 99999);
   Console.WriteLine("E Format: {0:E}", 99999.76543);
   Console.WriteLine("F3 Format: {0:F3}", 99999.9999);
   Console.WriteLine("N Format: {0:N}", 99999);
   Console.WriteLine("X Format: {0:X}", 99999);
   Console.WriteLine("x Format: {0:x}", 99999);
```

Kết xuất

C Format: $99,989.99
D9 Format: 000099999
E Format: 9.999977E+004
F3 Format: 100000.000
N Format: 99,999.00
X Format: 1869F
x Format: 1869f

Các ký tự định dạng trên không giới hạn vào hàm hành sự **Console.WriteLine()**. Những flag trên có thể sử dụng trong phạm vi hàm **String.Format()** cũng dùng để định dạng. Điều này rất hữu ích khi bạn cần tạo một chuỗi chữ chứa những trị số trong ký ức rồi cho hiển thị chuỗi này về sau:

```
// sử dụng một hàm hành sự static String.Format
// để tạo một chuỗi chữ mới
string formStr;
formStr = String.Format("bạn muốn có {0:C} trong tài khoản
                         của bạn ?", 99989,987);
Console.WriteLine(formStr);
```

2.5 Triển khai "Xin Chào Bà Con!"

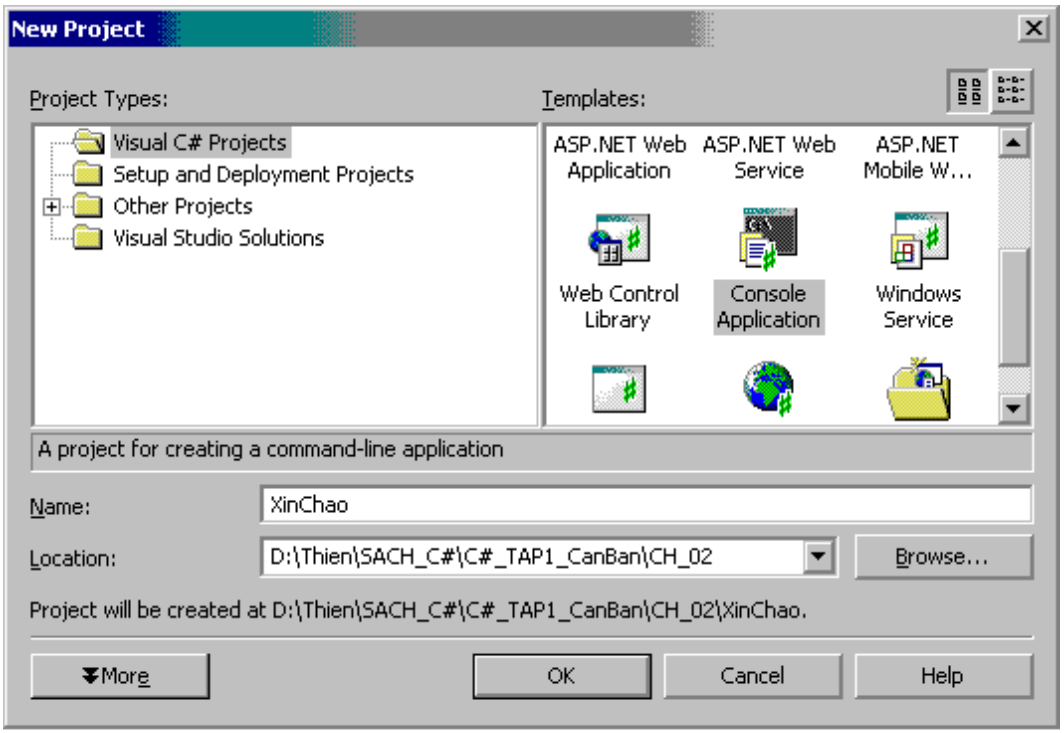
Trong tập sách này, có ít nhất 2 cách để khỏ vào chương trình, cho biên dịch và chạy thi hành chương trình: bạn sử dụng IDE (Integrated Development Environment – Môi trường Triển khai Tích hợp) của Visual Studio .NET, hoặc dùng một trình soạn thảo văn bản (text editor) và trình biên dịch dòng lệnh (command-line compiler), kèm theo một số công cụ mà chúng tôi sẽ đề cập sau.

Mặc dù bạn có thể triển khai phần mềm ngoài Visual Studio .NET, IDE đem lại cho bạn vô số lợi điểm. Bao gồm việc hỗ trợ indentation (canh thụt văn bản), Intellisense (hỗ trợ cú pháp), color coding (tô màu khác nhau đối với các từ chốt, biến … trên đoạn mã), kết nối với các tập tin help. Điểm quan trọng là IDE có sẵn một bộ gỡ rối (debugger) chương trình cực mạnh và vô số công cụ khác.

Mặc dù chúng tôi đặt giả thuyết là bạn sẽ sử dụng Visual Studio .NET, tập sách này đặt trọng tâm vào ngôn ngữ và sàn diễn của ngôn ngữ thay vì các công cụ. Bạn có thể sao chép các thí dụ trên tập sách này lên Notepad, hoặc một trình soạn thảo văn bản nào đó, Word chẳng hạn, rồi cho trữ dưới dạng tập tin văn bản, rồi cho biên dịch với trình biên dịch C# command-line được cung cấp với .NET Framework SDK, mang tên **csc.exe**. Bạn để ý là trong những chương đi sau, chúng tôi sử dụng các công cụ của Visual Studio .NET để tạo những Windows Forms và Web Forms, nhưng nếu bạn muốn bạn cũng có thể viết bằng tay sử dụng Notepad.

2.5.1 Hiệu đính "Xin Chào Bà Con!"

Muốn tạo một chương trình "Xin Chào Bà Con!" trên IDE, bạn chọn Visual Studio .NET từ trình đơn **Start** hoặc từ một desktop icon, rồi sau đó chọn **File | New | Project** từ thanh công cụ trình đơn (menu toolbar). Lúc này cửa sổ **New Project** sẽ được gọi vào. Hình 2-1 cho thấy cửa sổ New Project.



Hình 2-1: Tạo một ứng dụng C# console trên Visual Studio .NET

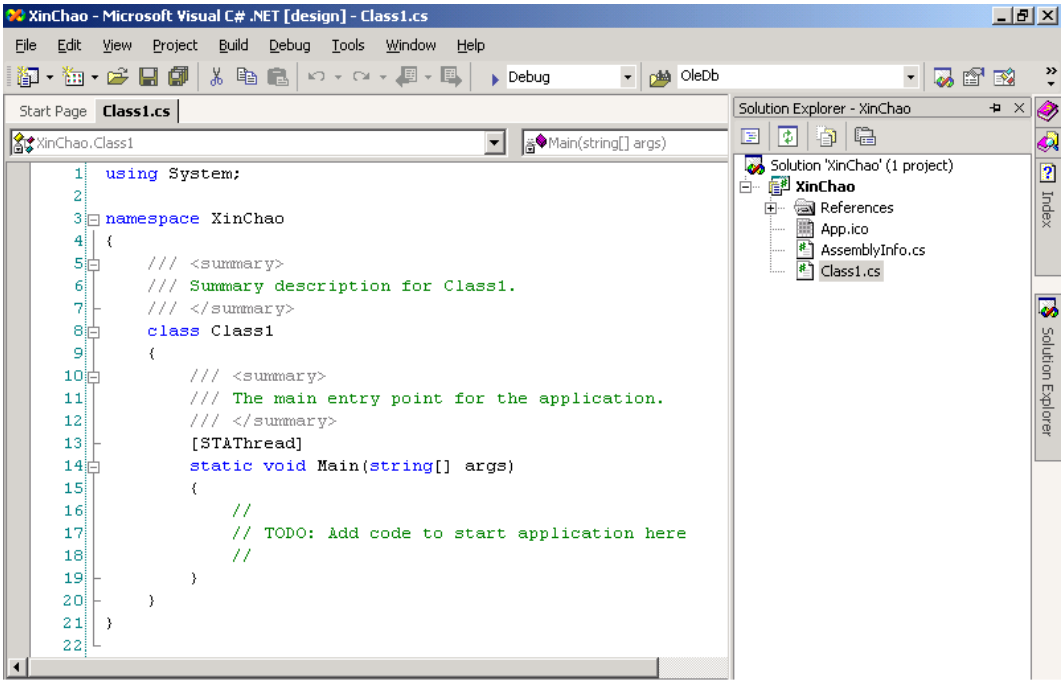
Bạn thấy cửa **sổ New Project** này có hai khung: phía tay trái là **Project Types**, phía tay phải là **Templates**. Muốn mở ứng dụng của bạn, bạn chọn **Visual C# Projects** trên khung **Project Types** và **Console Application** trên khung **Templates. Bạn** có thể khỏ tên dự án vào ô Name (ở đây ta khỏ XinChao), và chọn một thư mục để ghi trữ tập tin (ở đây ta chọn D:\Thien\SACH_C#\...). Rồi ấn nút <OK>. Và một cửa sổ khác lại hiện lên, như theo Hình 2-2.

Bạn để ý Visual Studio .NET tạo ra một namespace dựa trên tên dự án mà bạn cung cấp (XinChao), và thêm một câu lệnh **using System** vì hầu hết các chương trình bạn viết đều cần kiểu dữ liệu lấy từ namespace **System**.

Visual Studio .NET tạo một lớp cho mang tên **Class1** mà bạn hoàn toàn tự do thay đổi lại tên. Khi bạn đổi tên lớp thì cũng nên đổi lại tên tập tin (*Class1.cs*). Để viết lại thí dụ 2-1, chẳng hạn, bạn thay đổi tên lớp **Class1** thành **XinChao,** và đổi tên tập tin Class1.cs (được liệt kê trên cửa sổ Solution Explorer) thành *XinChao.cs.*

Cuối cùng, Visual Studio .NET tạo một khung sườn (skeleton) chương trình, trọn vẹn với dòng chú giải //TODO để cho bạn bắt đầu. Bạn cho gỡ bỏ trên hàm Main() phần đối

mục **string[] args**, xem như hàm Main() không có thông số, rồi bạn chép 3 hàng vào thân của Main():



Hình 2-2: Editor mở bởi dự án mới của Bạn

```
// Dùng đến đối tượng console của System
System.Console.WriteLine("Xin Chào Bà Con!");
System.Console.ReadLine();
```

Bạn để ý là chúng tôi thêm dòng lệnh `System.Console.ReadLine();` là để cho màn hình nằm một chỗ cho bạn có thời giờ nhìn xem, nếu không thì nó biến mất trong tíc tắc.

Nếu bạn không sử dụng Visual Studio .NET, thì bạn cho mở Notepad, khỏ vào đoạn mã trên thí dụ 2-1, rồi cho cất trữ dưới dạng tập tin văn bản cho mang tên XinChao.cs. Tất cả các tập tin mã nguồn (source code) C# đều mang phần đuôi (extension) là **.cs**, tắt chữ C Sharp.

Bạn để ý: Khi bạn sử dụng Visual Studio .NET IDE, nếu bạn muốn dùng tiếng Việt, thì bạn cho cất trữ tập tin .cs bằng cách ra lệnh **File | Save <Class1.cs> As…** Lúc này khung đối thoại Save File As hiện lên, bạn click lên ô mũi tên chúc xuống cạnh nút Save để cho hiện lên trình đơn shortcut, và bạn chọn Save with Encoding thì một khung đối thoại Advanced Save Options cho phép bạn chọn mục Unicode(UTF8 with signature) -

CodePage 65001, rồi ấn OK. Như vậy bạn có thể dùng tiếng Việt trên Visual Studio .NET.

2.5.2 Cho biên dịch và chạy "Xin Chào Bà Con!"

Có nhiều cách để biên dịch và chạy thi hành một chương trình "Xin Chào Bà Con!" trong lòng Visual Studio .NET. Điển hình, bạn có thể thực hiện mỗi công tác bằng cách chọn lệnh trên thanh công cụ trình đơn Visual Studio .NET, hoặc bằng cách dùng nút ấn, và trong nhiều trường hợp sử dụng đến các phím tắt (shortcut key).

Thí dụ, muốn biên dịch chương trình "Xin Chào Bà Con!", bạn ấn tổ hợp phím **<Ctrl+Shift+B>** hoặc chọn trên trình đơn: **Build | Build Solution.** Một phương án thay thế, bạn có thể bấm tắt (click) lên nút **Build** trên thanh công cụ **Build** (thanh công cụ này hiện lên khi bạn chọn **View | Toolbars | Build**).

Muốn chạy chương trình đã biên dịch xong, không cần đến debugger, bạn có thể ấn <Ctrl+F5>, hoặc chọn **Debug | Start Without Debugger** trên thanh trình đơn. Bạn cũng có thể chạy chương trình khỏi phải build trước, tùy theo chọn lựa được đặt để (**Tools | Options**), IDE sẽ cất trữ tập tin, cho build chương trình rồi cho chạy, với khả năng từng bước sẽ xin phép bạn.

Chúng tôi khuyên bạn nên dành thời giờ khám phá môi trường triển khai chương trình IDE của Visual Studio .NET. Chương 14, "Lập trình trên môi trường .NET", tập sách này sẽ đề cập đến IDE. Đây sẽ là công cụ chính nếu bạn là nhà viết phần mềm chuyên nghiệp, bạn nên biết sử dụng thuần thục, nhuần nhuyễn.

Bạn để ý: Khi bạn cho chạy chương trình XinChao trên IDE theo ứng dụng console, thì màn hình hiện lên rồi biến mất không kịp cho bạn nhìn xem. Để khắc phục điều này ở cuối chương trình trước khi thoát ra, bạn thêm một dòng lệnh:

```
Console.ReadLine();
```

thì lúc này màn hình sẽ không biến mất. Bạn tha hồ ngắm nghía chương trình đầu tay của bạn.

Muốn biên dịch và chạy chương trình "Xin Chào Bà Con!" sử dụng trình biên dịch command-line C#, **csc.exe** bạn theo các bước sau đây:

1. Khỏ thí dụ 2-1 trên Notepad, rồi cho cất trữ dưới dạng tập tin XinChao.cs.

2. Cho mở cửa sổ command, bằng cách Start | Run rồi khỏ vào **cmd**. Cửa sổ C:\WINNT\System32\cmd.exe hiện lên toàn đen.

3. Từ command line, bạn khỏ vào: **csc XinChao.cs**. Bước này sẽ biên dịch thành chương trình khả thi (EXE). Nếu có gì sai, thì trình biên dịch sẽ cho hiện lên các thông điệp sai.

4. Muốn chạy chương trình, bạn khỏ vào: **XinChao**. Bạn sẽ thấy dòng chữ "Xin Chào Bà Con!" hiện lên như theo hình 2-3

Hình 2-3: Kết quả XinChao.exe

Bây giờ ta thử tìm hiểu thêm trình biên dịch **csc.exe** (C Sharp Compiler). Trong phần giải thích biên dịch và chạy chương trình XinChao.cs trên, ta đơn giản khỏ:

```
csc XinChao.cs
```

Thật ra, trình biên dịch **csc.exe** cần một số flag cho biết những mục chọn (option) mà trình biên dịch sẽ hoạt động. Trước khi biết qua những mục chọn chủ chốt của trình biên dịch C#, điều đầu tiên là bạn phải hiểu cách khai báo loại tập tin kết xuất (output file) mà bạn muốn xây dựng, nghĩa là Console Application, DLL, Windows EXE application, v.v.. Mỗi mục chọn sẽ được gán cụ thể một flag được gởi cho csc.exe như là thông số dòng lệnh. Xem bảng 2-3.

Bảng 2-3: Output Options của trình biên dịch C#: csc.exe

File Output Option	Ý nghĩa
/doc	Yêu cầu csc.exe xử lý các chú giải (comment) trong mã nguồn thành một tập tin XML.
/out	Dùng khai báo tên tập tin kết xuất (chẳng hạn MyAssembly.dll, WordProcessingApp.exe, v.v..) Theo mặc nhiên, tên tập tin kết xuất sẽ giống như tên tập tin nhập *.cs, do đó /out có thể bỏ qua không kê khai ra.
/target:exe	Flag này sẽ tạo một ứng dụng EXE chạy trên console (nghĩa là một ứng dụng DOS). Đây là loại tập tin kết xuất mặc nhiên, nên có thể bỏ qua không kê khai ra. Bạn có thể viết ngắn gọn: /t:exe
/target:library	Mục chọn này tạo một DLL assembly, với một manifest đi kèm theo. Bạn có thể viết ngắn gọn: /t:library

/target:module	Mục chọn này sẽ tạo một "module", nghĩa là một DLL assembly nhưng có manifest đi kèm theo. Module này được dùng trong multifile assembly. Bạn có thể viết ngắn gọn: /t:module.
/target:winexe	Mặc dù bạn tự do xây dựng những ứng dụng Windows sử dụng flag /target:exe, mục chọn này che dấu cửa sổ console xuất hiện khi ứng dụng chạy. Bạn có thể viết ngắn gọn: /t:winexe.

Do đó, muốn biên dịch XinChao.cs thành một ứng dụng console, bạn phải sử dụng lệnh sau đây, với flag kết xuất nằm trước tên tập tin:

```
csc /target:exe XinChao.cs    hoặc
csc /t:exe XinChao.cs
```

Vì /t:exe là mặc nhiên đối với trình biên dịch C#, bạn cũng có thể biên dịch XinChao.cs, bằng cách đơn giản ra lệnh:

```
csc XinChao.cs
```

2.6 Sưu liệu dựa trên XML

Trong việc phát triển phần mềm, vấn đề gai góc nhất là ***sưu liệu*** (documentation). Lập trình viên sau khi viết và chạy thử nghiệm xong chương trình, là có chiều hướng quên đi việc viết sưu liệu, vì đây là một công việc rất nhàm chán. Do đó, giữ cho việc sưu liệu đi cùng nhịp với việc thi công chương trình là cả một thách thức. Người ta đi đến một cách là làm thế nào viết sưu liệu ngay trên mã nguồn cùng lúc viết đoạn mã (như vậy đầu óc người viết đang còn nóng hổi ý tưởng) rồi sau đó trích phần sưu liệu qua một tập tin khác. Tập tin sưu liệu này ở dạng XML.

Nếu ai đã từng viết Java, thì có thể đã biết qua trình tiện ích Javadoc lo việc sưu liệu. Trình này biến phần sưu liệu trên mã nguồn thành một tập tin HTML. Trong khi ấy, C# thì lại hỗ trợ một dạng thức sưu liệu dựa trên XML (Extended Markup Language). C# không dùng HTML, với lý do là XML là một công nghệ rất có khả năng. Tiến trình định dạng một mã nguồn thành một tập tin XML sẽ do trình biên dịch C# đảm nhiệm thay vì một trình tiện ích đơn độc.

Khi bạn muốn sưu liệu các lớp của bạn theo XML, bước đầu tiên là sử dụng đến một cú pháp đặc biệt: là ba dấu forward slash (///) thay vì hai (//) kiểu C++ hoặc (/*..*/) kiểu C. Sau ba dấu slash này bạn có thể dùng bất cứ XML tag nào được liệt kê sau đây, Bảng 2-4:

Bảng 2-4: Các tag của XML

Tag sưu liệu XML	Ý nghĩa
<c>	Cho biết phần văn bản nằm trong lòng một mô tả phải được đánh dấu là đoạn mã.
<code>	Cho biết nhiều hàng phải được đánh dấu là đoạn mã.
<example>	Cho biết một thí dụ sử dụng một lớp hoặc một hàm
<exception>	Cho biết những biệt lệ nào một lớp có thể tung ra.
<list>	Dùng chèn một danh sách liệt kê vào tập tin sưu liệu.
<param>	Mô tả một thông số đối với một hàm thành viên.
<paramref>	Gắn một tag XML nào đó với một thông số cụ thể trên một văn bản khác.
<permission>	Cho biết phép được truy xuất đối với một thành viên lớp.
<Remarks>	Một mô tả dài dòng đối với một thành viên nào đó.
<returns>	Cho biết trị trả về của một thành viên.
<see cref=”member”>	Cho biết một kết nối (link) về một thành viên hoặc vùng mục tin trong môi trường biên dịch hiện hành.
<seealso cref=”member”>	Cho biết một kết nối (link) về một đoạn “see also” của sưu liệu.
<Summary>	Cho biết một mô tả ngắn gọn liên quan đến một item nào đó.
<value>	Mô tả trị của một thuộc tính.

Sau đây là một lớp Payroll (Lương) với vài chú giải dựa theo XML. Bạn để ý các tag <summary>, <paramref> và <param>

Thí dụ 2-8: Sưu liệu thông qua XML

```
using System;

namespace Payroll
{
    /// <summary>
    /// Lớp NhanVien chứa dữ liệu liên quan đến một nhân viên.
    /// Lớp này chứa một <see cref="String">string</see>
    /// </summary>
    public class NhanVien
    {
        /// <summary>
        /// Hàm constructor đối với một thể hiện NhanVien. Ghi nhận
        /// <paramref name="ten">ten2</paramref> là một string.
        /// </summary>
        /// <param name="msnv">Mã số Nhân viên</param>
        /// <param name="tenHo">Tên Nhân viên</param>
        public NhanVien(int msnv, string tenHo)
        {
            this.msnv = msnv;
```

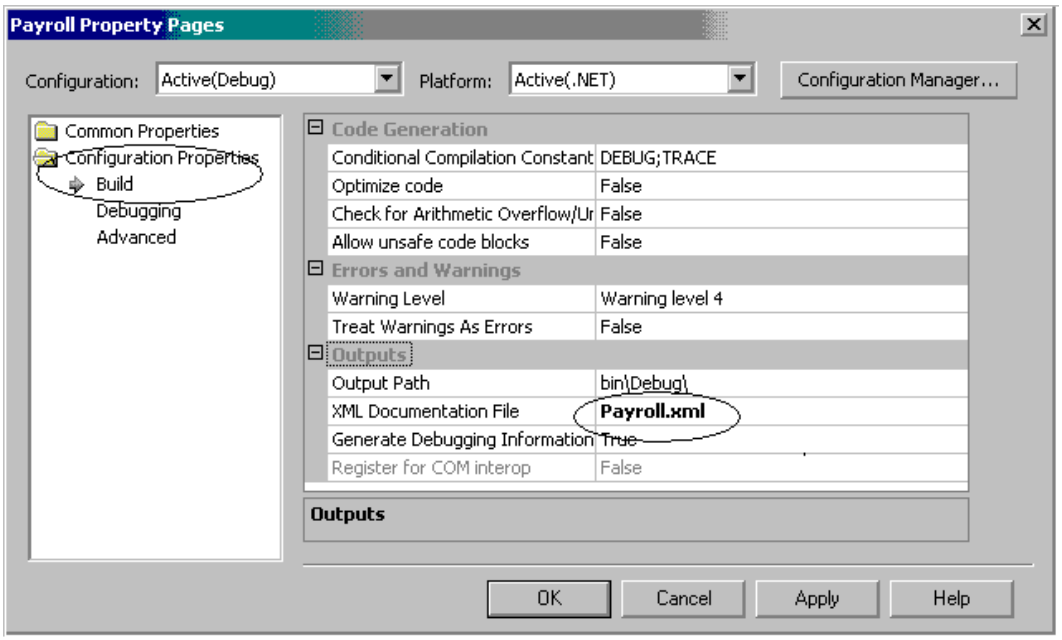
```
            this.tenHo = tenHo;
        }

        /// <summary>
        /// Hàm constructor không thông số đối với thể hiện NhanVien
        /// </summary>
        /// <remarks>
        /// <seealso cref="NhanVien(int, string)">NhanVien(int,
                                                    string)</seealso>
        /// </remarks>
        public NhanVien()
        {
            msnv = -1;
            tenHo = null;
        }
        private int msnv;
        private string tenHo;

        static void Main()
        {
            NhanVien nv = new NhanVien(1234, "Dương Quang Thiện");
            Console.WriteLine("Mã số: {0}, Tên họ: {1}}",
                              nv.msnv, nv.tenHo);
            Console.ReadLine();
        }
    }
}
```

Một khi bạn đã có sưu liệu XML, bạn có thể khai báo flag **/doc** như là input đối với trình biên dịch C#. Bạn để ý phải khai báo tên của tập tin kết xuất XML cũng như tập tin input:

```
csc /doc:payroll.xml payroll.cs
```

Như bạn có thể hy vọng, Visual Studio .NET IDE cho bạn khả năng khai báo tên tập tin XML để mô tả các lớp của bạn. Muốn thế, bạn click nút **Properties** trên cửa sổ Solution Explorer, để cho hiện lên khung đối thoại **Property Page**.(xem Hình 2-4).

Khi khung đối thoại **Property Page** hiện lên, bạn chọn mục **Build** trên thư mục **Configuration Properties**. Bạn tìm mục hiệu đính "XML Documentation File" hiện để trắng. Bạn khỏ vào "Payroll.xml". Đây là tên tập tin XML sẽ chứa những định nghĩa XML đối với những kiểu dữ liệu trên dự án Payroll. Tập tin XML này sẽ tự động kết sinh khi bạn cho Build lại dự án.



Hình 2-4: Khung đối thoại Property Page

Bạn có thể cho hiển thị nội dung của tập tin Payroll.xml. Bạn chỉ cần lên Solution Explorer, click nút **Show All Files**, chọn tập tin Payroll.xml, right-click lên tên tập tin này rồi chọn mục View Code. Sau đây là bảng liệt in tập tin Payroll.xml. Bạn tha hồ mà ngắm nghía, so sánh với bảng mã nguồn:

```
<?xml version="1.0"?>
<doc xmlns="http://tempuri.org/Payroll.xsd">
  <assembly>
    <name>Payroll</name>
  </assembly>
  <members>
    <member name="T:Payroll.NhanVien">
     <summary>
      Lớp NhanVien chứa dữ liệu liên_quan đến một nhân viên.
      Lớp này chứa một <see cref="T:System.String">string</see>
          </summary>
    </member>
    <member name="M:Payroll.NhanVien.#ctor(System.Int32,
                                           System.String)">
     <summary>
      Hàm constructor đối với một thể hiện NhanVien. Ghi nhận
      <paramref name="ten">ten2</paramref> là một string.
     </summary>
      <param name="msnv">Mã số Nhân viên</param>
      <param name="tenHo">Tên Nhân viên</param>
    </member>
```

```
        <member name="M:Payroll.NhanVien.#ctor">
        <summary>
           Hàm constructor không thông số_đối với thể hiện NhanVien
        </summary>
        <remarks>
        <seealso cref="M:Payroll.NhanVien.#ctor(System.Int32,
                        System.String)">NhanVien(int, string)
                </seealso>
        </remarks>
      </member>
    </members>
</doc>
```

Trên bảng liệt in này, bạn để ý vài tiền tố mang các ý nghĩa như sau:

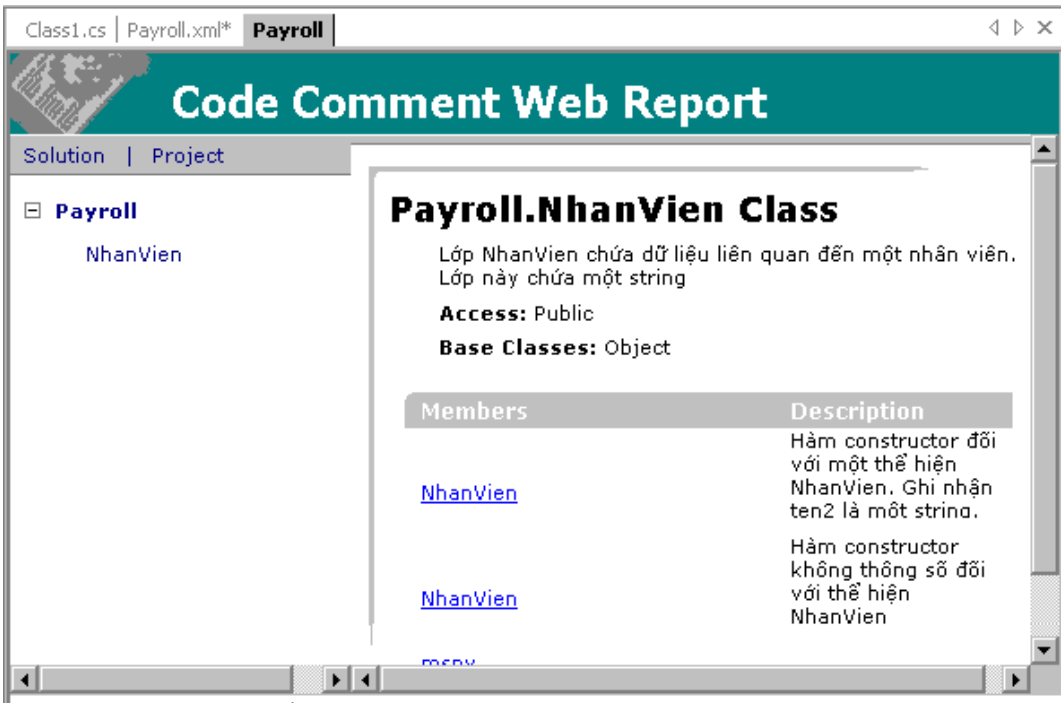
Bảng 2-4: Các ký tự định dạng XMl

Ký tự định dạng	Ý nghĩa
N	Cho biết một namespace
T	Tượng trưng cho type, một kiểu dữ liệu (lớp, giao diện, struct, v.v..)
F	Tượng trưng cho field, một vùng mục tin.
P	Tượng trưng cho property, một thuộc tính (kể cả indexer).
M	Tượng trưng cho method, một hàm hành sự (bao gồm hàm constructor và tác tử nạp chồng.
E	Cho biết là một event, một tình huống.
!	Tượng trưng cho một chuỗi sai lầm cung cấp thông tin liên quan đến sai lầm. Trình biên dịch C# kết sinh thông tin sai lầm đối với những kết nối không thể giải quyết được.

Tới đây bạn có một tập tin XML thô có thể được vẽ theo HTML sử dụng đến XSL style hoặc thao tác thông qua chương trình sử dụng đến các lớp .NET. Theo cách tiếp này, bạn có thể tạo một dáng dấp cho những chú giải mã nguồn của bạn, nhưng cũng có một phương án thay thế.

2.6.1 Hỗ trợ sưu liệu của Visual Studio .NET

Visual Studio .NET cung cấp cho bạn một mục chọn định dạng tập tin XML. Cũng tập tin Payroll.xml kể trên, bạn chỉ cần dùng lệnh trình đơn **Tools | Build Comment Web Pages**. Lúc này khung đối thoại **Build Comment Web Pages** hiện lên với hai nút radio: Build for entire Solution và Build for selected Projects. Bạn chọn một trong hai mục chọn.



Hình 2-5: Payroll.xml kết sinh bởi Visual Studio .NET

Lúc này, Visual Studio .NET sẽ tạo một thư mục mới trong thư mục dự án dùng giữ số hình ảnh và tập tin HTML được xây dựng dựa trên sưu liệu XML của bạn. Bạn có thể mở tập tin HTML chính và nhìn xem dự án được chú giải. Bạn xem hình 2-5.

Chương 3

Sử dụng Debugger thế nào ?

Có thể nói không ngoa, trong một môi trường triển khai phần mềm, công cụ quan trọng nhất là công cụ gỡ rối chương trình, được gọi là Debugger. Visual Studio.NET Debugger là một công cụ cực mạnh, bạn nên dành thời gian nghiên cứu kỷ cách sử dụng công cụ này một cách nhuần nhuyễn. Do đó, chúng tôi dành trọn một chương khá dài để bạn nắm vững vấn đề.

Cơ bản mà nói, việc gỡ rối khá đơn giản. Việc này đòi hỏi 3 "tài nghệ":

- Làm thế nào đặt một chốt ngừng (breakpoint) và làm thế nào cho chạy đến chốt ngừng.
- Làm thế nào chui vào và vượt lên trên các triệu gọi hàm hành sự.
- Làm thế nào quan sát và thay đổi trị của các biến, vùng mục tin v.v..

Bây giờ, chúng ta đi vào vấn đề.

Visual Studio.NET Debugger là một công cụ cực mạnh cho phép bạn quan sát cách hành xử vào lúc chạy của chương trình và xác định vị trí xảy ra sai lầm. Debugger hiểu rõ những chức năng được xây dựng trong một ngôn ngữ lập trình và những thư viện được gắn liền. Với Debugger, bạn có thể cho ngắt (ngưng) thi hành chương trình để có thể quan sát đoạn mã, định trị và hiệu đính các biến trong chương trình, nhìn xem nội dung các thanh ghi (register), các chỉ thị được tạo ra từ đoạn mã nguồn, cũng như nhìn xem vùng ký ức bị chiếm dụng bởi ứng dụng.

Visual Studio Debugger cung cấp cho bạn những mục chọn trình đơn (menu item), Windows, và các khung đối thoại (dialog) cho phép bạn sử dụng các công cụ của Debugger. Bạn có thể nhận giúp đỡ (help) thông qua nút <F1> cũng như bạn có thể lôi thả (drag and drop) để di chuyển thông tin gỡ rối giữa các cửa sổ.

3.1 Các điều cơ bản

Chương trình là do con người viết ra, nên lỗi lầm xảy ra là lẽ thường tình. Kể cả khi bạn là lập trình viên chuyên nghiệp kinh nghiệm đầy mình, cũng không tránh khỏi sai sót.

Có nhiều loại lỗi (tiếng lóng là bug) trong các đoạn mã (code) chương trình: lỗi cú pháp, lỗi ngữ nghĩa, lỗi lô gic:

- ***Lỗi cú pháp*** (syntax error): Còn gọi là compile-time error, sai lầm vào lúc biên dịch. Đây là loại lỗi dễ thấy nhất, xảy ra khi bạn viết những câu lệnh không đúng qui tắc của ngôn ngữ (phần lớn do khỏ sai, hoặc thiếu sót), và cũng dễ phát hiện bởi trình biên dịch thông qua những thông điệp sai lầm cho bạn biết là có vấn đề. Trong Visual Studio.NET, những thông điệp sai lầm sẽ xuất hiện trên cửa sổ **Output Window**. Càc thông điệp này cho bạn biết sai nằm ở đâu (hàng số mấy, tập tin nào) kèm theo mô tả vấn đề một cách khô khan, lẽ dĩ nhiên bằng tiếng Anh (rất khó hiểu đối với các bạn VN). Tìm ra nguyên nhân sai loại này thường thì dễ sửa một khi bạn đã quen và hiểu lời mô tả. Ngoài ra, trên câu lệnh chỗ nào sai hay thiếu sót thì Debugger cho hiện lên ngay dưới câu lệnh những dấu hiệu (kiểu con rắn ngoằn ngoèo) màu đỏ. Bạn tùy nghi suy đoán ra sai lầm rồi tự mà sửa lấy. Theo thiển ý, bạn nên cố tình viết sai một số sai về cú pháp để xem thông điệp báo lỗi là gì để mà hiểu ý nghĩa của nó. Về sau khi gặp những thông điệp này, thì bạn biết tỏng sai cái gì. Thật tình mà nói, những thông điệp báo sai của Microsoft rất tồi, và thường là tối nghĩa chả hiểu mô tê gì cả.

- ***Lỗi mang tính ngữ nghĩa*** (semantic error): Còn gọi là run-time error, sai lầm vào lúc chạy chương trình. Loại lỗi này là loại rất tế nhị khó phát hiện, chỉ xảy ra khi cú pháp đoạn mã xem ra không sai gì cả, nhưng về mặt ý nghĩa thì không đúng như bạn mong muốn. Vì câu lệnh tuân theo đúng qui tắc của ngôn ngữ, nên trình biên dịch không thể phát hiện loại sai ngữ nghĩa này. Bạn nhớ cho, trình biên dịch chỉ quan tâm đến qui tắc ngôn ngữ, đến cấu trúc đoạn mã, còn ý nghĩa của đoạn mã ra sao thì nó mù tịt. Thông thường loại sai lầm này sẽ làm cho chương trình bạn ngưng ngang xương, có hoặc không kèm theo thông điệp. Ta gọi chương trình của bạn "sụm bà chè" (crash) đối với loại sai lầm khó chịu này.

- ***Lỗi lô gic và lỗi thiết kế*** (logic and design error). Tuy nhiên, không phải tất cả các sai lầm ngữ nghĩa này đều gây ra "gảy đổ" trong chương trình của bạn. Có thể chương trình mang "mầm bệnh tiềm ẩn" này vẫn chạy, nhưng các biến có thể không chứa dữ liệu đúng đắn, hoặc chương trình đi sai đường đưa đến kết xuất sai. Những loại sai lầm này được gọi là lỗi lô gic, hoặc lỗi thiết kế, có nghĩa là chương trình không "sụm bà chè", nhưng lô gic mà chương trình thi hành là sai.

3.1.1 Trắc nghiệm

Cách duy nhất để phát hiện những sai lầm lô gic là cho thử nghiệm (test) chương trình bằng tay hoặc tự động rồi kiểm tra kết xuất xem có đúng như chờ đợi hay không. Trắc nghiệm được xem như là thành phần trong tiến trình triển khai phần mềm. Rất tiếc là khi trắc nghiệm, bạn có thể thấy là kết xuất sai, nhưng bí một điều là bạn không dò ra

manh mối sai ở đoạn nào trong chương trình. Chính lúc này, kỹ thuật gỡ rối (debugging) nhập cuộc.

3.1.2 Gỡ rối chương trình

Gỡ rối thường phải sử dụng đến một bộ gỡ rối (debugger), nhưng không phải lúc nào cũng dùng đến. Debugger là một công cụ cực mạnh cho phép bạn quan sát cách hành xử của chương trình vào lúc chạy và xác định nơi xảy ra sai lầm lô gic. Ngoài ra, bạn cũng có thể sử dụng vài chức năng gỡ rối được xây dựng ngay trong ngôn ngữ được gắn liền với những thư viện.

Nhiều lập trình viên lần đầu tiên khi gỡ rối thường hay cố cô lập một vấn đề bằng cách thêm vào chương trình những hàm triệu gọi các hàm kết xuất, như **MsgBox** chẳng hạn, để "xem tận mắt bắt tận tay" những sai lầm. Đây là cách làm rất hợp lệ, nhưng một khi bạn đã dò tìm đúng chỗ sai và sửa chữa xong, thì bạn buộc phải trở lui gỡ bỏ tất cả các triệu gọi hàm "chữa cháy" kể trên. Công việc dọn dẹp này, cũng như tất cả mọi công việc thuộc loại này, xem ra nhàm chán không hứng thú chi. Nhưng có điều, đôi khi việc thêm những triệu gọi hàm "chữa cháy" MsgBox chẳng hạn, cũng có thể thay đổi cách hành xử của đoạn mã mà bạn đang cố gỡ rối.

Với một debugger, bạn có thể xem xét nội dung các biến chương trình mà khỏi phải chèn những triệu gọi hàm để kết xuất trị của những biến này. Trên đoạn mã, bạn có thể chèn một ***chốt ngừng*** (breakpoint) để có thể cho ngưng thi hành chương trình ngay tại điểm bạn muốn quan sát. Khi chương trình ngừng (ta gọi là chương trình đang ở chế độ "break mode"), bạn có thể quan sát các biến cục bộ (local variable) và các dữ kiện khác sử dụng những tiện nghi như cửa sổ **Watch Window,** khung đối thoại **QuickWatch,** và cửa sổ Memory. Không những bạn có thể quan sát nội dung khi đang ở break mode, bạn còn có thể thay đổi hoặc hiệu đính nội dung nếu bạn muốn. Trong phần lớn trường hợp, bạn sẽ đặt chốt ngừng trong một cửa sổ mã nguồn, nơi mà bạn đang viết đoạn mã và hiệu đính. Nhưng thỉnh thoảng, có thể bạn chọn đặt chốt ngừng trên cửa sổ Disassembly của debugger. Cửa sổ này cho bạn thấy chỉ thị được tạo ra từ mã nguồn. Nói tóm lại, thiết đặt một chốt ngừng sẽ không thêm những triệu gọi hàm vào mã nguồn của bạn, do đó sẽ không thay đổi cách hành xử của chương trình.

3.1.2.1 Cơ bản về gỡ rối: Các chốt ngừng

Một chốt ngừng được xem như là một dấu hiệu báo cho debugger biết tạm ngưng thi hành chương trình ở một điểm nào đó. Lúc này chương trình của bạn ở "break mode" (chế độ ngắt). Khi đang ở chế độ break này, chương trình chỉ tạm ngưng thi hành, chứ

không chấm dứt luôn, và bạn có thể cho tiếp tục chạy lại (continued, hoặc resumed) bất cứ lúc nào bạn muốn.

Khi đang ở break mode, các hoạt động của các biến, hàm và đối tượng trong ký ức đều bị "đóng băng" và

- Lúc này bạn có thể quan sát tình trạng của các biến, biểu thức và hàm. Bạn có quyền thay đổi điều chỉnh trị của một biến chẳng hạn khi chương trình đang ở break mode.

- Bạn có thể di chuyển theo từng bước (step) một trong đoạn mã.

- Ngoài ra, bạn cũng có thể vào ra trên các hàm hoặc di chuyển điểm thi hành, nghĩa là thay đổi câu lệnh sẽ được thi hành kế tiếp khi chương trình tiếp tục chạy (thông qua một chức năng cực mạnh được gọi là **Edit and Continue**).

Chốt ngừng là một phương tiện cực mạnh cho phép bạn ngưng thi hành chương trình bất cứ nơi nào và vào lúc nào bạn muốn. Thay vì nhảy từng bước một xuyên qua đoạn mã, nghĩa là từng câu lệnh một, bạn có thể cho phép chương trình chạy cho tới khi gặp chốt ngừng, rồi từ đó bạn bắt đầu gỡ rối. Như vậy tốc độ gỡ rối sẽ tăng mạnh. Không có chốt ngừng thì hầu như khó lòng gỡ rối một chương trình rộng lớn đồ sộ.

Nhiều ngôn ngữ lập trình có sẵn những câu lệnh hoặc cấu trúc cho phép ngưng thi hành chương trình và đặt chương trình ở chế độ break mode. Thí dụ, Visual Basic có câu lệnh **Stop**. Chốt ngừng khác với những câu lệnh vừa kể trên ở chỗ chốt ngừng không phải là câu lệnh được thêm vào chương trình của bạn. Bạn không có khỏ một câu lệnh chốt ngừng vào cửa sổ mã nguồn. Bạn yêu cầu một chốt ngừng thông qua giao diện Debugger, và Debugger cho đặt theo yêu cầu. *Muốn đặt một chốt ngừng, đơn giản bạn chỉ cần click vào vùng biên màu xám phía tay trái cạnh dòng lệnh nơi mà bạn muốn có chốt ngừng. Bạn* có thể xử lý những chốt ngừng phức tạp thông qua cửa sổ **Breakpoints Window**.

Chốt ngừng có nhiều lợi điểm hơn câu lệnh gỡ rối trong ngôn ngữ như câu lệnh **Stop** trên VB. Chốt ngừng có thể bị gỡ bỏ hoặc thay đổi mà khỏi thay đổi mã nguồn. Vì chốt ngừng không phải là những câu lệnh, nên nó không tạo ra đoạn mã phụ khi bạn xây dựng phiên bản "xuất xưởng" (release version). Nếu bạn sử dụng câu lệnh **Stop** trong chương trình, bạn phải tự mình gỡ bỏ bằng tay các câu lệnh **Stop** này trước khi phát ra phiên bản "xuất xưởng", hoặc dùng đến những câu lệnh điều kiện, như sau:

```
#If DEBUG Then
   Stop
#End If
```

Nếu bạn muốn tạm thời "phế bỏ võ công" của câu lệnh **Stop**, bạn cần biết rõ nó ở đâu trên đoạn mã nguồn rồi biến nó thành chú giải (động tác này ta gọi là comment out hoặc uncomment):

```
' Stop
```

Sẽ không vấn đề gì nếu chỉ có một vài câu lệnh **Stop**. Tuy nhiên, đối với một chương trình rộng lớn bạn sẽ có vô số câu lệnh **Stop**, và việc truy tìm cũng như comment out từng câu lệnh một này là việc làm mất thời gian. Với chốt ngừng, bạn có thể chọn cho hiệu lực hoặc vô hiệu hoá bất cứ chốt ngừng nào hoặc tất cả từ cửa sổ **Breakpoints Window.**

Cuối cùng, breakpoints có một điểm lợi lớn so với câu lệnh **Stop** là mức độ uyển chuyển. Câu lệnh **Stop** cho ngưng chương trình ngay tại dòng lệnh trên đoạn mã nguồn nơi **Stop** được đặt. Bạn cũng có thể cho đặt chốt ngừng làm công việc tương tự. Tuy nhiên, bạn cũng có thể đặt một chốt ngừng trên một hàm hoặc trên một vị chỉ ký ức (memory address), mà với **Stop** thì không thể được. Ngoài ra, Debugger của Visual Studio.NET còn cung cấp một loại chốt ngừng được gọi là *data breakpoint.* Một *data breakpoint* sẽ được đặt trên một biến toàn cục hoặc biến cục bộ, thay vì một vị trí trên đoạn mã. Cho đặt để một data breakpoint sẽ làm cho chương trình ngưng thi hành khi trị của biến thay đổi.

Để tăng độ uyển chuyển lên cao hơn nữa, Debugger của Visual Studio cho phép bạn đặt để hai thuộc tính **Hit Count** và **Condition** làm thay đổi cách hành xử của chốt ngừng:

1. Thuộc tính ***Hit Count*** cho phép bạn xác định phải bao nhiêu lần chốt ngừng "bị tộng" (hit) trước khi Debugger tạm ngưng thi hành chương trình. Theo mặc nhiên, mỗi lần chốt ngừng bị tộng thì Debugger cho ngừng thi hành chương trình. Bạn có thể cho **Hit count** về một con số nào đó (cứ 10 lần hoặc cứ 512 lần chẳng hạn) cho Debugger biết số lần mà chốt ngừng bị đụng thì mới ngưng thi hành chương trình một lần**. Hit count** có thể hữu ích vì một vài bug sẽ không xuất hiện lần đầu tiên khi chương trình bạn thi hành một vòng lặp, triệu gọi một hàm hoặc truy xuất một biến. Thỉnh thoảng, có thể bug sẽ không xuất hiện cho tới khi chạy vòng 100 lần hoặc 1000 lần. Muốn gỡ rối vấn đề này, bạn cho đặt chốt ngừng với một hit count bằng 100 hoặc bằng 1000.

2. Thuộc tính ***Condition*** là một biểu thức xác định liệu xem breakpoint bị đụng hoặc bị nhảy bỏ qua hay không. Khi Debugger đạt đến chốt ngừng, nó sẽ định trị thuộc tính **Condition**. Breakpoint chỉ sẽ bị đụng khi điều kiện Condition là thoả (`true`). Do đó, bạn có thể sử dụng một điều kiện nào đó để cho phép đặt một chốt ngừng ở một nơi nào đó khi điều kiện **Condition** bằng `true`. Giả sử, bạn đang gỡ

rối một chương trình ngân hàng theo đấy bảng cân số của một tài khoản không bao giờ được âm. Bạn có thể đặt chốt ngừng ở chỗ nào đó trong đoạn mã và cho gắn điều kiện như sau: **balance < 0** ở mỗi nơi. Khi bạn cho chạy chương trình, việc thi hành chỉ sẽ tạm ngưng ở những nơi nào mà cân số tài khoản trở thành âm. Bạn có thể quan sát các biến và tình trạng chương trình ở chốt ngừng, rồi tiếp tục cho tới chốt ngừng thứ hai, v.v

Tip Một kỹ thuật đặc biệt hữu ích là cho đặt để chốt ngừng trên cửa sổ Call Stack**,** cho phép đặt để một chốt ngừng trên một triệu gọi hàm riêng biệt nào đó. Điều này rất hữu ích khi bạn phải gỡ rối một hàm đệ quy (recursive function, một hàm tự triệu gọi mình). Nếu cho ngưng thi hành chương trình sau một số lần triệu gọi hàm, bạn có thể dùng cửa sổ Call Stack cho đặt để một chốt ngừng trên một triệu gọi hàm trước đó chưa được trả về. Debugger sẽ gặp phải chốt ngừng này và ngưng thi hành chương trình trên đường thoát khỏi những triệu gọi hiện hành

3.1.3 Các công cụ gỡ rối dùng quan sát chương trình

Mục tiêu chính của bất cứ Debugger nào là cho hiển thị thông tin liên quan đến tình trạng của chương trình đang được gỡ rối và trong vài trường hợp, thay đổi tình trạng này. Visual Studio.NET Debugger cung cấp cho bạn vô số công cụ cho phép bạn quan sát và thay đổi tình trạng chương trình. Phần lớn các chức năng của các công cụ này chỉ hoạt động ở chế độ break mode.

3.1.3.1 DataTips

Một trong những công cụ lặng lẽ nhất là **DataTips**, một loại ô thông tin vọt lên cho phép bạn nhìn thấy trị của một biến trong phạm vi (scope) hiện hành khi bạn đưa con nháy lên biến trên đoạn mã nguồn, và khi Debugger đang ở chế độ break mode. Nếu biểu thức nằm ngoài phạm vi, hoặc khi biểu thức không hợp lệ (thí dụ chia cho zero), hoặc những biểu thức đòi hỏi định trị hàm, thì **DataTips** sẽ không hiện lên.

3.1.3.2 Các cửa sổ và khung đối thoại của Debugger

Để có thể quan sát một cách chi tiết chương trình của bạn, Visual Studio.NET Debugger cung cấp cho bạn vô số cửa sổ và khung đối thoại. Sau đây chúng tôi liệt kê các cửa sổ mà bạn sẽ làm quen trong chầu gỡ rối:

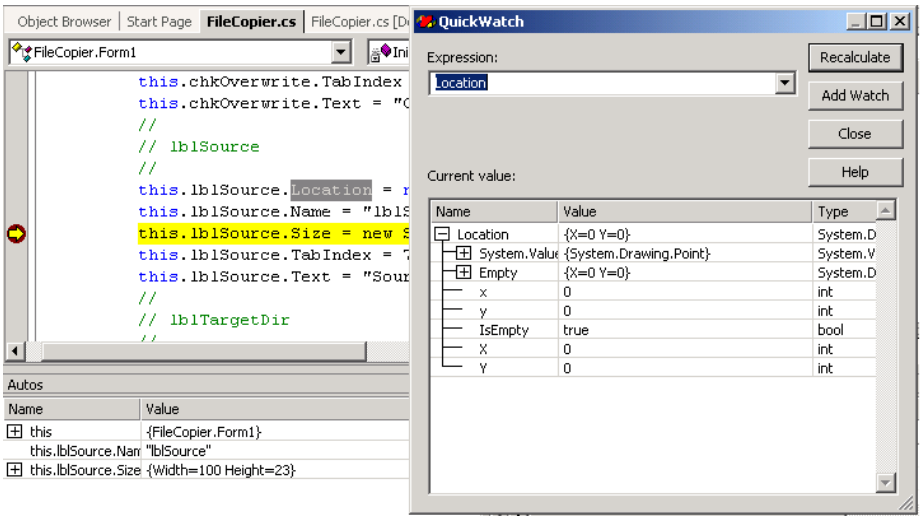
bạn thử dùng…	**Dùng để nhìn xem…**
Autos Window	Cho hiển thị các biến trên câu lệnh hiện hành và câu lệnh đi trước, rồi sau đó có thể chọn và hiệu đính bất cứ biến nào.
Locals Window	Biến cục bộ. Bạn có thể chọn và hiệu đính bất cứ biến cục bộ nào.
This Window	Đối tượng được gắn liền với hàm hành sự hiện hành. Cho phép bạn quan sát các biến thành viên của đối tượng được gắn liền với hàm hành sự hiện hành.
QuickWatch Dialog Box	Biến, nội dung thanh ghi, và bất cứ biểu thức hợp lệ nào nhận diện được bởi Debugger. Cửa sổ này cho phép bạn nhìn thấy nhanh trị các biến hoặc biểu thức, cho định trị chúng, rồi nếu muốn có thể thay đổi trị của chúng để sửa chữa sai lầm hoặc muốn xem diễn tiến sự việc ra sao.
Watch Window	Biến, nội dung thanh ghi, và bất cứ biểu thức hợp lệ nào nhận diện được bởi Debugger.
Registers Window	Nội dung thanh ghi
Memory Window	Nội dung ký ức
Call Stack Window	Tên các hàm trên call stack, các kiểu dữ liệu thông số, trị các thông số.
Disassembly Window	Mã assembly được kết sinh bởi trình biên dịch.
Threads Window	Thông tin liên quan đến các mạch trình (dòng lệnh được thi hành tuần tự) được tạo ra bởi chương trình.
Modules Window	Module (DLL/EXE) mà chương trình dùng đến.

Running Documents Window	Một danh sách các tài liệu, mã kịch bản (script code), được nạp vào trong qui trình hiện hành (current process).

3.1.3.3 Sử dụng cửa sổ QuickWatch

Khung đối thoại **QuickWatch** (Hình 3-1) cho phép bạn nhìn thấy nhanh trị các biến hoặc biểu thức, cho định trị chúng, rồi nếu muốn có thể thay đổi trị của chúng để sửa chữa sai lầm hoặc muốn xem diễn tiến sự việc ra sao. Muốn sử dụng **QuickWatch,** bạn cho hiện lên khung đối thoại này bằng cách cho ngời sáng một biến bạn nhắm tới, rồi ấn **<Ctrl+Alt+Q>** hoặc chọn mục trình đơn **Debug | QuickWatch,** hoặc bạn right-click ngay trên biến để cho hiện lên trình đơn shortcut rồi chọn mục **QuickWatch**.

QuickWatch là một khung đối thoại kiểu modal nên bạn phải đóng lại khung đối thoại trước khi bạn có thể tiếp tục gỡ rối. Bạn tự hỏi vì sao **QuickWatch** là hữu ích. Tại sao không thêm biến và biểu thức vào cửa sổ **Watch Window**? Thật ra, bạn cũng có thể làm được như thế, nhưng giả sử bạn đơn giản muốn tính lẹ trên một vài biến. Bạn không muốn dồn cục lên cửa sổ **Watch Window** với những tính toán như thế. Do đó **QuickWatch** nhập cuộc.



Hình 3-1: Cửa sổ QuickWatch

Một chức năng "mũm mĩm" khác của khung đối thoại **Watch dialog box** là nó có thể thay đổi kích thước được. Nếu bạn muốn nhìn những thành viên của một đối tượng đồ sộ,

thì dễ dàng bung (expand, bằng cách click lên ô vuông có dấu thập) và nhìn vào cây **QuickWatch** hơn là vào các cửa sổ **Watch, Locals,** hoặc cửa sổ **Autos**.

Tip Trên Visual Studio, bạn có thể nhìn nhanh trị của biến bằng cách đưa con nháy (cursor) lên biến. Một khung nhỏ mang tên **DataTip** sẽ xuất hiện cho hiển thị trị của biến. Nhưng với **Data Tip**, bạn chỉ có thể nhìn mà thôi, không thể hiệu đính trị của biến.

Sử dụng QuickWatch thế nào?

Định trị một biểu thức trên khung đối thoại QuickWatch

Bạn có thể sử dụng khung đối thoại QuickWatch để định trị (evaluate) nhanh một biến, một biểu thức, hoặc một thanh ghi bằng cách:

1. Trên khung đối thoại **QuickWatch**, bạn khỏ hoặc dán (paste) biến, thanh ghi, hoặc biểu thức vào ô văn bản **Expression**.

2. **Tip** (1) Nếu tên thanh ghi xem ra khớp với một tên tượng trưng, bạn nên dùng ký hiệu @ như là tiền tố (prefix) đối với tên thanh ghi. (2) bạn có thể thay đổi dạng thức hiển thị (display format) bằng cách khỏ một format specifier theo sau biểu thức. (3) bạn có thể dùng context operator để khai báo phạm trù chính xác của một biến hoặc biểu thức. Điều này có thể hữu ích nếu bạn muốn quan sát một biến nằm ngoài phạm vi hiện hành. Tất cả 3 tip trên chỉ dùng với mã nguyên sinh (native code).

3. Bạn click nút **Recalculate**.

4. Trị được tính toán sẽ hiện lên trên ô label **Current value**.

5. Nếu bạn khỏ vào tên của một bản dãy hoặc một biến đối tượng (object variable) vào ô văn bản **Expression,** thì một tree control sẽ xuất hiện cạnh bên tên của ô **Current value**. Bạn click dấu (+) hoặc dấu (–) trên cột **Name** để cho bung ra hoặc teo lại biến.

Để bạn khỏi nhọc công khỏ lại một biểu thức trước đó đã dùng, bạn có thể chọn biểu thức này từ một ô liệt kê kéo xuống (drop-down list) được gắn liền với ô Expression. Bạn click mũi tên phía tay phải ô **Expression,** chọn một biểu thức trên bảng liệt kê được rủ xuống rồi click nút **Recalculate**.

Hiệu đính một trị trên QuickWatch

Bạn có thể dùng **QuickWatch** để hiệu đính trị của một biến hoặc thanh ghi bằng cách:

1. Trên khung đối thoại **QuickWatch**, bạn khỏ vào biến, tên thanh ghi, vào ô văn bản **Expression**.

2. Bạn click nút **Recalculate**.

3. Nếu bạn khỏ vào tên của một bản dãy hoặc một biến đối tượng (object variable) vào ô văn bản **Expression,** thì một tree control sẽ xuất hiện cạnh bên tên của ô **Current value**. Bạn click dấu (+) hoặc dấu (–) trên cột **Name** để cho bung ra hoặc teo lại biến để tìm ra phần tử mà bạn muốn hiệu đính trị.

4. Bạn double-click cột **Value** của ô **Current value** để chọn ra trị.

5. Bạn dùng con chuột hoặc bàn phím để hiệu đính trị.

6. Bạn ấn <ENTER>.

Thêm một biểu thức QuickWatch lên cửa sổ Watch

Muốn thêm một biến hoặc biểu thức lên cửa sổ Watch, bạn theo các bước sau đây:

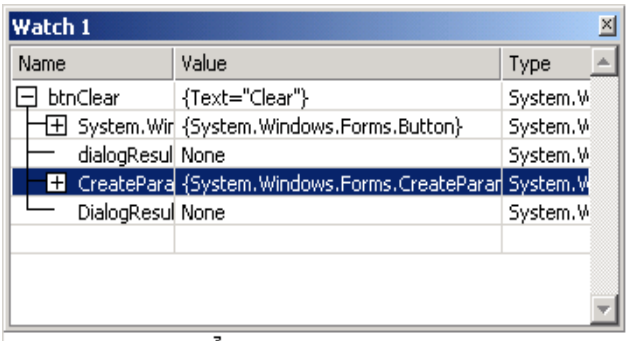
1. Trên cửa sổ khung đối thoại **QuickWatch,** bạn khỏ vào biến hoặc biểu thức lên ô văn bản **Expression.**

2. Bạn click nút **Add Watch**. Như vậy biến hoặc biểu thức sẽ được thêm cửa sổ **Watch1.**

3.1.3.4 Sử dụng cửa sổ Watch Window

Lệnh này cũng tương tự như lệnh **QuickWatch**, theo đấy bạn được phép chỉ định những biến hoặc biểu thức nào cần được thêm vào cửa sổ Watch Window. Thông thường cửa sổ Watch nằm ở cuối màn hình, nhưng bạn có thể di chuyển neo qua phải hoặc qua trái màn hình cũng được. Cửa sổ **Watch** có 4 trang: Watch1, Watch2, Watch3, và Watch 4. Mỗi trang dùng liệt kê một danh sách những biến dựa theo chức năng. Bạn cũng có thể thay đổi trị của các biến trên cửa sổ **Watch**. Giống như QuickWatch, cửa sổ **Watch** cũng cho hiển thị những dấu (+) và (-) trong cột **Name** khi ta quan sát một bản dãy, một đối tượng hoặc một struct. Click lên các dấu này, bạn có thể bung hoặc cho teo lại cái nhìn đối với biến.

Muốn cho cửa sổ **Watch Window** hiện lên, Debugger phải ở tư thế chạy hoặc ở chế độ break mode, rồi bạn chọn mục **Debug | Windows | Watch**, rồi click lên một trong **Watch1**, **Watch2**, **Watch3**, hoặc **Watch4**. (Xem hình 3-2)



Hình 3-2 : Cửa sổ Watch

Bạn có thể dùng cửa sổ **Watch Window** để định trị các biến và biểu thức rồi giữ lấy kết quả. Bạn cũng có thể dùng cửa sổ này để hiệu đính trị của một biến hoặc của thanh ghi. Mỗi trang Watch1…Watch4, có 3 cột:

- **Name:** trên cột này bạn có thể khỏ vào bất cứ biểu thức nào hợp lệ mà Debugger chấp nhận.

- **Value**: Debugger sẽ định trị biểu thức được hiển thị trên cột **Name** rồi ghi kết quả lên trên cột Value này. Nếu biểu thức là một biến hoặc tên thanh ghi, bạn có thể hiệu đính trị trên cột này để thay đổi nội dung của biến hoặc thanh ghi. Bạn không thể hiệu đính trị của những biến **const. Bạn** chỉ có thể hiệu đính và hiển thị trị thanh ghi đối với đoạn mã nguyên sinh mà thôi. Bạn có thể thay đổi dạng thức số (numeric format) của trị thành hexa trong khung đối thoại do việc chọn mục **Options | Debugging | General,** hoặc bằng cách right-click cửa sổ **Watch Window** rồi chọn mục **Hexadecimal Display** trên trình đơn shortcut.

- **Type:** Cột này cho biết kiểu dữ liệu của biến hoặc biểu thức.

Tip Bạn có thể lôi tên các biến và biểu thức từ đoạn mã hoặc từ cửa sổ **Locals Window**, hoặc từ **Auto Window** lên cửa sổ **Watch Window**. Như vậy lợi điểm của Watch Window là bạn được phép quan sát các đối tượng từ nhiều nguồn cửa sổ khác nhau trong cùng một lúc.

Định trị một biểu thức trên cửa sổ Watch Window

Bạn có thể dùng cửa sổ **Watch Window** để định trị các biến, thanh ghi hoặc biểu thức theo thủ tục sau đây, với điều kiện là Debugger phải ở chế độ break mode:

1. Trên cửa sổ **Watch Window**, bạn double-click một hàng rỗng trên cột **Name**.

2. Bạn khỏ vào hoặc dán biến, thanh ghi hoặc biểu thức lên trên hàng đã được tuyển, rồi ấn Enter..

3. Kết quả xuất hiện trên cột **Value**. Nếu bạn khỏ vào tên một bản dãy hoặc một biến đối tượng, thì một tree control hiện lên cạnh tên trên cột **Name**. Bạn click dấu (+) hoặc dấu (–) trên cột **Name** để cho bung ra hoặc teo lại biến.

4. Biểu thức ở yên tại chỗ trên **Watch Window** cho tới khi nào bạn gỡ bỏ nó đi..

Hiệu đính một trị trên cửa sổ Watch Window

Bạn có thể sử dụng cửa sổ **Watch Window** để hiệu đính trị của một biến hoặc của thanh ghi. Bạn theo thủ tục sau đây:

1. Debugger phải ở chế độ break mode.

2. Nếu bạn khỏ tên bản dãy hoặc tên một biến đối tượng lên ô văn bản **Expression,** thì một tree control xuất hiện cạnh tên trên ô văn bản **Current value. Bạn** click dấu (+) hoặc (–) trên cột N**ame** để cho bung ra hoặc teo lại biến để tìm ra phần tử mà bạn muốn hiệu đính trị của nó.

3. Bạn double-click cột **Value**, trên hàng tương ứng với biến hoặc với biểu thức mà bạn muốn hiệu đính trị.

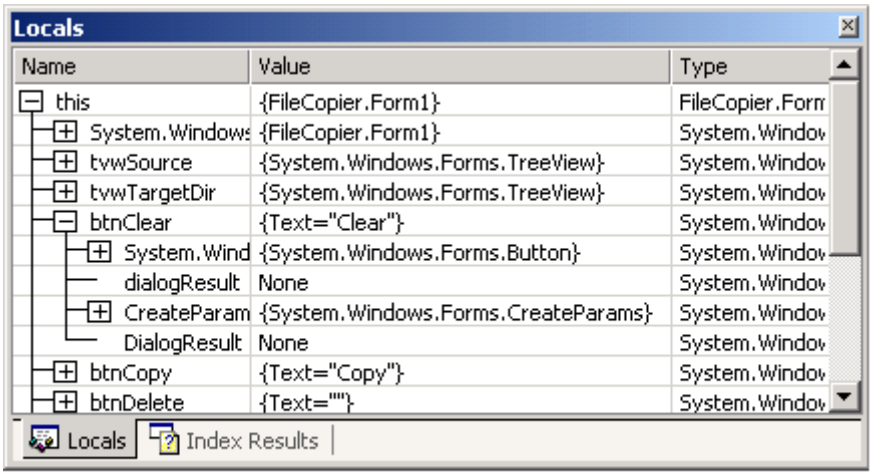
4. Bạn dùng con chuột hoặc bàn phím để hiệu đính trị, rồi ấn **<ENTER>**.

3.1.3.5 Sử dụng cửa sổ Locals Window

Cửa sổ **Locals Window** (hình 3-3) cho hiển thị những biến cục bộ (local variable) trong phạm vi hiện hành. Muốn cho hiển thị cửa sổ Locals Window***:*** Debugger phải đang chạy hoặc ở chế độ break mode. Bạn chọn mục trình đơn **Debug** | **Windows | Locals**, hoặc bạn ấn tổ hợp phím <Ctrl + Alt + V, L>.

Bạn thấy cửa sổ **Locals Window** cũng giống như cửa sổ **Watch Window** gồm 3 cột **Name, Value** và **Type**. Phạm vi mặc nhiên của cửa sổ **Locals Window** là hàm chứa vị trí thi hành hiện hành. Bạn cũng có thể chọn một phạm vi thay thế để hiển thị cửa sổ **Locals Window,** như theo thủ tục sau đây:

- (1) bạn dùng thanh công cụ **Debug Location** (nếu thanh công cụ này chưa xuất hiện bạn chọn mục trình đơn **View | Toolbars | Debug Location**) để chọn ra function, thread, hoặc program bạn mong muốn làm việc, hoặc (2) bạn double-click trên một mục tin (item) trên cửa sổ Call Stack



Hình 3-3 : Cửa sổ Locals Window

Muốn nhìn xem hoặc thay đổi thông tin trên cửa sổ **Locals window**, Debugger phải ở chế độ break mode. Nếu bạn chọn lệnh **Continue**, thì một vài thông tin có thể xuất hiện trên cửa sổ **Locals window** trong khi chương trình đang thi hành, nhưng nó sẽ không hiện hành cho tới khi lần tới chương trình bị ngắt. (Nói cách khác, nó đụng một chốt ngừng hoặc bạn chọn mục **Break All** từ trình đơn **Debug)**

Muốn thay đổi trị của một biến trên cửa sổ Locals Window

1. Debugger phải ở chế độ break mode. Trên cửa sổ **Locals** window, bạn chọn ra trị bạn muốn hiệu đính bằng cách double-click lên trị hoặc bằng cách dùng phím **<TAB>**.

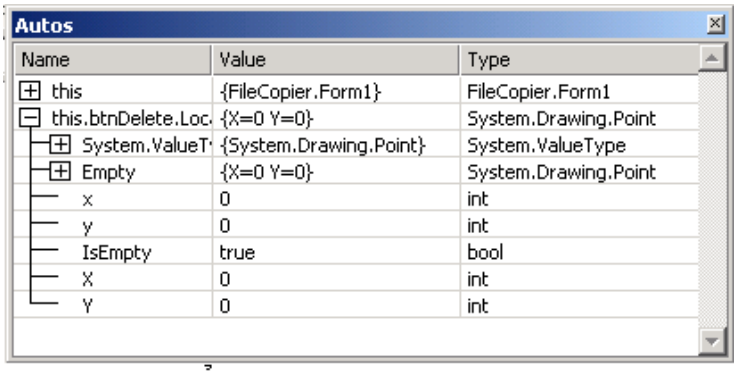
2. Bạn khỏ vào trị mới của biến, rồi ấn **<ENTER>**.

3.1.3.6 Sử dụng cửa sổ Autos Window

Cửa sổ **Autos Window** dùng hiển thị những biến được dùng trong câu lệnh hiện hành và câu lệnh đi trước, được hiển thị trên một tree control. Câu lệnh hiện hành là câu lệnh ở ngay vị trí thi hành hiện hành (nghĩa là câu lệnh sẽ được thi hành kế tiếp nếu việc thi hành được tiếp tục). Debugger sẽ tự động nhận diện những biến này giùm bạn, do đó có tên Auto (tắt chữ Automatic). Các biến struct hoặc biến bản dãy sẽ có một tree control (ô control kiểu nhánh cây) cho phép bạn hiển thị hoặc che dấu các phần tử.

Muốn hiển thị cửa sổ Autos Window

- Debugger phải đang chạy hoặc ở chế độ break mode. Bạn có hai cách để hiển thị cửa sổ: (1) bạn chọn mục trình đơn **Debug | Windows | Autos**. (2) bạn ấn tổ hợp phím <Ctrl + Alt + V, A>. Hình 3-4 hiện lên.



Hình 3-4 : Cửa sổ Autos Window

Cửa sổ **Autos Window** chứa thông tin sau đây:

- **Name: C**ột này chứa tên tất cả các biến trên câu lệnh hiện hành và câu lệnh đi trước. Các biến struct hoặc bản dãy sẽ hiện lên dưới dạng một tree control.

- **Value**: Cột này hiển thị trị của mỗi biến. Theo mặc nhiên, biến số nguyên được hiển thị theo dạng thức thập phân. Bạn có thể thay đổi dạng thức số sang dạng hexa bằng cách chọn mục **Options | Debugging | General,** hoặc bằng cách right-click cửa sổ **Autos Window** rồi chọn mục **Hexadecimal Display** trên trình đơn shortcut.

- **Type:** Cột này cho biết kiểu dữ liệu của biến được liệt kê trên cột **Name**.

Khi gỡ rối một ứng dụng nguyên sinh, và khi nhảy vượt qua (step over) một triệu gọi hàm (F10), cửa sổ **Autos Window** sẽ hiển thị trị trả về đối với hàm này và bất cứ hàm nào có thể được triệu gọi bởi hàm này.

Bạn có thể chọn ra bất cứ biến nào rồi hiệu đính. Tri sẽ được hiển thị theo màu đỏ trên Autos window. Bất cứ thay đổi nào đều có hiệu lực ngay lập tức.

Muốn thay đổi trị của một biến trên cửa sổ Autos Window

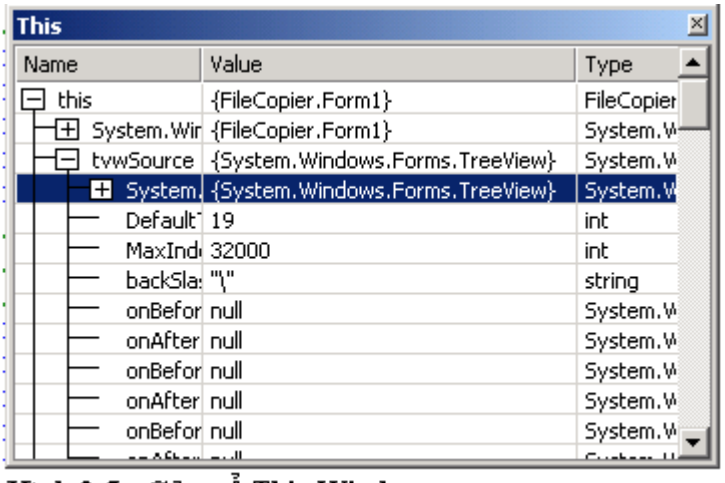
1. Debugger phải ở chế độ break mode. Cho hiển thị cửa sổ **Autos Window**, nếu thấy cần thiết.

2. Trên cột **Value**, bạn double-click trị bạn muốn thay đổi hoặc click để chọn hàng, rồi ấn phím <TAB>.

3. Khỏ vào trị mới của biến được chọn, rồi ấn **<ENTER>**.

3.1.3.7 Sử dụng cửa sổ This Window

Cửa sổ **This Window** (hình 3-5) cho phép bạn quan sát các biến thành viên của đối tượng được gắn liền với hàm hành sự hiện hành. Muốn hiển thị cửa sổ **This Window**: Debugger phải đang chạy hoặc ở chế độ break mode. Bạn có hai cách để hiển thị cửa sổ: (1) bạn chọn mục trình đơn **Debug | Windows | This,** hoặc (2) bạn ấn tổ hợp phím <Ctrl + Alt + V, T>. Hình 3-5 hiện lên.



Hình 3-5 : Cửa sổ This Window

Cửa sổ **This Window** cho thấy 3 cột thông tin:

- **Name: C**ột này cho thấy đối tượng mà con trỏ **this** chỉ về. Biến đối tượng sẽ hiện lên dưới dạng một tree control cho phép bạn bung ra để xem các thành viên.

- **Value**: Cột này hiển thị trị của mỗi biến. Theo mặc nhiên, biến số nguyên được hiển thị theo dạng thức thập phân. Bạn có thể thay đổi dạng thức số sang dạng hexa bằng cách chọn mục **Options | Debugging | General,** hoặc bằng cách right-click cửa sổ **This Window** rồi chọn mục **Hexadecimal Display** trên trình đơn shortcut**.**

- **Type**: Nhận diện kiểu dữ liệu của mỗi biến được liệt kê trên cột **Name**.

Muốn thay đổi trị của một biến trên cửa sổ This Window

1. Debugger phải ở chế độ break mode. Cho hiển thị cửa sổ **This Window**, nếu thấy cần thiết.

2. Trên cột **Value**, bạn double-click trị bạn muốn thay đổi hoặc click để chọn hàng, rồi ấn phím <TAB>.

3. Khỏ vào trị mới, rồi ấn <ENTER>.

3.1.3.8 Sử dụng cửa sổ Registers Window

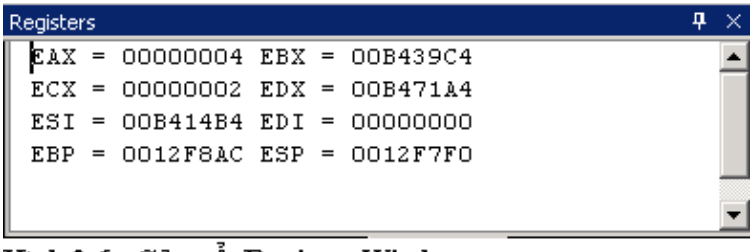
Nếu bạn không rành ngôn ngữ hợp ngữ (assembly language) , thì có thể nhảy bỏ mục này.

Các thanh ghi (register) là những vị chỉ ký ức rất đặc biệt nằm riêng trong lòng bộ xử lý **(**CPU), dùng trữ dữ kiện mà CPU đang làm việc với. Khi biên dịch hoặc diễn dịch các đoạn mã nguồn, ta thường kết sinh (generate) những chỉ thị cho di chuyển dữ kiện vào ra thanh ghi khi cần thiết. Truy xuất dữ liệu thông qua thanh ghi rất nhanh so với truy xuất dữ liệu trong ký ức trung ương. Do đó, đoạn mã nào cho phép processor giữ lại dữ liệu trên thanh ghi và truy xuất dữ liệu này liên tục thì có chiều hướng chạy nhanh hơn là với đoạn mã đòi hỏi phải nạp vô ra thanh ghi liên tục. Để cho trình biên dịch giữ lại dữ liệu trên thanh ghi dễ dàng hơn và thực hiện những tối ưu hoá khác, bạn nên tránh sử dụng những biến toàn cục (global variable) và nên sử dụng càng nhiều càng tốt những biến cục bộ (local variable).

Thanh ghi được chia làm hai loại: loại dành mục đích chung (general purpose) và loại dành cho mục đích đặc biệt (special purpose). General-purpose registers dùng trữ dữ liệu cho những dạng trả về tổng quát, như cộng hai số lại với nhau, hoặc dùng qui chiếu một phần tử bản dãy. Còn special-purpose registers được dùng vào mục đích đặc thù mang ý nghĩa đặc biệt. Thí dụ: một stack-pointer register (thanh ghi con trỏ stack), mà bộ processor dùng theo dõi các triệu gọi chương trình trữ trên một stack. Lập trình viên như bạn sẽ không thao tác trực tiếp lên stack pointer. Tuy nhiên, nó rất cần thiết cho hoạt động đúng đắn của chương trình, vì rằng không có stack pointer, bộ processor sẽ không biết đâu mà về sau khi triệu gọi hàm đã làm xong việc.

Phần lớn những general-purpose registers chỉ trữ một phần tử dữ liệu duy nhất (thí dụ, một số nguyên đơn, một con số floating-point, hoặc một phần tử bản dãy). Trên những bộ processor tiên tiến sau này có những register rộng lớn hơn, mà ta gọi là **vector registers**, có thể trữ một bản dãy nhỏ dữ liệu. Do đó, vector register cho phép thực hiện những tác vụ rất nhanh đối với những bản dãy. Vector register được sử dụng trước tiên trên các siêu máy tính mắc tiền, hiệu năng cao, nhưng bây giờ đã có sẵn trên các máy vi tính có thể đem sử dụng trong các công tác đồ hoạ.

Một processor thường có hai bộ general-purpose registers, một bộ dành tối ưu hoá các tác vụ floating-point, còn bộ kia dành cho những tác vụ số nguyên. Do đó, bạn



Hình 3-6 : Cửa sổ Register Window

khỏi thắc mắc khi gọi chúng là floa- ting-point reg- ister và inte- ger register.

Đoạn mã chạy trên CLR, managed code sẽ được biên dịch về mã "nguyên sinh" và truy xuất những thanh ghi vật lý của bộ vi xử lý **(**microprocessor). Cửa sổ **Registers** sẽ cho hiển thị những thanh ghi vật lý này đối với CLR hoặc đối với mã nguyên sinh. Muốn cho hiện lên cửa sổ này (hình 3-6), bạn hoặc ấn tổ hợp phím **<Ctrl+Alt+G>** hoặc chọn mục trình đơn **Debug |Window | Register.**

Khi bạn nhìn vào cửa sổ **Registers**, bạn sẽ thấy những mục vào (entries) chẳng hạn thí dụ sau đây:

```
EBX = 00B439C4
```

Ký hiệu ở vế trái dấu = là tên thanh ghi (EBX, trong trường hợp này). Con số nằm ở vế tay phải dấu = cho biết nội dung của thanh ghi (ghi theo hexa).

Cửa sổ **Registers** cho phép bạn làm nhiều việc hơn là chỉ đọc nội dung của thanh ghi. Khi bạn đang ở chế độ break mode trên native code, bạn có thể click trên nội dung của register và hiệu đính trị. Điều này bạn không thể làm một cách hồ đồ vô ý thức. Trừ phi bạn hiểu rõ register bạn đang hiệu đính, cũng như dữ liệu được trữ, kết quả của một hiệu đính vô ý thức sẽ dẫn đến hậu quả là chương trình của bạn có thể "sụm bà chè", và nhiều hậu quả tai hại khó lường. Nếu bạn thực sự muốn hiểu register nên tìm xem sách nói về ngôn ngữ hợp ngữ (assembly language).

Register chia thành nhóm

Để giảm thiểu hiện tượng dồn cục, cửa sổ **Registers** tổ chức các thanh ghi thành nhóm. Nếu bạn right-click lên cửa sổ **Registers**, bạn sẽ thấy một shortcut menu chứa một danh sách các nhóm (CPU, CPU Segment và Flags, v.v..) mà bạn có thể cho hiển thị hoặc cho cất giấu tùy ý. Bạn có thể thay đổi trị của một thanh ghi hoặc đặt để một flag trong khi gỡ rối chương trình. Sau đây là ý nghĩa của những nhóm này:

- **CPU** Nhóm này chứa những thanh ghi loại basic general-purpose integer registers.

- **CPU Segments** Nhóm này chứa Segment Registers (thanh ghi phân đoạn), là những thanh ghi đặc thù, đối với những ứng dụng trên máy PC sử dụng ký ức chia thành phân đoạn (segment); có thể bạn sẽ không sử dụng nhóm này.

- **Floating Point** Nhóm này gồm những thanh ghi loại general-purpose floating-point registers.

- **MMX, SSE, SSE2, 3Dnow!** Nhóm này gồm một số thanh ghi cực mạnh được thêm vào những processor mới nhất sau này (MMX, Pentium 3, Pentium 4, và Athlon) dành cho đồ hoạ cũng như những công tác đòi hỏi tốc độ cao.

- **Register Stack** Một nhóm gồm 8 thanh ghi floating-point được tổ chức như là một stack, dành cho những tác vụ tính toán floating-point nhanh.

- **Application** Thanh ghi ứng dụng (application register) hỗ trợ những hàm khác nhau trên processor 64-bit.

- **Branch** Nhóm thanh ghi này trữ những thông tin rẽ nhánh 64-bit.

- **Flags** Flag (cờ hiệu) là những thanh ghi one-bit chứa thông tin mà các processor và hệ điều hành dùng đến. Bản thân bạn là lập trình viên sẽ không thường xuyên dùng đến.

- **Effective Address** Nhóm thanh ghi này được dùng bởi những chỉ thị sử dụng đến Effective Address mode (chế độ vị chỉ thực thụ). Nếu bạn không hiểu thì mở một quyển sách về lập trình ngôn ngữ assembly.

Bạn chỉ sử dụng cửa sổ này khi bạn thật rành ngôn ngữ assembly. Bằng không thì nên "kính nhi viễn chi".

Muốn hiệu đính trị của một thanh ghi

Bạn theo thủ tục sau đây:

1. Trên cửa sổ **Registers Window**, bạn dùng phím **<TAB>** hoặc con chuột di chuyển con trỏ (insertion point) về trị bạn muốn hiệu đính. Khi bạn bắt đầu khỏ thì con nháy phải nằm ngay trước trị bạn một viết đè lên.

2. Bạn khỏ trị mới vào, rồi ấn **<ENTER>**.

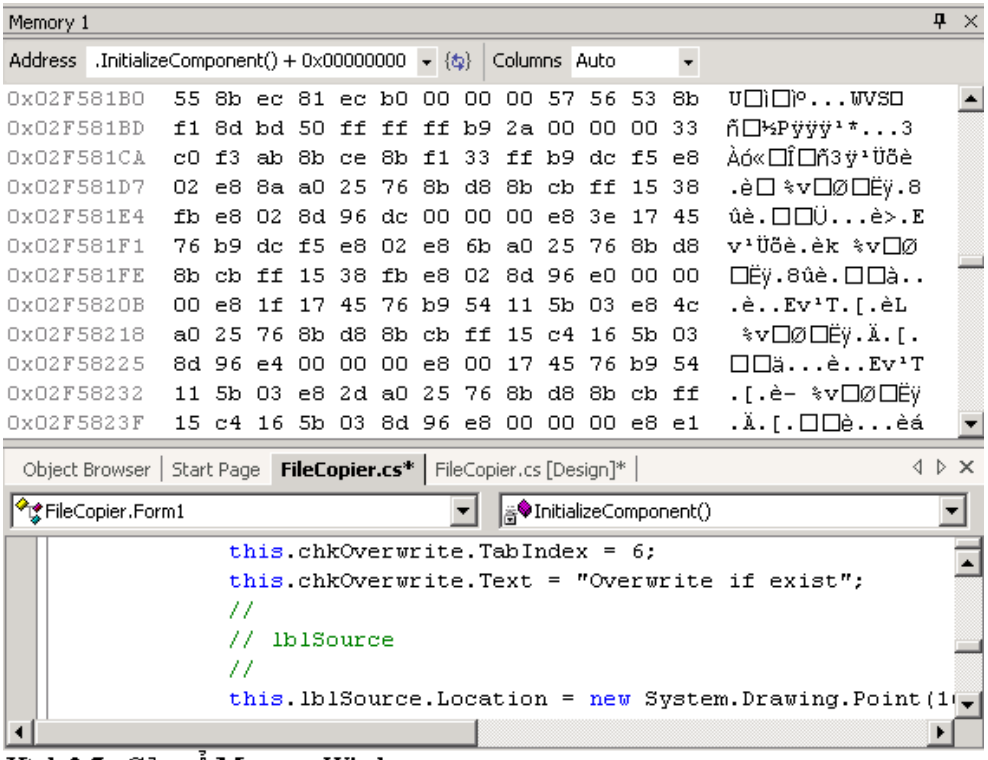
3.1.3.9 Sử dụng cửa sổ Memory

Bạn có thể cho hiện lên cửa sổ Memory (hình 3.7) bằng cách ấn tổ hợp phím **<Ctrl+Alt+M, 1 hoặc 2 hoặc 3>** hoặc chọn mục trình đơn **Debug | Windows | Memory | Memory 1** hoặc Memory 2 hoặc Memory3.

Cửa sổ **Memory** cho phép bạn có một cái nhìn (view) vào vùng ký ức mà chương trình bạn đang chiếm dụng. Các cửa sổ **Watch Window, QuickWatch, Autos Window,**

Locals Window, và **This Window** cung cấp một cách nhìn vào nội dung của các biến được trữ trên ký ức vào một vị trí nào đó, còn cửa sổ Memory thì lại cho bạn một hình ảnh ký ức qui mô lớn hơn. Cái nhìn này có điểm tiện lợi là có thể quan sát một khối dữ liệu lớn (vùng đệm, hoặc một chuỗi chữ dài chẳng hạn) mà ta không thể hiển thị tốt trên các cửa sổ khác. Tuy nhiên, cửa sổ Memory không chỉ giới hạn vào việc hiển thị dữ liệu. Theo định nghĩa, **Memory Window** cho hiển thị mọi thứ trong ký ức cho dù là data, code, hoặc "rác rưởi" đối với vùng ký ức không được gán trị.

Cửa sổ **Memory Window** có sẵn cho đoạn mã managed và unmanaged. Khi quan sát nội dung của ký ức trên cửa sổ Memory, bạn có thể "lèo lái" (navigate) giữa biển ký ức sử dụng thanh rảo (scrollbar) thuộc thành phần giao diện người sử dụng của Window hoặc khỏ vào một vị chỉ ký ức (memory address) vào ô **Address**. Nếu bạn biết vị chỉ ký ức của một mẫu dữ liệu mà bạn muốn xem, bạn khỏ vào vị chỉ ký ức là cách dễ dàng nhất. Bạn có thể khỏ vào một con trỏ (pointer) chĩa về mục dữ liệu bạn muốn nhìn xem hoặc một biểu thức sử dụng vị chỉ của một tác tử để lấy vị chỉ của một mục dữ liệu.



Hình 3.7 : Cửa sổ Memory Window.

Trên đỉnh cửa sổ Memory có một thanh công cụ gồm 3 ô control: **Address, Reevaluate Automatically** và **Columns**

- **Address** Muốn rảo qua màn hình đi về một vị chỉ ký ức nào đó, bạn khỏ vào ô này vị chỉ nhắm tới rồi ấn <Enter>. Vị chỉ có thể là số, một tên hàm hoặc một biểu thức.

- **Reevaluate Automatically** Xác định liệu xem trị được hiển thị là "sống động" (live), nghĩa là tự động tính toán lại khi chương trình chạy, hoặc là "lưu tồn" (persitent).

- **Columns** Cho đặt để số cột khi hiển thị ký ức.

"Uốn nắn" cách hiển thị cửa sổ Memory

Bạn có thể "uốn nắn" (customize) cách cửa sổ **Memory** cho hiển thị nội dung ký ức dữ liệu theo nhiều dạng thức (format) khác nhau, bao gồm kiểu hexa hoặc kiểu thập phân. Theo mặc nhiên, nội dung ký ức hiện lên như là những số nguyên one-byte theo dạng thức hexadecimal, và số côt được tự động ấn định bởi chiều rộng màn hình.

Muốn thay đổi dạng thức trình bày nội dung ký ức

1. Bạn right-click lên cửa sổ Memory.

2. Từ trình đơn shortcut, bạn chọn dạng thức bạn nhắm tới: hexadecimal display, signed display, unsigned display, 1-byte integer v.v..

Muốn thay đổi số cột trên cửa sổ Memory

1. Trên thanh công cụ ở đầu cửa sổ **Memory**, bạn tìm đến ô liệt kê **Columns**.

2. Từ ô liệt kê **Columns** này, bạn chọn ra số cột bạn muốn cho hiển thị hoặc bạn chọn mục **Auto** tự động chỉnh theo chiều dài cửa sổ.

Page Up hoặc Page Down trên cửa sổ Memory

Khi nhìn xem nội dung ký ức trên cửa sổ Memory hoặc cửa sổ Disassembly, bạn có thể dùng thanh vertical scrollbar để cho lên xuống khoảng ký ức được hiển thị. Bạn click lên thanh này phía trên (page up) hoặc phía dưới (page down) cái ô nhỏ ở giữa (gọi là thumb). Bạn để ý là thumb hoạt động hơi kỳ kỳ vì nó luôn nằm ở giữa không nhúc nhích.

Muốn lên xuống một chỉ thị

- bạn click lên ô mũi tên ở trên đầu hoặc ở cuối thanh vertical scrollbar.

Muốn "chuyển bật" (toggling) định trị biểu thức sống động trên Memory Window

Nếu bạn không muốn nội dung của cửa sổ **Memory** thay đổi khi chương trình thi hành, bạn có thể cho "tắt" việc định trị biểu thức sống động (live expression evaluation). Bạn theo thủ tục sau đây:

1. Bạn right-click cửa sổ Memory.

2. Từ trình đơn shortcut, bạn click **Reevaluate Automatically**.

Muốn "chuyển bật" thanh công cụ Memory Window

Bạn có thể cất dấu hoặc hiển thị thanh công cụ trên đỉnh cửa sổ **Memory**. Bạn theo thủ tục sau đây:

1. Bạn right-click trên cửa sổ Memory.

2. Từ trình đơn shortcut, bạn click lên mục **Show Toolbar**. Thanh công cụ sẽ hiện lên hoặc biến mất tùy theo tình trạng trước đó.

Chọn lấy một vị trí ký ức

Nếu bạn muốn di chuyển ngay lập tức về một vị trí được chọn trên ký ức, bạn có thể lôi thả hoặc khỏ trị vào ô **Address;** ô này chấp nhận những biểu thức cũng như những trị số. Theo mặc nhiên, **Memory Window** xem một biểu thức Address như là một biểu thức sống (live expression), được định trị lại khi chương trình thi hành. Live expression có thể hữu ích. Thí dụ, bạn có thể dùng chúng để xem ký ức có liên quan đến con trỏ (pointer).

Muốn chọn một vị trí ký ức sử dụng lôi thả

1. Từ bất cứ cửa sổ nào, bạn chọn lấy một vị chỉ ký ức hoặc một biến con trỏ (pointer variable) chứa một vị chỉ ký ức.

2. Bạn lôi vị chỉ hoặc con trỏ về cửa sổ **Memory**, rồi thả lên trên ấy.

Muốn chọn một vị trí ký ức bằng cách hiệu đính

1. Trên cửa sổ **Memory**, bạn chọn ô **Address**.

2. Bạn khỏ hoặc dán vị chỉ mà bạn muốn xem rồi ấn **<ENTER>**.

Theo dõi một con trỏ thông qua cửa sổ Memory

Trên những ứng dụng mã nguyên sinh, bạn có thể dùng tên thanh ghi như là live expressions. Thí dụ, bạn có thể dùng stack pointer để theo dõi stack. Bạn theo dõi một con trỏ thông qua cửa sổ Memory, bằng cách theo thủ tục sau đây:

1. Trên ô **Address** trên cửa sổ Memory, bạn khỏ vào một pointer expression. Biến con trỏ phải ở trong phạm vi hiện hành. Tùy theo ngôn ngữ, có thể bạn cần phải dereference con trỏ.

2. Tiếp theo, bạn ấn **<ENTER>**.

Giờ đây, khi bạn dùng một lệnh thi hành, **Step** chẳng hạn, vị chỉ ký ức được hiển thị sẽ tự động thay đổi khi con trỏ thay đổi.

3.1.3.10 Sử dụng cửa sổ Disassembly

Bạn có thể cho hiện lên cửa sổ Diassembly Window (hình 3.8) bằng cách ấn tổ hợp phím **<Ctrl+Alt+D>** hoặc chọn mục trình đơn **Debug | Windows | Diassembly**.

Cửa sổ **Disassembly Window** cho hiển thị những chỉ thị cơ bản được tạo ra đối với đoạn mã nguồn khi được biên dịch. Thay vì bắt bạn đọc chỉ thị ở dạng nhị phân hoặc hexa, những chỉ thị này được trình bày theo dạng ngôn ngữ hợp ngữ. Từ diassembly (rã ngữ ?) là làm ngược lại với assembly (hợp ngữ). Ngoài ra, cửa sổ có thể cho bạn thấy những thông tin tùy chọn như sau:

- **Show Address**: Vị chỉ ký ức nơi mà mỗi chỉ thị "cư ngụ". Đối với những ứng dụng mã nguyên sinh, đây là vị chỉ hiện thời. Còn đối với đoạn mã C# hoặc đoạn mã managed, thì đây là vị chỉ pseudomemory theo đấy khoảng ký ức mà ứng dụng chiếm dụng bắt đầu từ vị trí zero.

- **Show Source Code**: Đoạn mã nguồn từ đó đoạn mã assembly được dẫn xuất.

- **Show Code Bytes**: Code bytes, là cách biểu diễn theo byte của chỉ thị mã máy hiện thời.

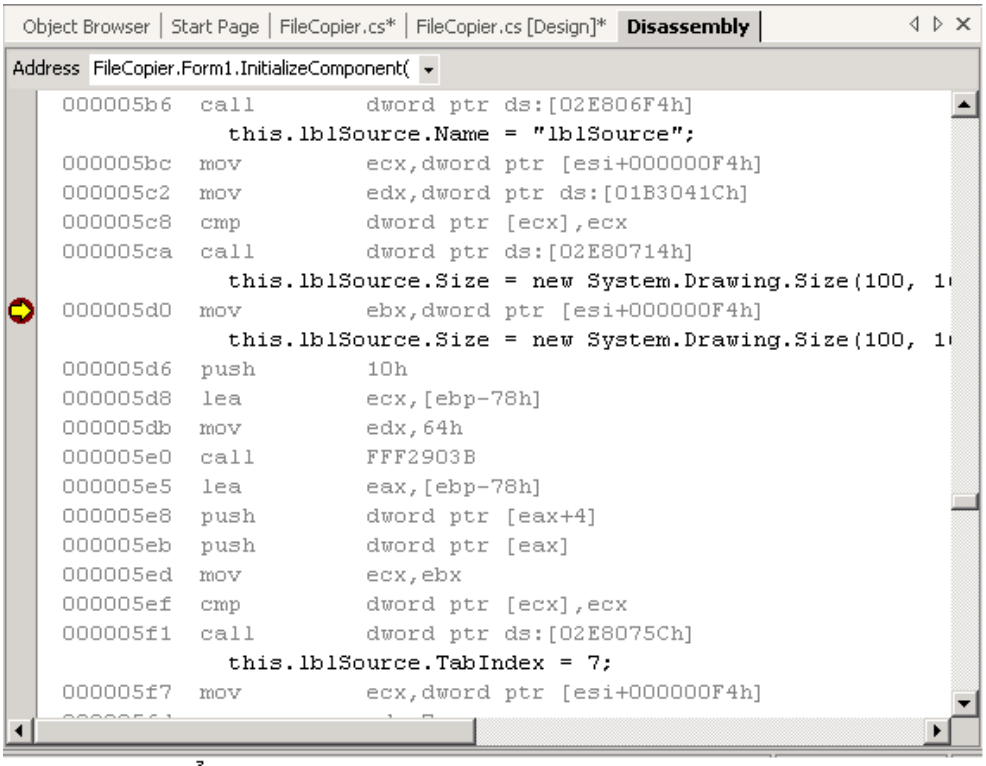
- **Show Symbolic Names**: Các tên tượng trưng đối với vị chỉ ký ức.

- **Show Line Numbers**: Hàng được đánh số tương ứng với mã nguồn.

Muốn on/off các thông tin tùy chọn kể trên, bạn có thể right-click lên cửa sổ để cho hiện lên trình đơn shortcut, rồi bạn chọn click mục nào bạn nhắm tới.

Đoạn mã viết theo ngôn ngữ hợp ngữ (assembly-language) thường gồm những "ký hiệu dễ nhớ" (mneumonics), là những viết tắt đối với tên các chỉ thị, và những ký hiệu (symbols) tượng trưng cho các biến, thanh ghi và hằng. Mỗi chỉ thị máy được biểu diễn bởi một assembly-language mneumonic, theo sau thông thường bởi một hoặc nhiều biến, thanh ghi và hằng.

Nếu bạn rành ngôn ngữ hợp ngữ thì bạn mới có thể tận dụng được cửa sổ Disassembly. Ví đoạn mã assembly thường dựa chặt chẽ vào thanh ghi nên bạn sẽ thấy rất hữu ích khi phối hợp sử dụng với cửa sổ **Registers Window**, cho phép bạn quan sát nội dung các thanh ghi.



Hình 3-8 : Cửa sổ Disassembly Window

Nếu bạn muốn xem các chỉ thị máy ở dạng thô sơ, bạn có thể sử dụng cửa sổ Memory dùng cho mục đích này hoặc chọn **Code Bytes** từ shortcut menu trên cửa sổ Disassembly.

Trên đỉnh cửa sổ Disassembly là một thanh công cụ chỉ gồm một ô control duy nhất: **Address**. Ô này có công dụng là muốn cho màn hình rảo về một vị chỉ ký ức nào đó, bạn khỏ vị chỉ vào ô **Address,** rồi ấn <Enter>. Vị chỉ có thể là một con số hoặc tên một hàm. Bạn có thể cất dấu thanh công cụ bằng cách right-click lên cửa sổ rồi click mục chọn **Show Toolbar** trên trình đơn shortcut.

Muốn cho lên xuống một trang ký ức trên cửa sổ Disassembly, bạn có thể xem mục "Page Up hoặc Page Down trên cửa sổ Memory.

Một mũi tên màu vàng nằm ở biên trái đánh dấu vị trí của điểm thi hành hiện hành. Đối với đoạn mã nguyên sinh, điều này tương ứng với cái đếm chương trình của CPU (program counter). Vị trí này cho biết chỉ thị kế tiếp sẽ được thi hành.

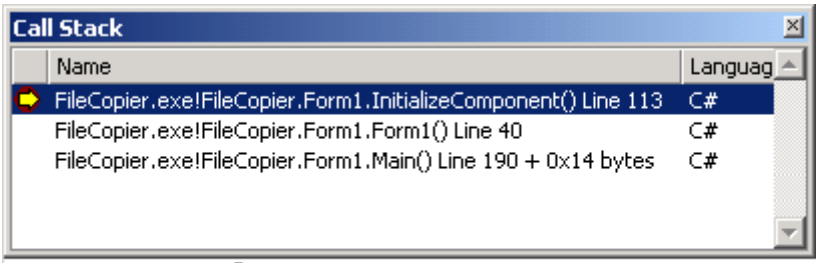
3.1.3.11 Sử dụng cửa sổ Call Stack

Call Stack là gì?

Khi một hàm được triệu gọi, thì tên hàm được ấn (push) vào stack, và khi hàm trở về, thì tên hàm được tống ra (pop). Hàm nào hiện đang được thi hành thì nằm ở trên đầu stack, còn các hàm cũ chưa hoàn thành công tác thì nằm ở dưới. Do đó stack này được gọi là call stack, (stack chứa các triệu gọi hàm) cho phép bạn theo dõi dấu vết của hàm nào được triệu gọi bởi chương trình bạn đang gỡ rối. Nói tóm lại, stack là một vùng ký ức dành riêng hoạt động như tờ "giấy nháp" đối với CPU. CPU có một thanh ghi được gọi là *stack pointer* (SP) dùng qui chiếu vùng ký ức mà stack data chiếm dụng.

Sử dụng cửa sổ Call Stack thế nào?

Muốn mở cửa sổ Call Stack, bạn phải đang ở chế độ Debug. Bạn chọn mục **Debug | Windows | Call Stack** hoặc ấn tổ hợp phím <Ctrl + Alt + C>. Hình 3-9.



Hình 3-9 : Cửa sổ Call Stack

Cửa sổ **Call Stack** cho phép bạn nhìn xem tên các hàm hành sự trên call stack, các kiểu dữ liệu thông số, và trị các thông số. Thông tin Call stack chỉ được cho thấy khi bạn đang gỡ rối chương trình và đang ở chế độ break mode. Bạn thấy:

Cột Name Cho thấy tên của mỗi hàm trên call stack. Để bạn có thể phân biệt giữa những triệu gọi khác nhau đối với cùng hàm, mỗi tên sẽ được trình bày với thông tin tùy chọn sau đây:

- Tên module (không có sẵn đối với SQL debugging)
- Tên thông số
- Kiểu dữ liệu thông số
- Trị thông số
- Số hàng
- Di số byte (byte offset) (không có sẵn đối với SQL debugging)

Bạn có thể cho on/off thông tin tùy chọn này bằng cách right-click cửa sổ Call Stack để cho hiện lên trình đơn shorcut rồi bạn chọn mục nào đó theo tùy thích.

Cột Language Cột này cho biết ngôn ngữ nào hàm được viết. Sẽ để trắng nếu không xác định được loại ngôn ngữ nào.

Trên cửa sổ Call Stack, có một mũi tên màu vàng dùng nhận diện stack frame theo đấy con trỏ thi hành (execution pointer) hiện đang đứng. Theo mặc nhiên, đây là frame theo đấy mã nguồn xuất hiện trên cửa sổ mã nguồn và trên cửa sổ Disassembly và theo đấy các biến cục bộ xuất hiện trên các cửa sổ Locals, Watch, và Autos. Nếu bạn muốn nhìn xem đoạn mã và các biến đối với một frame khác, bạn có thể bật qua (switch) frame này trên cửa sổ Call Stack. Bạn theo thủ tục sau đây: (1) Trên cửa sổ **Call Stack**, bạn right-click lên frame mà bạn muốn xem đoạn mã và dữ liệu. (2) Từ trình đơn shortcut, bạn chọn click mục **Switch to Frame**.

Một hình tam giác màu vàng xuất hiện cạnh frame bạn đã chọn. Con trỏ thi hành vẫn ở nguyên tại frame nguyên thủy, được đánh dấu bởi mũi tên màu vàng. Nếu bạn chọn mục **Step** hoặc **Continue** từ trình đơn **Debug,** việc thi hành sẽ tiếp tục trên frame nguyên thủy, chứ không phải trên frame bạn vừa chọn.

Trình đơn shortcut trên cửa sổ Call Stack còn có những mục chọn:

- **Include Calls To/From Other Threads**: theo mặc nhiên cửa sổ Call Stack không cho hiển thị những triệu gọi hàm về hoặc từ một thread khác. Tuy nhiên, nếu bạn muốn xem những triệu gọi hàm này, thì bạn chọn mục này.

- **Go To Source Code**: bạn có thể xem nhanh đoạn mã nguồn đối với một hàm nào đó có mặt trên call satck, bạn chọn right-click lên hàm này rồi chọn mục này trên trình đơn shorcut.

- **Go To Disassembly**: bạn có thể xem nhanh đoạn mã disassembly đối với một hàm nào đó có mặt trên call satck, bạn chọn right-click lên hàm này rồi chọn mục này trên trình đơn shorcut

Tới đây xem như bạn đã "cưỡi ngựa xem hoa" các cửa sổ khác nhau dùng trong việc gỡ rối. Về sau, bạn sẽ trở lại xem các cửa sổ này được sử dụng thế nào.

3.2 Điều khiển việc thi hành chương trình

Visual Studio Debugger cung cấp những lệnh cực mạnh cho phép bạn điều khiển việc thi hành chương trình của bạn. Sau đây chúng tôi liệt kê những công tác mà bạn có thể thực hiện:

- Bắt đầu (hoặc tiếp tục) thi hành (Start/Continue) - mục 3.2.1
- Ngắt thi hành - mục 3.2.2
- Ngưng thi hành - mục 3.2.3
- Đi lần từng bước xuyên qua ứng dụng - mục 3.2.4
- Chạy về một nơi chỉ định - mục 3.2.5
- Cho đặt để điểm thi hành - mục 3.2.6

Tất cả các công tác kể trên đều được thực hiện thông qua trình đơn **Debug,** hoặc thanh công cụ Debug (bạn chọn **View | Toolbar | Debug & Debug Location** để cho hiện lên thanh công cụ). Xem hình 3-10.

3.2.1 Bắt đầu gỡ rối

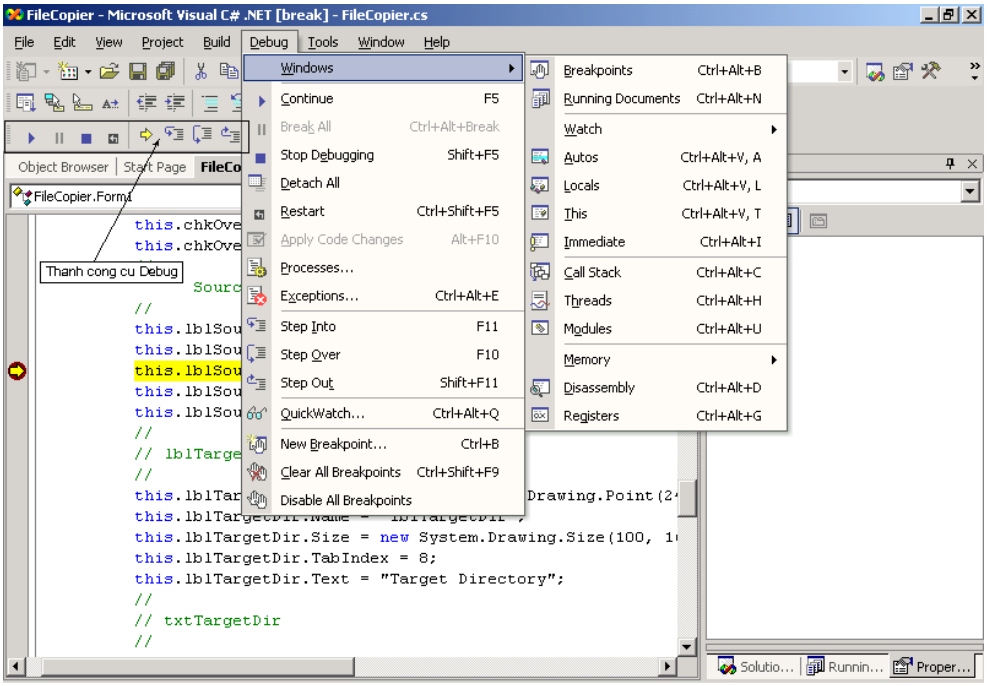
Muốn bắt đầu gỡ rối bạn có hai cách:

- Cách thứ nhất: bạn chọn mục trình đơn **Debug | Start**, hoặc **Debug | Step Into**, hoặc **Debug | Step Over.** Nếu bạn chọn **Start**, ứng dụng của bạn sẽ bắt đầu từ đó và chạy cho tới khi gặp chốt ngừng. Bạn có thể ngắt thi hành bất cứ lúc nào để quan sát các trị, thay đổi biến và quan sát tình trạng chương trình của bạn. Nếu bạn chọn **Step Into** hoặc **Step Over**, ứng dụng của bạn sẽ bắt đầu từ đó và thi hành rồi ngắt trên hàng đầu tiên.

- Cách thứ hai: bạn right-click cửa sổ mã nguồn rồi chọn **Run to Cursor** từ trình đơn shortcut. Nếu bạn chọn **Run to Cursor**, ứng dụng của bạn sẽ bắt đầu từ đó và chạy

cho tới khi gặp chốt ngừng hoặc nơi con nháy (cursor) đang nằm tùy theo cái nào tới trước. Bạn có thể đặt để vị trí con nháy trên cửa sổ mã nguồn. Trong vài trường hợp việc thi hành không xảy ra. Đây có nghĩa là việc thi hành không bao giờ đạt đến đoạn mã mà con nháy được đặt để.

Giải pháp (solution) có thể chứa nhiều dự án. Trong trường hợp này, bạn có thể chọn một dự án làm startup project (dự án khởi động) được khởi động bởi các lệnh thi hành trên trình đơn **Debug**. Một cách khác, bạn cũng có thể khởi động một dự án được chọn từ **Solution Explorer**.



Hình 3-10: Trình đơn Debug và Debug | Windows và thanh công cụ Debug

3.2.2 Ngắt thi hành (breaking execution)

Khi bạn đang gỡ rối ứng dụng với Visual Studio Debugger, hoặc ứng dụng đang chạy (thi hành) hoặc nó đang ở chế độ break mode. Nhiều chức năng của debugger, chẳng hạn định trị biểu thức trên cửa sổ **Watch Window**, chỉ có hiệu lực khi ở chế độ break mode.

Debugger sẽ cho ngắt thi hành chương trình khi việc thi hành đạt đến một chốt ngừng hoặc khi biệt lệ (exception) xảy ra.

Muốn ngắt thi hành chương trình bằng tay

- Từ trình đơn **Debug**, bạn chọn **Break All,** hoặc ấn tổ hợp phím <Ctrl + Alt + Break>.

  Debugger sẽ ngưng thi hành tất cả các chương trình đang chạy dưới sự kiểm tra của Debugger. Chương trình không thoát ra (exit) và bạn có thể tiếp tục lại bất cứ lúc nào. Debugger và ứng dụng giờ đây đang ở break mode.

  Ghi chú: Nếu không thể ngừng một chương trình "nguyên sinh" vì một lý do gì đó (thí dụ, chương trình đang thi hành ở chế độ kernel mode), thì Debugger sẽ cho đóng băng tất cả các thread và mô phỏng một break. Đây được gọi là một "soft break." Nếu bạn cho thi hành một lệnh như **Go** hoặc **Step** sau một soft break, Debugger sẽ kích hoạt lại (thaws) các threads. Kết quả là có thể bạn cần chọn đến hai lần lệnh **Go** hoặc **Step** để việc thi hành tiếp tục lại. Một khung thông báo sẽ báo cho bạn biết khi nào xảy ra soft break.

Nếu bạn đang gỡ rối nhiều chương trình, lệnh **Break All** hoặc chốt ngừng sẽ tác động lên tất cả các chương trình được gỡ rối theo mặc nhiên. Bạn có thể thay đổi tình trạng mặc nhiên này nếu bạn muốn chỉ ngắt thi hành chương trình hiện hành mà thôi.

Muốn thay đổi cách thức ngắt khi gỡ rối nhiều chương trình

1. Từ trình đơn **Tools,** bạn chọn mục **Options**. Khung đối thoại hiện lên.

2. Trên khung đối thoại **Options**, bạn chọn folder **Debugging**.

3. Trên folder **Debugging**, bạn chọn trang **General**.

4. Trên nhóm **General**, bạn chọn đánh dấu hoặc xoá ô duyệt **In break mode, only stop execution of the current process**.

3.2.3 Ngưng thi hành

Ngưng thi hành có nghĩa là chấm dứt (hoặc thoát khỏi) chương trình mà bạn đang gỡ rối và chấm dứt luôn chầu gỡ rối. Không nên nhầm lẫn ngưng thi hành với ngắt thi hành; hành động ngắt chỉ tạm thời ngưng thi hành chương trình bạn đang gỡ rối và chầu gỡ rối vẫn đang còn hoạt động.

Muốn ngưng hoàn toàn việc gỡ rối

- Từ trình đơn **Debug**, bạn chọn **Stop Debugging,** hoặc ấn tổ hợp phím <Shift + F5>.

Chú ý: Không phải bao giờ **Stop Debugging** cũng chấm dứt chương trình. Nếu bạn gắn liền với chương trình bạn đang gỡ rối, thường thì **Stop Debugging** sẽ tách khỏi chương trình nhưng vẫn để cho nó chạy. Bạn có thể nhìn xem và thay đổi cách hành xử này bằng cách làm việc với khung đối thoại **Processes** (do việc chọn **Debug | Process…**)

Nếu bạn muốn ngưng thi hành chương trình hiện bạn đang gỡ rối, và cho chạy ngay lập tức lại một thi hành mới, bạn có thể sử dụng lệnh **Restart**.

Muốn ngưng gỡ rối rồi cho chạy lại

- Từ trình đơn **Debug,** bạn chọn mục **Restart,** hoặc ấn tổ hợp phím <Ctrl + Shift + F5>.

Việc gỡ rối sẽ tự động ngưng nếu bạn thoát khỏi chương trình bạn đang gỡ rối (còn nếu bạn đang gỡ rối nhiều chương trình, thì việc gỡ rối tiếp tục cho đến khi bạn thoát khỏi chương trình chót). Nếu bạn đang gỡ rối một dự án do một ứng dụng khác làm chủ, chẳng hạn một dự án web do Internet Explorer "chủ xị", thì việc gỡ rối sẽ ngưng nếu bạn thoát khỏi ứng dụng "chủ xị"(nghĩa là Internet Explorer).

Trên Visual Basic và C#, nếu bạn đang gỡ rối một dịch vụ và ứng dụng khách sử dụng dịch vụ này chấm dứt, thì việc gỡ rối dịch vụ cũng ngưng luôn.

Lệnh **Restart** cho phép bạn bắt đầu lại. Nó cho nạp lại chương trình vào ký ức, bỏ qua những trị hiện hành của tất cả các biến. Các chốt ngừng và các biểu thức quan sát (watch expression) vẫn còn hiệu lực.

3.2.4 Cho thi hành từng bước một (Stepping)

Một trong những thủ tục phổ biến nhất trong việc gỡ rối là "lần theo từng bước" gọi là *stepping*, nghĩa là thi hành đoạn mã theo từng hàng một trong một lúc. Trình đơn **Debug** cung cấp cho bạn 3 lệnh để "lần theo từng bước" xuyên qua đoạn mã: **Step Into, Step Over** và **Step Out.**

Step Into và **Step Over** chỉ khác nhau ở một điểm: cách chúng thụ lý các triệu gọi hàm (function calls). Cả hai lệnh đều yêu cầu Debugger thi hành hàng kế tiếp trên đoạn mã. Nếu hàng này là một triệu gọi hàm, thì **Step Into** chỉ thi hành bản thân hàm mà thôi, rồi sau đó ngưng ở hàng đầu tiên của đoạn mã trong lòng hàm. Còn **Step Over** thì lại cho thi hành trọn vẹn hàm, rồi ngừng ở hàng đầu tiên nằm ngoài hàm. Bạn sử dụng **Step Into** nếu bạn muốn nhìn xem trong ruột triệu gọi hàm, còn dùng **Step Over** nếu bạn muốn tránh chui vào hàm. Có thể nói lệnh **Step Over** là lệnh được sử dụng nhiều nhất, cho

phép bạn đi lần theo từng hàng chương trình. Nếu bạn dùng lệnh này trên một triệu gọi hàm, thì hàm sẽ được thi hành không chui vào hàm. Nếu muốn chui vào hàm, thì dùng **Step Into.**

Trên một triệu gọi hàm nằm lồng nhau, **Step Into** sẽ chui vào trong hàm nằm sâu nhất. Nếu bạn dùng **Step Into** trong một triệu gọi hàm như **Func1(Func2())**, thì Debugger sẽ chui vào hàm **Func2**. Nếu bạn muốn chọn một hàm nằm lồng nào đó để chui vào, bạn nên sử dụng lệnh **Step Into Specific** từ trình đơn shortcut (chỉ C/C++ nguyên sinh mà thôi — nếu bạn đang dùng Managed Extensions đối với C++, lệnh này không hiệu lực).

Bạn sử dụng **Step Out** khi bạn đang ở trong ruột một triệu gọi hàm và bạn muốn trở về hàm phát đi triệu gọi. **Step Out** tiếp tục thi hành đoạn mã cho tới khi hàm trở về, rồi cho ngắt ở điểm trở về trong hàm triệu gọi (calling function). Debugger sẽ tự động ngừng ở hàng kế tiếp, cho dù không có chốt ngừng đặt ở chỗ này.

Bạn không thể chọn các lệnh **Step** khi ứng dụng bạn đang chạy. Các lệnh này chỉ hiện diện khi bạn đang ở chế độ break mode hoặc trước khi bạn khởi động ứng dụng.

Muốn chui vào chương trình chưa được thi hành

- Từ trình đơn **Debug**, bạn chọn ấn mục **Step Into**, hoặc ấn nút <F11>

Muốn chui vào chương trình khi đang gỡ rối

1. Debugger phải ở chế độ break mode.
2. Từ trình đơn **Debug**, bạn chọn ấn **Step Into**, **Step Out**, hoặc **Step Over**.

3.2.5 Cho chạy về một vị trí nhất định nào đó

Thỉnh thoảng, khi đang gỡ rối, bạn lại muốn thi hành ở một chỗ nào đó trên đoạn mã, rồi ngắt. Nếu bạn có một breakpoint được đặt để ở chỗ này, bạn có thể theo thủ tục sau đây:

Muốn nhảy về một nơi nào đó nếu bạn có đặt sẵn một chốt ngừng

- Trên trình đơn **Debug**, bạn click **Start** hoặc **Continue**, hoặc ấn phím <F5>.

Tuy nhiên, không nhất thiết phải cho đặt một chốt ngừng trong tất cả mọi trường hợp. Visual Studio Debugger cung cấp cho bạn những lệnh đặc thù để chạy về vị trí con nháy hoặc về một hàm nào đó..

3.2.6 Cho đặt điểm thi hành

Trên Visual Studio Debugger, bạn có thể di chuyển điểm thi hành, cho đặt để về câu lệnh kế tiếp (hoặc chỉ thị ngôn ngữ assembly) cần phải thi hành. Một mũi tên màu vàng trên vùng biên trái của cửa sổ mã nguồn (hoặc trên cửa sổ Disassembly) đánh dấu cho biết vị trí hiện hành của điểm thi hành. Bằng cách di chuyển điểm thi hành, bạn có thể nhảy bỏ một khúc trên đoạn mã hoặc trở về hàng đã được thi hành trước đó. Điều này có thể hữu ích trong vài trường hợp, thí dụ khi bạn muốn nhảy bỏ một khúc của đoạn mã có chứa bug và tiếp tục gỡ rối các phần kế tiếp.

Cẩn thận: Thay đổi điểm thi hành làm cho cái đếm (counter) trong chương trình nhảy trực tiếp về vị chỉ ký ức mới. Bạn nên sử dụng lệnh này một cách cẩn thận. Bạn để ý: các chỉ thị giữa các điểm thi hành cũ và mới sẽ không được thi hành; nếu bạn cho di chuyển điểm thi hành lui lại sau, thì các chỉ thị xen kẽ sẽ không được thi hành; còn di chuyển điểm thi hành về một hàm khác hoặc phạm vi khác thường gây ra "ung thối" trên call-stack, gây ra sai lầm vào lúc chạy hoặc biệt lệ; và nếu bạn cố di chuyển điểm thi hành về một phạm vi khác, Debugger sẽ mở một khung đối thoại báo động cho bạn cơ may hủy bỏ tác vụ.

Ghi chú Trên managed code, bạn không thể thay đổi điểm thi hành sau khi một biệt lệ được tung ra.

Bạn không thể cho đặt điểm thi hành trong khi chương trình bạn đang hoạt động. Bạn phải ở trong chế độ break mode.

Muốn đặt để câu lệnh kế tiếp phải thi hành

- Trên một cửa sổ mã nguồn hoặc cửa sổ Disassembly, bạn right-click lên câu lệnh hoặc lên chỉ thỉ ngôn ngữ assembly mà bạn muốn cho thi hành kế tiếp và chọn ấn mục **Set Next Statement** từ trình đơn shortcut. Hàng lệnh sẽ được tô màu vàng với mũi tên cũng màu vàng chĩa về câu lệnh.

Nếu điểm thi hành hiện hành trong cùng tập tin mã nguồn giống như câu lệnh bạn muốn đặt để, bạn có thể di chuyển điểm thi hành bằng cách dùng con chuột lôi mũi tên vàng.

Muốn đặt để câu lệnh kế tiếp phải thi hành *(phương án thay thế)*

- Trên một cửa sổ mã nguồn, bạn click dấu chỉ điểm thi hành (mũi tên màu vàng) rồi lôi đi về vị trí trên cùng tập tin mã nguồn, nơi mà bạn muốn cho đặt để câu lệnh kế tiếp phải thi hành.

3.2.6.1 Nhảy về vị trí con nháy

Bạn có thể yêu cầu Debugger chạy ứng dụng cho tới khi đạt đến vị trí mà con nháy (cursor) được đặt để. Vị trí này có thể nằm trên cửa sổ mã nguồn hoặc trên cửa sổ Disassembly.

Muốn chạy về cursor trên cửa sổ mã nguồn

1. Trên một tập tin mã nguồn, bạn click lên một hàng mã nguồn để cho con nháy di về đấy.

2. Bạn right-click cho trình đơn shorcut hiện lên, rồi chọn mục **Run To Cursor**.

3. Chương trình sẽ bị ngắt thi hành ngay ở câu lệnh chứa con nháy.

Muốn chạy về cursor trên cửa sổ Disassembly

1. Nếu cửa sổ **Disassembly** chưa hiện lên, bạn chọn mục **Windows** từ trình đơn **Debug,** rồi chọn tiếp mục **Disassembly** (bạn phải ở chế độ break mode để có thể nhìn thấy cửa sổ **Disassembly**).

2. Trên cửa sổ **Disassembly**, bạn click lên một hàng để cho di con nháy về điểm này

3. Bạn right-click khi đang ở trên cửa sổ **Disassembly** để cho trình đơn shortcut hiện lên, rồi chọn mục **Run To Cursor**.

3.2.6.2 Chạy về một hàm được chỉ định

Bạn có thể yêu cầu Debugger cho chạy ứng dụng cho tới khi đạt đến hàm nào đó được chỉ định. Bạn khai báo hàm theo tên (sử dụng thanh công cụ - toolbar) hoặc bạn có thể chọn tên hàm từ call stack.

Muốn chạy về một hàm được chỉ định trên một tập tin mã nguồn

1. Trên thanh công cụ chuẩn, bạn khỏ vào tên hàm trên khung đối thoại **Find** rồi ấn **<ENTER>**. Con nháy sẽ di chuyển về hàm được chỉ định hoặc một triệu gọi hàm này.

2. Lặp lại bước 1, nếu thấy cần thiết.

3. Trên cửa sổ mã nguồn, bạn right-click lên phần khai báo hàm để cho trình đơn shortcut hiện lên, rồi chọn mục **Run To Cursor**

Muốn chạy về một hàm nằm trên call stack

1. Debugger phải ở chế độ break mode.

2. Cho mở cửa sổ Call Stack, nếu thấy cần thiết. (bạn chọn mục **Call Stack** từ trình đơn **Debug** menu, **Windows** submenu.)

3. Trên cửa sổ Call Stack, bạn right-click tên hàm rồi chọn mục **Run To Cursor** từ trình đơn shortcut.

Muốn chèn một function breakpoint trên một cửa sổ mã nguồn

1. Trên cửa sổ mã nguồn, bạn click lên tên hàm nơi mà bạn muốn chèn một chốt ngừng. Một trình đơn shortcut hiện lên.

2. Bạn chọn mục **New Breakpoint** từ trình đơn shortcut này. Khung đối thoại **New Breakpoint** hiện lên Khung đối thoại **New Breakpoint** xuất hiện với tab **Function**. Tên hàm bạn đã chọn xuất hiện trên ô **Function** còn ngôn ngữ được viết trên ô **Language.** Theo mặc nhiên, các ô **Line** và **Character** được cho về 1. Nếu bạn chấp nhận những trị mặc nhiên này, breakpoint sẽ được đặt để ở đầu hàm..

3. Bạn có thể hiệu đính các ô **Line** và **Character** nếu bạn muốn đặt breakpoint về ở một nơi khác trên hàm. Bạn click **OK**

3.3 Chốt ngừng

Một chốt ngừng yêu cầu Debugger cho ngắt ứng dụng (tạm ngưng thi hành) tại một điểm nào đó hoặc khi một điều kiện nào đó xảy ra. Khi biến cố ngắt xảy ra, chương trình và Debugger ở trong chế độ break mode.

3.3.1 Các loại chốt ngừng và thuộc tính

Visual Studio Debugger có 4 loại chốt ngừng:

- ***Một chốt ngừng hàm*** (function breakpoint) làm cho chương trình bị ngắt khi việc thi hành đạt đến một vị trí nào đó được chỉ định trong lòng một hàm nào đó được chỉ định.

- ***Một chốt ngừng tập tin*** (file breakpoint) sẽ làm ngắt chương trình khi việc thi hành đạt đến một vị trí nào đó được chỉ định trong lòng một tập tin nào đó.

- ***Một chốt ngừng vị chỉ*** (address breakpoint) sẽ làm ngắt chương trình khi việc thi hành đạt đến một vị chỉ ký ức nào đó được chỉ định

- ***Một chốt ngừng dữ liệu*** (data breakpoint) sẽ làm ngắt chương trình khi trị của một biến thay đổi. Bạn có thể đặt để một data breakpoint trên một biến toàn cục hoặc trên một biến cục bộ, trên một phạm vi cao nhất của một hàm (chỉ C++ mà thôi).

Để cung cấp sức mạnh và tính uyển chuyển, bạn có thể thay đổi cách hành xử của một chốt ngừng bằng cách thêm những thuộc tính:

- Một thuộc tính **Hit Count**, xác định bao nhiêu lần một chốt ngừng phải bị đụng (hit) trước khi việc thi hành bị ngắt.

- Một thuộc tính **Condition**, là một biểu thức xác định liệu xem chốt ngừng bị đụng hoặc bị nhảy bỏ qua.

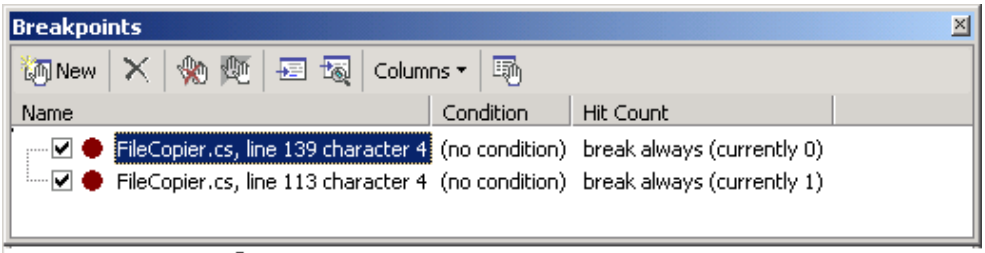
Bạn có thể cho đặt để (set), cho hiệu lực (enable), vô hiệu hóa (disable), hiệu đính (edit), hoặc gỡ bỏ (delete) các chốt ngừng trên cửa sổ mã nguồn, trên cửa sổ **Breakpoints** (Hình 3-11), hoặc trên cửa sổ **Disassembly**.

3.3.2 Cửa sổ Breakpoints

Cửa sổ **Breakpoints** liệt kê tất cả các chốt ngừng hiện hành được đặt để trong chương trình và cho hiển thị thuộc tính của chúng. Trong cửa sổ **Breakpoints**, bạn có thể đặt để (tạo) những chốt ngừng mới, gỡ bỏ chốt ngừng, cho hiệu lực hoặc vô hiệu hóa chốt ngừng, hiệu đính các thuộc tính chốt ngừng hoặc đi về đoạn mã nguồn hoặc đoạn mã disassembly tương ứng với một chốt ngừng

Hình 3-11 trên cho thấy một thanh công cụ và một danh sách các chốt ngừng hiện được đặt để trong chương trình. Thanh công cụ chứa những công cụ sau đây (từ trái qua phải):

New (Breakpoint) : Nút này mở khung đối thoại **New Breakpoint,** cho phép bạn chủ động đặt để những mục chọn tùy ý để tạo một chốt ngừng mới.



Hình 3-11 : Cửa sổ Breakpoints.

Delete: Cho gỡ bỏ bất cứ chốt ngừng hiện đang được chọn. Bạn không thể undo tác vụ này.

Clear All Breakpoints: Cho gỡ bỏ tất cả các chốt ngừng hiện đang được đặt để. Bạn không thể undo tác vụ này.

Disable All Breakpoints: Cho vô hiệu hoá tất cả các chốt ngừng hiện đang được đặt để. Bạn không thể cho hiệu lực lại các chốt ngừng (undo tác vụ) bằng cách click nút này lại.

Go to Source Code: Nút này cho mở một cửa sổ mã nguồn, nếu thấy cần thiết, là cho thấy vị trí của các chốt ngừng được đặt để.

Go to Disassembly : Nút này cho hiện lên cửa sổ **Disassembly** window và cho thấy vị trí của các chốt ngừng được đặt để.
Columns: Đây là một ô liệt kê các cột thông tin xuất hiện trên danh sách các chốt ngừng.

Properties : Nút này cho mở khung đối thoại **Breakpoint Properties**, cho phép bạn hiệu đính bất cứ thuộc tính nào của chốt ngừng.

Danh sách các chốt ngừng chứa 3 cột thông tin theo mặc nhiên:

Name: Tên mô ta chốt ngừng, do debugger tạo ra dựa trên vị trí chốt ngừng và các thuộc tính khác, cho phép bạn nhận diện chốt ngừng và tuyển chọn chốt ngừng theo ý muốn của bạn. Có một ô checkbox nằm cạnh tên, cho phép bạn vô hiệu hoá hoặc cho hiệu lực chốt ngừng.

Condition: Một thuộc tính tùy chọn xác định liệu xem chương trình sẽ bị ngắt hay không khi đạt đến chốt ngừng. Điều kiện có thể là bất cứ điều kiện nào hợp lệ mà debugger nhận diện được. Debugger sẽ định trị và cho ngắt việc thi hành khi điều kiện là thỏa.

Hit Count: Một thuộc tính tùy chọn khác xác định liệu xem chương trình sẽ bị ngắt hay không khi đạt đến chốt ngừng. Nếu thuộc tính này không được đặt để, Debugger sẽ ngắt mỗi lần chốt ngừng bị đụng, với giả thuyết là biểu thức trên cột **Condition** là thoả. Hit count có thể báo cho Debugger phải ngắt khi chốt ngừng bị đụng lần thứ N, hoặc trên mỗi bội số N, hoặc trên lần thứ N và mỗi lần sau đó.

Các cột sử dụng không được hiển thị theo mặc nhiên, nhưng bạn có thể cho hiện lên bằng cách sử dụng nút cột **Columns**:

Language: Ngôn ngữ theo đấy chốt ngừng được đặt để.

Function: Tên của hàm nơi chốt ngừng được đặt để và vị trí trong lòng hàm.

File: Tên của tập tin nơi chốt ngừng được đặt để và số hàng trong lòng hàm.

Address: Vị chỉ ký ức nơi chốt ngừng được đặt để.

Data: Đối với một data breakpoint, đây là tên biến nơi mà chốt ngừng được đặt để và phạm vi theo đấy chốt ngừng được đặt để.

Program: Tên chương trình theo đấy chốt ngừng được đặt để.

Cửa sổ mã nguồn và **Disassembly** (Hình 3-8) cho bạn thấy vị trí của một chốt ngừng bằng cách tô màu phần văn bản nơi mà chốt ngừng được đặt để và cho hiển thị một ký hiệu ở biên phía tay trái.

3.3.2.1 Sử dụng cửa sổ Breakpoints

Cửa sổ **Breakpoints** (hình 3-11) liệt kê tất cả các chốt ngừng hiện hành được đặt để trong chương trình và cho hiển thị những thuộc tính của chúng. Trên cửa sổ **Breakpoints**, bạn có thể đặt để những chốt ngừng mới, gỡ bỏ chốt ngừng, cho có hiệu lực hoặc vô hiệu hóa chốt ngừng, cho hiệu đính các thuộc tính của chốt ngừng, hoặc tìm ra đoạn mã nguồn hoặc đoạn mã disassembly tương ứng với một chốt ngừng. Phần này sẽ mô tả những thông tin liên quan đến:

- *Hiển thị cửa sổ **Breakpoints.*** Từ trình đơn **Debug**, bạn chọn mục **Windows** rồi click **Breakpoints.**

- ***Nhìn xem thông tin về chốt ngừng*:** cửa sổ **Breakpoints** chứa một danh sách các chốt ngừng hiện hành được đặt để trong chương trình, gồm 3 cột thông tin theo mặc nhiên. Bạn có thể xem thông tin bổ sung bằng cách thêm vào nhiều cột nữa. Muốn

thế, trên thanh công cụ của cửa sổ **Breakpoints**, bạn click **Columns tool** và chọn tên cột bạn muốn cho hiển thị. Muốn cho ẩn mình một cột nào đó, bạn cho hủy chọn (deselect) tên cột trên thanh công cụ vừa kể trên. Bạn có thể sắp xếp thứ tự các cột bằng cách lôi thả column header.

- ***Chốt ngừng con-cái***. Khi bạn nhìn vào cửa sổ Breakpoints Window, có thể bạn để ý đến một tree control trên cột Name đối với vài chốt ngừng. Ô control cho phép bạn bung một chốt ngừng có những "chốt ngừng con-cái"(child breakpoint). Chốt ngừng con-cái xảy ra khi hai hoặc nhiều chốt ngừng được tạo từ một yêu cầu chốt ngừng đơn độc, và xảy ra lúc đặt để chốt ngừng hoặc về sau. Nếu bạn đặt để một chốt ngừng trên một hàm bị nạp chồng (overloaded), thì Debugger sẽ tạo chốt ngừng con-cái cho mỗi hàm duy nhất.

  Nếu bạn gắn liền về một thể hiện thứ hai của một chương trình đang được gỡ rối thì Debugger sẽ tạo hai chốt ngừng con-cái cho mỗi chốt ngừng hiện hữu, Program column sẽ cho thấy mỗi chốt ngừng con-cái nào thuộc thể hiện nào.

  Bạn có thể vô hiệu hóa riêng rẽ các chốt ngừng con-cái, nhưng bạn không thể gỡ bỏ chốt ngừng con-cái, trừ phi bạn gỡ bỏ chốt ngừng cha-mẹ.

- ***Cho hiệu lực hoặc vô hiệu hóa chốt ngừng*:** Một ô duyệt (check box) xuất hiện trên cột Name cạnh bên mỗi chốt ngừng. Tình trạng ô này cho biết liệu xem chốt ngừng có hiệu lực hay vô hiệu hoá. Muốn cho hiệu lực hay vô hiệu hoá một chốt ngừng duy nhất, bạn chọn hoặc xoá ô duyệt nằm cạnh bên chốt ngừng.

- ***Cho đặt để, gỡ bỏ và hiệu đính các chốt ngừng*:** (1) Muốn đặt để một chốt ngừng mới, trên thanh công cụ ở đầu cửa sổ **Breakpoints**, bạn chọn icon **New Breakpoint** để cho hiện lên khung đối thoại **New Breakpoint,** rồi từ khung đối thoại này bạn chọn loại chốt ngừng và những mục chọn bạn muốn. Cuối cùng bạn ấn OK. (2) Còn nếu muốn gỡ bỏ một chốt ngừng, bạn click lên chốt ngừng bạn muốn gỡ bỏ trên cửa sổ **Breakpoints** Window. Rồi trên thanh công cụ ở đầu chóp Window, bạn chọn mục **Delete.** (3) Còn nếu muốn hiệu đình một chốt ngừng, bạn click lên chốt ngừng bạn muốn hiệu đính, rồi chọn mục **Properties** tool, rồi bạn chọn loại chốt ngừng trên khung đối thoại **Breakpoint Properties** cũng như những mục chọn bạn muốn, rồi ấn OK.

- ***Nhìn xem vị trí chốt ngừng*:** Đối với những chốt ngừng được đặt để trên một vị trí tập tin nguồn hoặc vị chỉ ký ức, bạn có thể nhìn xem vị trí chốt ngừng trên tập tin mã nguồn hoặc đoạn mã disassembly. (Một chốt ngừng được đặt để trên một biến, được gọi là một data breakpoint, thì không có vị trí như thế.). (1) Muốn nhìn xem một vị trí chốt ngừng trên mã nguồn, trên cửa sổ **Breakpoints Window**, bạn click lên chốt ngừng bạn muốn nhìn. Trên thanh công cụ nằm trên chóp Window, bạn chọn mục **Go**

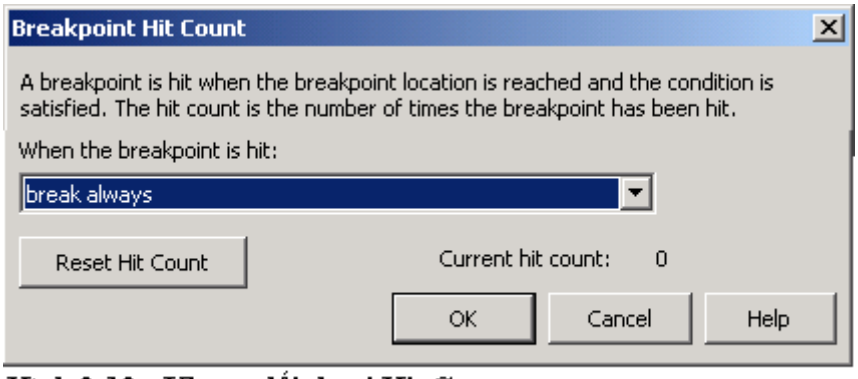
To Source Code tool.. (2) Còn nếu muốn nhìn vị trí của một chốt ngừng trên mã disassembly, bạn làm như điều (1) nhưng chọn mục **Go To Disassembly** tool.

3.3.3 Thuộc tính Hit Count

Hit Count là số lần một chốt ngừng bị đụng. Đối với một location breakpoint (file, address và function), đây là số lần thi hành đạt đến vị trí được chỉ định và thoả điều kiện ngắt, nếu một điều kiện được đặt để. Đối với một data breakpoint, đây là số lần trị của biến bị thay đổi. Thuộc tính **Hit Count** xác định chốt ngừng sẽ bị đụng bao nhiêu lần trước khi thi hành bị ngắt. Theo mặc nhiên, Hit Count = 1. Nếu bạn khai báo thuộc tính **Hit Count** bạn có thể chọn một trong những điều sau đây:

- Bao giờ cũng ngắt (trị mặc nhiên).
- Ngắt khi hit count bằng một trị được chỉ định.
- Ngắt khi hit count bằng bội số (multiple) của một trị được chỉ định.
- Ngắt khi hit count lớn hơn hoặc bằng một trị được chỉ định.

Bạn có thể khai báo một hit count đối với bất cứ loại chốt ngừng nào: function, file, address, hoặc data. Muốn khai báo hoặc hiệu đính thuộc tính **Hit Count,** bạn phải cho mở khung đối thoại **Hit Count Dialog Box (**Hình 3-12)



Hình 3-12 : Khung đối thoại Hit Count.

Nếu bạn khai báo một thuộc tính **Hit Count** đối với một chốt ngừng, bạn có thể mở khung đối thoại **Hit Count dialog box** (hình 3-12) để nhìn xem hoặc đặt để lại trị hiện hành của hit count vào bất cứ lúc nào khi chương trình đang chạy.

3.3.3.1 Khai báo hoặc thay đổi Hit Count

Muốn khai báo hoặc thay đổi một hit count khi tạo một chốt ngừng mới

- Trên trình đơn **Debug**, bạn chọn mục **Debug | New BreakPoint…** để cho hiện lên khung đối thoại **New BreakPoint**.

- Trên khung đối thoại **New Breakpoint**, bạn click nút **Hit Count**. Khung đối thoại **Hit Count** (hình 3-12) hiện lên.

Muốn khai báo hoặc thay đổi một hit count đối với một chốt ngừng hiện hữu

1. Trên cửa sổ **Breakpoints** Window, bạn chọn chốt ngừng bạn muốn.

2. Trên thanh công cụ nằm trên đầu Window, bạn chọn công cụ **Properties**, hoặc right-click để cho hiện lên trình đơn shortcut, rồi chọn mục **Properties**. Khung đối thoại **Breakpoint Properties** hiện lên.

3. Trên khung đối thoại **Breakpoint Properties**, bạn click nút **Hit Count**.

Muốn khai báo hoặc thay đổi một hit count đối với một chốt ngừng hiện hữu trên một cửa sổ mã nguồn, cửa sổ Disassembly, hoặc cửa sổ Call Stack

1. Bạn right-click trên chốt ngừng; trình đơn shortcut hiện lên, bạn chọn mục **Breakpoint Properties** từ trình đơn này.

2. Khi khung đối thoại **Breakpoint Properties** hiện lên, bạn click nút **Hit Count** (hình 3-12).

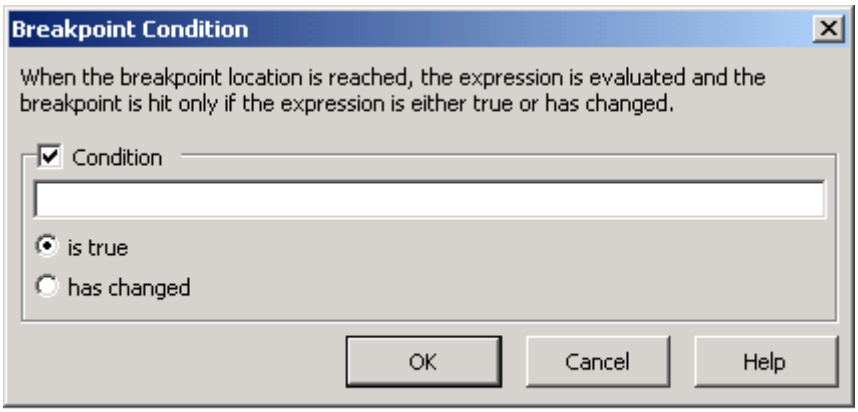
3.3.4 Thuộc tính Condition

Một điều kiện ngắt (break condition) là một biểu thức cần được định trị khi đạt đến chốt ngừng. Kết quả định trị sẽ cho biết chốt ngừng bị đụng hay chưa. Nếu chốt ngừng bị đụng và hit count đúng, thì Debugger cho ngắt thi hành.

Muốn khai báo một điều kiện ngắt, bạn phải cho mở khung đối thoại **Breakpoint Condition** (Hình 3-13). Điều kiện có thể là bất cứ biểu thức hợp lệ nhận diện bởi Debugger.

- Bạn có thể khai báo một điều kiện được thỏa: khi biểu thức là true (**is true** trên khung đối thoại **Breakpoint Condition**), hoặc khi trị của biểu thức thay đổi (**has changed** trên khung đối thoại **Breakpoint Condition**).

Ghi chú Khi lần đầu tiên cho đặt để một chốt ngừng sử dụng **"has changed"**, biểu thức chưa có trị. Khi đạt đến chốt ngừng, biểu thức được định trị và được trữ. Debugger sẽ không xem việc định trị và cất trữ này là một thay đổi, do đó Debugger sẽ không ngừng lần đầu tiên khi chốt ngừng được đạt đến.



Hình 3-13 : Khung đối thoại Breakpoint Condition

3.3.4.1 Khai báo hoặc thay đổi điều kiện chốt ngừng

Bạn có thể khai báo hoặc thay đổi điều kiện chốt ngừng bằng cách dùng một trong những thủ tục sau đây:

Muốn khai báo hoặc thay đổi một điều kiện chốt ngừng khi đang tạo một chốt ngừng mới.

1. Trên cửa sổ **Breakpoints** Window (hình 3-11), bạn chọn breakpoint nào bạn muốn.

2. Trên thanh công cụ nằm trên đầu Window, bạn click công cụ **Properties** để cho hiện lên khung đối thoại **New Breakpoint** hiện lên.

3. Trên khung đối thoại **New Breakpoint**, bạn click nút **Condition** (hình 3-13).

Muốn khai báo hoặc thay đổi một điều kiện chốt ngừng khi tạo một chốt ngừng mới từ một cửa sổ mã nguồn, từ cửa sổ Disassembly, hoặc từ Call Stack Window

1. Bạn right-click ngay chỗ chốt ngừng rồi chọn **Breakpoint Properties** từ trình đơn shortcut.

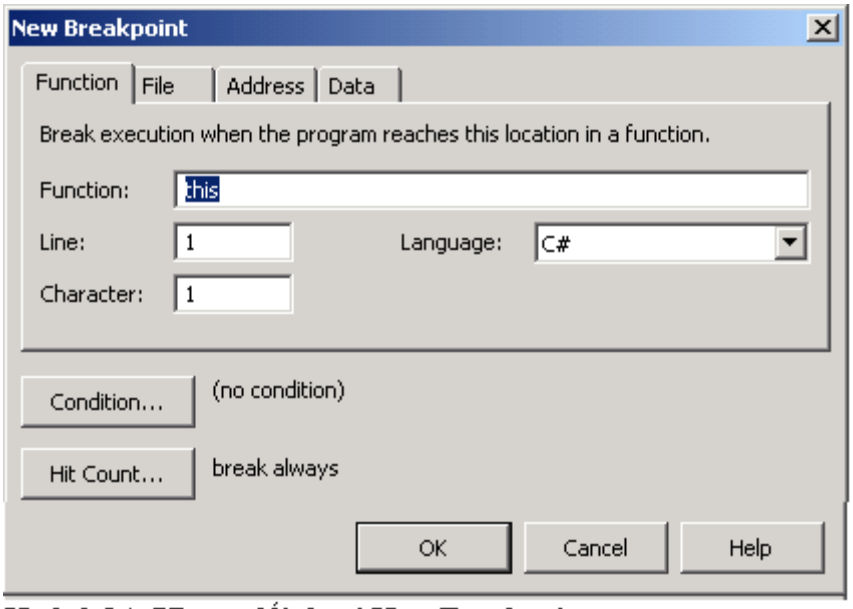
2. Trên khung đối thoại **Breakpoint Properties,** bạn click nút **Condition** (hình 3-13).

3.3.5 Chèn một chốt ngừng mới từ Debug

Không cần biết cửa sổ nào bạn đang dùng, bạn có thể chèn một chốt ngừng mới bất cứ loại nào bằng cách chọn mục **New Breakpoint…** từ trình đơn **Debug**.

Muốn chèn một chốt ngừng mới

1. Trên trình đơn **Debug**, bạn click mục **New Breakpoint…**. hoặc ấn tổ hợp phím <Ctrl + B>. Khung đối thoại **New Breakpoint** hiện lên (hình 3-14).



Hình 3-14: Khung đối thoại New Breakpoint.

2. Bbreakpoint nào bạn muốn đặt để. (Data breakpoint chỉ có sẵn đối với đoạn mã nguyên sinh (native code). Bạn có thể đặt để một data breakpoint trên một biến toàn cục hoặc trên một biến cục bộ trên đầu phạm vi hàm).

3. Cho đặt để các thuộc tính trên trang đã chọn.

4. Bạn click các nút **Condition** hoặc **Hit Count**, nếu bạn một thêm cả hai thuộc tính kể trên. Cuối cùng bạn click **OK.**

Ghi chú Nếu bạn chuyển các trang tab bất cứ lúc nào trước khi click OK, thì breakpoint sẽ được đặt để dựa trên những đặt để của tab được chọn chót nhất.

3.3.6 Gỡ bỏ tất cả các chốt ngừng

Muốn gỡ bỏ tất cả các chốt ngừng trên dư án

- Trên trình đơn **Debug**, bạn click mục **Clear All Breakpoints**.

Tất cả các ký hiệu chốt ngừng sẽ biến mất trên dự án cũng như trên cửa sổ Breakpoints Window.

3.3.7 Các tác vụ chốt ngừng trên cửa sổ mã nguồn

Trên một cửa sổ mã nguồn, bạn có thể nhìn xem, đặt để, cho hiệu lực, vô hiệu hóa, gỡ bỏ hoặc hiệu đính một file, function, hoặc address breakpoint.

Các mục sau đây mô tả những tác vụ chốt ngừng mà bạn có thể thực hiện từ cửa sổ mã nguồn:

- Chèn một file breakpoint trên một cửa sổ mã nguồn
- Chèn một function breakpoint trên một cửa sổ mã nguồn
- Chèn một address breakpoint trên một cửa sổ Disassembly
- Gỡ bỏ một chốt ngừng trên một cửa sổ mã nguồn
- Cho hiệu lực hoặc vô hiệu hóa một chốt ngừng trên một cửa sổ mã nguồn
- Hiệu đính một chốt ngừng trên một cửa sổ mã nguồn

Muốn chèn một file breakpoint trên một cửa sổ mã nguồn

- Trên cửa sổ mã nguồn, bạn click trên biên màu xám kế bên hàng lệnh nơi mà bạn muốn cho đặt để breakpoint. Một ký hiệu vòng tròn đỏ hiện lên.

Ghi chú: Nếu hàng này không phải là một lệnh khả thi thì tự động Debugger nhảy về hàng kế tiếp chứa lệnh khả thi để đặt chốt ngừng.

Gỡ bỏ một chốt ngừng trên cửa sổ mã nguồn

Trên một cửa sổ mã nguồn, bạn có thể nhìn xem file, function hoặc address breakpoint như là những ký hiệu phía biên trái. Bạn có thể gỡ bỏ những chốt ngừng này sử dụng thủ tục dưới đây:

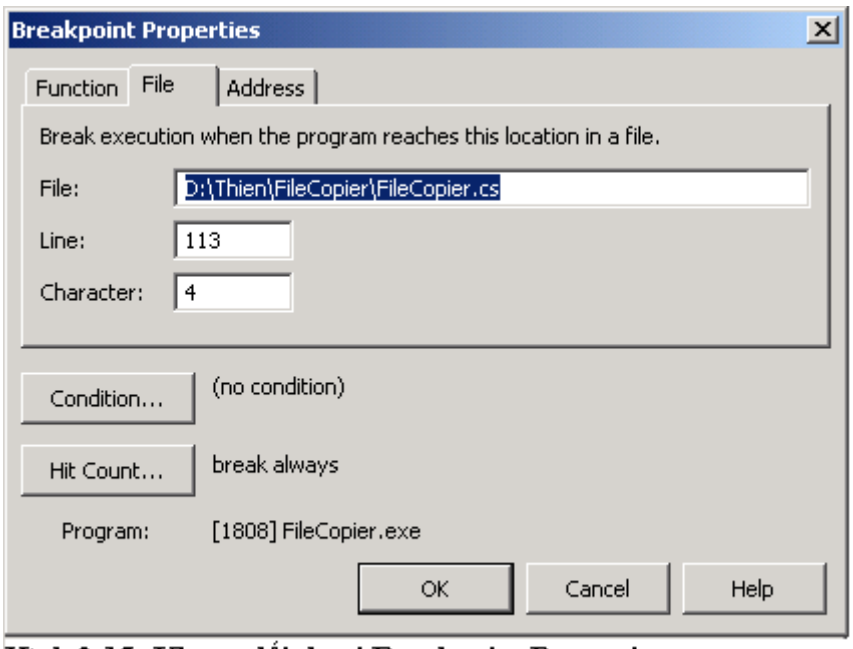
- Trên cửa sổ mã nguồn, bạn click lên ký hiệu chốt ngừng (vòng tròn màu đỏ) trên vùng biên màu xám.

Bạn không thể nhìn data breakpoint trên một cửa sổ mã nguồn, do đó bạn không thể gỡ bỏ một data breakpoint từ một cửa sổ mã nguồn, bạn phải dùng đến cửa sổ **Breakpoints**.

Hiệu đính một chốt ngừng trên cửa sổ mã nguồn

Trên cửa sổ mã nguồn, bạn có thể nhìn xem file, function hoặc address breakpoint dựa theo những ký hiệu trên biên trái màu xám. Bạn có thể hiệu đính những chốt ngừng này sử dụng thủ tục dưới đây:

1. Bạn right-click lên hàng chứa ký hiệu của breakpoint (một chấm đậm, một hình tròn rỗng, tùy thuộc tình trạng của chốt ngừng). Từ trình đơn shortcut, bạn chọn mục **Breakpoint Properties**. Khung đối thoại **Breakpoint Properties** hiện lên. Hình 3-15.



Hình 3-15: Khung đối thoại Breakpoint Properties.

Chú ý: Nếu chốt ngừng được đặt để trên một ký tự đặc biệt, bạn phải right-click đúng ký tự nơi mà chốt ngừng được đặt để.

2. Trên khung đối thoại **Breakpoint Properties**, bạn cho hiệu đính những thuộc tính của chốt ngừng mà bạn muốn thay đổi.

Bạn thấy khung đối thoại này chỉ có 3 trang tab, không có Tab Data, nghĩa là bạn không thể nhìn data breakpoint trên một cửa sổ mã nguồn, do đó bạn không thể hiệu đính một data breakpoint từ một cửa sổ mã nguồn, bạn phải dùng đến cửa sổ **Breakpoints Window**.

Vô hiệu hoá một chốt ngừng trên cửa sổ mã nguồn.

Trên cửa sổ mã nguồn, bạn có thể nhìn xem file, function hoặc address breakpoint dựa trên những ký hiệu nằm ở biên trái màu xám. Bạn có thể cho hiệu lực hoặc vô hiệu hóa một chốt ngừng trên một cửa sổ mã nguồn theo thủ tục dưới đây:

1. Bạn right-click hàng có chứa một ký hiệu của chốt ngừng có hiệu lực (một chấm tròn đỏ đầy) để cho trình đơn shortcut hiện lên.

2. Từ trình đơn shortcut, bạn chọn mục **Disable Breakpoint**. Ký hiệu của chốt ngừng biến thành một vòng tròn rỗng.

Một khi bạn đã vô hiệu hóa một chốt ngừng, bạn có thể cho hiệu lực lại cũng theo thủ tục tương tự.

Cho hiệu lực một chốt ngừng trên cửa sổ mã nguồn

1. Bạn right-click hàng có chứa một ký hiệu chốt ngừng bị vô hiệu hóa (một hình tròn rỗng). Trình đơn shortcut hiện lên.

2. Từ trình đơn shortcut, bạn chọn mục **Enable Breakpoint**. Ký hiệu chốt ngừng biến thành dấu chấm đậm.

3.3.8 Các tác vụ chốt ngừng trên cửa sổ Disassembly

Trên cửa sổ **Disassembly**, bạn có thể view, set, enable, disable, delete, hoặc edit một file, function, hoặc address breakpoint. Bạn có thể chèn một data breakpoint, nhưng data

breakpoints lại không xuất hiện trên cửa sổ **Disassembly**. Một khi bạn đã tạo ra một data breakpoint, bạn không thể view, set, enable, disable, delete, hoặc edit nó từ cửa sổ **Disassembly**. Muốn thế bạn nên sử dụng cửa sổ **Breakpoints**. Sau đây là những đề mục mô tả những tác vụ chốt ngừng mà bạn có thể thực hiện từ cửa sổ **Disassembly**:

- Chèn một Address Breakpoint trên cửa sổ **Disassembly**
- Cho hiệu lực hoặc vô hiệu hóa một chốt ngừng trên cửa sổ **Disassembly**
- Gỡ bỏ một chốt ngừng trên cửa sổ **Disassembly**
- Hiệu đính một chốt ngừng trên cửa sổ **Disassembly**

Chèn một Address Breakpoint trên cửa sổ Disassembly

- Trên cửa sổ **Disassembly**, bạn click trên vùng biên màu xám kế cận hàng lệnh mà bạn muốn đặt để một chốt ngừng. Ký hiệu chốt ngừng hiện lên.

Gỡ bỏ một chốt ngừng trên cửa sổ Disassembly

Trên cửa sổ **Disassembly**, bạn có thể nhìn xem file, function, hoặc address breakpoints dựa trên các ký hiệu nằm ở biên phía tay trái. Bạn có thể gỡ bỏ những chốt ngừng này sử dụng thủ tục sau đây.

- Trên cửa sổ **Disassembly**, bạn click lên ký hiệu chốt ngừng trên biên trái màu xám.

Cho hiệu lực hoặc vô hiệu hoá một chốt ngừng trên cửa sổ Disassembly

Bạn có thể cho có hiệu lực hoặc vô hiệu hóa những chốt ngừng này bằng cách dùng các thủ tục ghi dưới đây.

1. Bạn right-click trên hàng chứa một chốt ngừng có hiệu lực (hoặc bị vô hiệu hoá).
2. Từ trình đơn shortcut, bạn chọn mục **Disable Breakpoint** (hoặc **Enable Breakpoint**). Ký hiệu chốt ngừng sẽ đảo ngược.

Một khi bạn đã vô hiệu hóa một chốt ngừng, bạn có thể cho hiệu lực lại chốt ngừng theo thủ tục tương tự:

Hiệu đính một chốt ngừng trên cửa sổ Disassembly

Bạn có thể hiệu đính những thuộc tính của các chốt ngừng này sử dụng thủ tục dưới đây:

1. Bạn right-click trên hàng chứa ký hiệu chốt ngừng (một dấu tròn đậm đặt hoặc dấu tròn rỗng tùy theo tình trạng chốt ngừng).

2. Từ trình đơn shortcut, bạn chọn mục **Breakpoint Properties**. khung đối thoại **Breakpoint Properties** hiện lên. (Hình 3-15)

3. Trên khung đối thoại **Breakpoint Properties,** bạn hiệu đính những thuộc tính chốt ngừng nào đó bạn muốn. Rồi bạn click <OK>.

3.3.9 Các tác vụ chốt ngừng trên cửa sổ Call Stack

Các đề mục sau đây mô tả các tác vụ chốt ngừng mà bạn có thể thực hiện từ cửa sổ **Call Stack:**

- Cho đặt để một chốt ngừng trên một triệu gọi hàm.
- Gỡ bỏ một chốt ngừng trên cửa sổ Call Stack
- Cho hiệu lực hoặc vô hiệu hóa một chốt ngừng trên cửa sổ **Call Stack**
- Hiệu đính một chốt ngừng trên cửa sổ **Call Stack**

Cho đặt để một chốt ngừng trên một triệu gọi hàm trên Call Stack

Trên cửa sổ **Call Stack**, bạn có thể đặt để một chốt ngừng tại một triệu gọi hàm nào đó trên call stack theo thủ tục sau đây. Chốt ngừng được đặt để ở chỉ thị khả thi kế tiếp trên triệu gọi hàm.

- Trên cửa sổ **Call Stack**, bạn right-click triệu gọi hàm rồi chọn mục **Insert Breakpoint** từ trình đơn shortcut. (Debugger phải ở trong chế độ break mode). Một dấu chấm đỏ hiện lên cạnh triệu gọi hàm.

Khi bạn nhìn xem các thuộc tính chốt ngừng, chốt ngừng này xem như là một address breakpoint với một vị chỉ ký ức tương ứng với chỉ thị khả thi kế tiếp trên hàm.

Gỡ bỏ một chốt ngừng trên Call Stack Window

Bạn có thể gỡ bỏ chốt ngừng theo thủ tục sau đây:

1. Bạn right-click khung chứa ký hiệu chốt ngừng (một chấm đậm đặc hoặc một chấm tròn rỗng, tùy tình trạng chốt ngừng).

2. Từ trình đơn shortcut, bạn chọn mục **Remove Breakpoint**.

Muốn cho có hiệu lực hoặc vô hiệu hoá một chốt ngừng trên Call Stack Window

Bạn có thể cho hiệu lực hoặc vô hiệu hóa một chốt ngừng theo thủ tục sau đây:

1. Bạn right-click khung chứa một chốt ngừng có hiệu lực (hoặc đã bị vô hiệu hoá).

2. Từ trình đơn shortcut, bạn chọn mục **Disable Breakpoint** (hoặc **Enable Breakpoint**). Ký hiệu sẽ đảo ngược.

Một khi bạn đã vô hiệu hóa một chốt ngừng, bạn có thể cho hiệu lực lại một chốt ngừng theo thủ tục tương tự

Hiệu đính một chốt ngừng trên Call Stack Window

Bạn có thể hiệu đính những thuộc tính của một chốt ngừng theo thủ tục dưới đây:

1. Bạn right-click khung chứa ký hiệu chốt ngừng (một dấu chấm đậm đặc hoặc dấu tròn rỗng, tùy theo tình trạng của chốt ngừng). Từ trình đơn shortcut, bạn chọn mục **Breakpoint Properties**. Khung đối thoại **Breakpoint Properties** xuất hiện.

2. Trên khung đối thoại **Breakpoint Properties**, bạn cho hiệu đính các thuộc tính chốt ngừng mà bạn muốn thay đổi, rồi click **OK**.

Đọc tới đây, không biết đầu óc bạn có ê ẩm hay không. Nhưng người viết thì có đấy. Theo chúng tôi nghỉ, bạn nên lấy một thí dụ thực tế, rồi bắt đầu từ đấy mà thực hành, theo từng giai đoạn một, rồi rút kinh nghiệm. Khi sử dụng đến cửa sổ nào thì bạn giở chương này ra đọc lại để thực hành.

Chương 4

Căn bản Ngôn ngữ C#

Chương 2 đi trước đã biểu diễn cho bạn xem một chương trình C# đơn giản. Tuy nhiên, cũng khá phức tạp cho dù là một chương trình cỏn con, với một số chi tiết đã bị bỏ qua. Chương này sẽ dẫn dắt bạn vào sâu những chi tiết thông qua cú pháp và cấu trúc của bản thân ngôn ngữ C#.

Chương này sẽ đề cập đến kiểu dữ liệu hệ thống (type system) trên C#, đưa ra một sự phân biệt giữa những kiểu dữ liệu "bẩm sinh" (built-in), như **int, bool** chẳng hạn, với kiểu dữ liệu tự tạo bởi người sử dụng, như lớp và giao diện chẳng hạn. Chương này cũng bao quát những điều cơ bản về lập trình, chẳng hạn làm thế nào tạo và sử dụng những biến (variable) và hằng (constant). Tiếp theo là những mục về liệt kê (enumeration), chuỗi (string), identifier (diện từ), biểu thức (expression) và câu lệnh (statement).

Phần hai của chương sẽ giải thích và minh hoạ cách sử dụng những cấu trúc điều khiển như **if, switch, while, do..while, for** và **foreach**. Ngoài ra, cũng sẽ đề cập đến các tác tử (operator) bao gồm các tác tử gán (assignment), lô gic, quan hệ (relational) và toán số học. Tiếp theo là dẫn nhập vào namespace và phần giáo đầu về C# precompiler.

Mặc dù C# chủ yếu liên quan đến việc tạo và thao tác lên những đối tượng (object), tốt hơn là ta bắt đầu với những "khối xây dựng" (building block) căn bản: những phần tử từ đấy các đối tượng được tạo ra. Điểm này bao gồm các kiểu dữ liệu bẩm sinh được xem như là thành phần nội tại (intrinsic) của ngôn ngữ C# cũng như những phần tử cú pháp của C#.

4.1 Kiểu dữ liệu (type)

Kiểu dữ liệu bẩm sinh và tự tạo

C# là một ngôn ngữ được kiểm soát chặt chẽ về mặt kiểu dữ liệu (strongly typed language), nghĩa là bạn phải khai báo kiểu dữ liệu của mỗi đối tượng khi bạn tạo ra (chẳng hạn số nguyên, số thực, chuỗi, window, nút ấn v.v..), và trình biên dịch sẽ giúp bạn ngăn ngừa những lỗi sai cú pháp (gọi là bug), bằng cách "tuần tra" chỉ cho phép dữ liệu đúng kiểu mới được gán cho những đối tượng này. Kiểu dữ liệu của một đối tượng báo cho trình biên dịch biết *kích thước của đối tượng này* (nghĩa là một **int** cho biết là

một đối tượng chiếm 4 bytes trong ký ức) cũng như khả năng của đối tượng (chẳng hạn nút ấn – button - có thể được vẽ ra, bị ấn xuống, v.v..).

Giống như C++ và Java, C# chia kiểu dữ liệu thành hai nhóm: kiểu dữ liệu ***bẩm sinh*** *(built-in)* còn được gọi là kiểu dữ liệu nội tại (intrinsic type) do ngôn ngữ cung cấp sẵn (giống như ta khi vừa sinh ra ta đã có một số hành động bẩm sinh không ai dạy như khóc hét, tìm vú mẹ bú, v.v..) và kiểu dữ liệu ***tư tạo*** (user-defined) bởi người sử dụng.

Dữ liệu kiểu trị và kiểu qui chiếu

Ngoài ra, C# lại chia các kiểu dữ liệu thành 2 loại khác nhau, dựa chủ yếu trên cách các trị được trữ thế nào trên ký ức: ***kiểu trị (value type)*** và ***kiểu qui chiếu (reference type)***. Nghĩa là, trên một chương trình C#, dữ liệu được trữ trên một hoặc hai nơi tùy theo đặc thù của kiểu dữ liệu.

Chỗ thứ nhất là **stack**, một vùng ký ức dành trữ dữ liệu chiều dài cố định, chẳng hạn **int** chiếm dụng 4 bytes ký ức. Mỗi chương trình khi đang thi hành đều được cấp phát riêng một stack riêng biệt mà các chương trình khác không được mó tới. Khi một hàm được triệu gọi hàm thi hành thì tất cả các biến cục bộ của hàm được ấn vào (push) stack, và khi hàm hoàn thành công tác thì những biến cục bộ của hàm đều bi tống ra (pop). Đây là cách thu hồi ký ức đối với những hàm hết hoạt động. Stack hoạt động theo kiểu LIFO (last-in-first-out, vào sau ra trước) và được cấu tạo theo kiểu vòng xoay (cicular). Nếu một lúc nào đó, bạn viết một chương trình đệ quy (recursive) cho ra thông điệp "out of stack space error" (sai lầm vì thiếu ký ức trên stack), có nghĩa là chương trình ấn vào stack quá nhiều biến đến nỗi không còn chỗ chứa.

Chỗ thứ hai là **heap**, một vùng ký ức dùng trữ dữ liệu có bề dài thay đổi và khá đồ sộ, string chẳng hạn, hoặc dữ liệu có một cuộc sống dài hơn hàm hành sự, một đối tượng chẳng hạn. Thí dụ, khi một hàm hành sự hiển lộ (instantiate) một đối tượng, đối tượng này được trữ trên heap, và nó sẽ không bị tống ra khi hàm hoàn thành công tác giống như trên stack, mà ở nguyên tại chỗ và có thể được trao cho các hàm hành sự khác thông qua một qui chiếu. Trên C#, heap này được gọi là **managed heap**, khôn lanh vì heap này có một bộ phận gọi là garbage collector (GC, dịch vụ hốt rác) chuyên lo thu hồi ký ức "lâu ngày" không dùng đến (nghĩa là không qui chiếu đến).

Nói tóm lại, một biến kiểu trị sẽ cho trữ trị hiện thực lên ký ức stack (hoặc nó được cấp phát như là thành phần của một đối tượng kiểu qui chiếu), còn một biến kiểu qui chiếu thì vị chỉ (address) biến nằm trên stack, nhưng trị hiện thực của đối tượng thì lại trữ trên ký ức heap. Nếu bạn có một đối tượng kích thước khá lớn, thì nên cho trữ trên heap.

Vị trí ký ức của một kiểu dữ liệu sẽ cho bạn thấy dữ liệu sẽ ứng xử thế nào trong phạm vi một câu lệnh gán (assign). Gán một biến kiểu trị cho một biến kiểu trị khác sẽ tạo ra hai bản sao riêng biệt cùng nội dung và được trữ lên stack. Ngược lại, gán một biến

qui chiếu cho một biến qui chiếu khác sẽ đưa đến việc hai qui chiếu (nằm trên stack) đều chĩa về cùng một vị trí ký ức chứa nội dung đối tượng (nằm trên heap).

C# cũng hỗ trợ *kiểu con trỏ* (pointer type) giống như trên C++, nhưng ít khi dùng đến, và chỉ dùng khi làm việc với đoạn mã unmanaged. Đoạn mã unmanaged là đoạn mã được tạo ra ngoài sàn diễn .NET, chẳng hạn những đối tượng COM.

Để hiểu thêm những khác biệt giữa dữ liệu kiểu trị và dữ liệu kiểu qui chiếu, đề nghị bạn xem Bảng 4-1 trước những câu hỏi "hóc búa":

Bảng 4-1: Dữ liệu kiểu trị và dữ liệu kiểu qui chiếu cạnh bên nhau.

Các câu hỏi hóc búa	**Dữ liệu kiểu trị**	**Dữ liệu kiểu qui chiếu**
Dữ liệu được cấp phát ký ức ở đâu	Trên ký ức stack	Trên ký ức heap
Biến được biểu diễn thế nào?	Biến kiểu trị là những bản sao cục bộ.	Biến kiểu qui chiếu là con trỏ chĩa về vùng ký ức được cấp phát cho thể hiện lớp (đối tượng).
Lớp cơ bản là gì?	Phải được dẫn xuất trực tiếp từ namespace **System. ValueType**.	Có thể được dẫn xuất từ bất cứ lớp nào (ngoại trừ lớp **System.ValueType**) miễn là lớp này không "vô sinh" (sealed).
Kiểu dữ liệu này có thể hoạt động như là một lớp cơ bản cho các lớp khác được không?	Không. Dữ liệu kiểu trị thuộc lớp "vô sinh" nên không "đẻ" ra các lớp khác, cũng như không thể nới rộng.	Được. Nếu lớp không thuộc loại "vô sinh" thì nó có thể dùng làm lớp cơ bản cho các lớp khác.
Cách hành xử thế nào của thông số được trao qua?	Biến được trao qua theo kiểu trị (nghĩa là một bản sao của biến sẽ được trao qua cho hàm bị triệu gọi)	Biến được trao qua theo qui chiếu (nghĩa là vị chỉ của biến sẽ được trao qua cho hàm bi triệu gọi).
Có khả năng phủ quyết (override) hàm **Object. Finalize()** hay không? Hàm này lo dọn dẹp "rác rưởi"	Không. Dữ liệu kiểu trị không nằm trên heap nên khỏi cần được dọn dẹp thu hồi ký ức.	Có. Nhưng gián tiếp mà thôi.

Ta có thể định nghĩa một hàm constructor cho kiểu dữ liệu này được không ?	Được, nhưng hàm construc tor mặc nhiên sẽ được dành riêng không được mó tới (nghĩa là hàm constructor tự tạo phải có thông số).	Lẽ dĩ nhiên là được!
Khi nào thì các biến kiểu này "ngủm" luôn?	Khi nó ra ngoài phạm vi được định nghĩa.	Khi nào vùng heap được thu gom bởi Garbage Collector.

Nếu một số điều đang còn mù mờ đối với bạn, thì không sao. Về sau bạn sẽ hiểu thôi mà!

4.1.1 Làm việc với kiểu dữ liệu bẩm sinh

C# cung cấp vô số kiểu dữ liệu bẩm sinh (built-in, hoặc intrinsic type) mà ta có thể chờ đợi ở một ngôn ngữ tiên tiến, mỗi kiểu dữ liệu sẽ ánh xạ (mapping) với kiểu nằm ẩn sau, được hỗ trợ bởi .NET Common Language Specification (CLS - Đặc tả Ngôn ngữ Thông dụng) để bảo đảm là những đối tượng được tạo ra trên C# có thể đem sử dụng thay thế lẫn nhau với những đối tượng được tạo ra bởi các ngôn ngữ khác phù hợp với .NET CLS, chẳng hạn VB.NET.

Mỗi kiểu dữ liệu sẽ có một kích thước đặc thù và không thay đổi. Khác với C++, một C# **int** bao giờ cũng chiếm 4 bytes vì nó khớp với **Int32** trên .NET CLS. Bảng 4-2 sau đây liệt kê tất cả các kiểu dữ liệu bẩm sinh mà C# cung cấp của bạn.

Bảng 4-2: Các kiểu dữ liệu bẩm sinh của C#

C# Type	**.NET Framework type**	**Size(in byte)**	
bool	**System.Boolean**	1	**true** hoặc **false**
byte	**System.Byte**	1	Unsigned (trị 0-255)
sbyte	**System.SByte**	1	Signed (trị -128 đến 127)
char	**System.Char**	2	Ký tự Unicode
decimal	**System.Decimal**	8	Tinh xác[28] cố định (fixed precision) lên đến 28 digit và vị trí dấu chấm thập phân. Kiểu này thường được dùng trong tính toán tài chính. Đòi hỏi suffix "m" hoặc "M" (tắt chữ Money).
double	**System.Double**	8	Tinh xác kép (double precision) dấu chấm di động; trữ trị xấp xỉ từ $-\pm 5.0 \times 10^{-324}$ đến $\pm 1.7 \times 10^{308}$ với 15-16 số có ý nghĩa.
float	**System.Single**	4	Số dấu chấm di động. Trữ trị xấp xỉ từ $\pm 1.5 \times 10^{-}$

[28] Precision chúng tôi dịch là "tinh xác", chứ không phải là chính xác, vì chính xác là Exactitude.

			^{45} đến $\pm 3.4 \times 10^{38}$ với 7 số có ý nghĩa.
int	**System.Int32**	4	Số nguyên có dấu (signed). Trị đi từ -2,147,483,648 đến +2,147,483,647
uint	**System.UInt32**	4	Số nguyên không dấu (unsigned). Trị đi từ 0 đến 4,294,967,295
long	**System.Int64**	8	Số nguyên có dấu, đi từ –9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807
ulong	**System.UInt64**	8	Số nguyên không dấu (unsigned). Trị đi từ 0 đến 18,446,744,073,709,551,615
object	**System.Object**		Kiểu dữ liệu **object** dựa trên **System.Object** của .NET Framework. Bạn có thể gán trị thuộc bất cứ kiểu dữ liệu nào lên biến kiểu **object**. Thuộc kiểu qui chiếu.
short	**System.Int16**	2	Số nguyên có dấu (signed). Trị đi từ -32,768 đến 32,767.
ushort	**System.UInt16**	2	Số nguyên không dấu (unsigned). Trị đi từ 0 đến 65,535.
string	**System.String**		Kiểu **string** tượng trưng cho một chuỗi ký tự Unicode. **string** là một alias đối với **System.String** trên .NET Framework. Thuộc kiểu qui chiếu.

Tất cả các kiểu dữ liệu "bẩm sinh" đều thuộc kiểu trị, ngoại trừ **object** và **string.** Còn tất cả các kiểu dữ liệu tự tạo đều thuộc kiểu qui chiếu, ngoại trừ **struct.** Ngoài những kiểu dữ liệu bẩm sinh vừa kể trên, C# còn có những kiểu dữ liệu khác: **enum** (sẽ được đề cập về sau trong chương này) và **struct** (sẽ được đề cập ở chương 8). Ngoài ra, chương 5 cũng sẽ đề cập đến những tế nhị khác đối với dữ liệu kiểu trị, chẳng hạn ép dữ liệu kiểu trị hành động như dữ liệu kiểu qui chiếu thông qua một cơ chế được gọi là "đóng hộp" (boxing), cũng như dữ liệu kiểu trị không mang tính kế thừa.

4.1.1.1 Kiểu dữ liệu số nguyên (integer type)

Theo bảng 4-2 kể trên, C# hỗ trợ 8 kiểu dữ liệu số nguyên: **sbyte, short, int, long, byte, ushort, uint** và **ulong. Bạn** thấy là C# hỗ trợ những số nguyên có dấu (âm dương) hoặc không dấu đi từ 8 bit (1 byte) đến 64 bit (8 byte). Các máy tính trong tương lai sẽ dùng 64 bit. Do đó C# chuẩn bị sẵn vào tương lai.

Một **byte** là kiểu dữ liệu chuẩn 8-bit đối với những trị đi từ 0 đến 255. C# phân biệt rõ ràng **byte** và **char** là hai kiểu dữ liệu khác nhau. Nếu có chuyển đổi (conversion) giữa hai kiểu dữ liệu này thì phải yêu cầu rõ ra (explicit). Bạn để ý, kiểu dữ liệu **byte** theo mặc nhiên là số nguyên không dấu. Phiên bản có dấu là **sbyte** có trị đi từ -128 đến 127.

Với .NET, kiểu dữ liệu **short** giờ đây không còn là "ngắn" nữa; đặc biệt bây giờ nó chiếm 16 bit (2 byte), có trị đi từ -32.768 đến 32.767. Phiên bản không dấu là **ushort** với trị đi từ 0 đến 65,535.

Số nguyên lớn hơn là **int** chiếm 32-bit (4 byte), có trị đi từ -2,147,483,648 đến +2,147,483,647. Phiên bản không dấu là **uint**, với trị đi từ 0 đến 4,294,967,295.

Cuối cùng là kiểu dữ liệu **long** dành cho số lớn nhất, chiếm 64-bit (8 byte), với trị đi từ –9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807. Phiên bản không dấu là **ulong** với trị đi từ 0 đến 18,446,744,073,709,551,615.

Tất cả các biến kiểu số nguyên có thể được gán trị theo thập phân hoặc theo thập lục phân (hexadecimal, gọi tắt là hexa). Nếu ghi theo hexa, thì mang tiền tố **0x**:

```
long x = 0x12ab; // ghi theo hexa
```

Nếu có bất cứ nhập nhằng gì liệu xem một số nguyên là một **int, uint, long** hoặc **ulong**, thì mặc nhiên là **int**. Để khai báo rõ ràng kiểu dữ liệu số nguyên nào, bạn cho ghi nối đuôi các ký tự U (đối với uint), L (đối với long), và UL (đối với ulong) như sau:

```
uint ui = 1234U; // cũng có thể viết uint ui = 1234u;
long l = 1234L;  // cũng có thể viết long l = 1234l;
                 // nhưng rất khó phân biệt chữ "l" với số "1"
ulong ul = 1234UL;
```

4.1.1.2 Kiểu dữ liệu số dấu chấm di động (floating point number)

C# cung cấp hai kiểu dữ liệu floating point number là **float** và **double**. Độ tinh xác của **floa**t là 7 digit (ký số) còn với **double** thì tăng xấp đôi nghĩa là 15 digit. Nếu có bất cứ nhập nhằng gì liệu xem một floating point number là một **float,** hoặc **double**, thì mặc nhiên là **float**. Nếu muốn khai báo trị là một **float** thì bạn cho nối đuôi ký tự F (hoặc f) như sau:

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

4.1.1.3 Kiểu dữ liệu số thập phân (decimal type)

C# cung cấp một kiểu dữ liệu **decimal** tượng trưng cho một dấu chấm di động với độ tinh xác rất cao 28 digit. Đăc biệt là C# dành kiểu dữ liệu này cho giới tài chính, dùng

khai báo số lượng tiền tệ đi từ +/- $1.0x10^{-28}$ đến +/- $7.9x\ 10^{28}$. Muốn khai báo kiểu dữ liệu là decimal, bạn có thể cho ghi nối đuôi ký tự M (hoặc m) như sau:

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

4.1.1.4 Kiểu dữ liệu bool

Kiểu dữ liệu **bool** dùng chứa tri **false** hoặc **true**. Bạn không thể chuyển đổi trị bool thành trị số nguyên hoặc ngược lại. Nếu một biến (hoặc trị trả về của một hàm) được khai báo là **bool**, thì ta chỉ có thể sử dụng **true** hoặc **false**. Ta sẽ nhận thông báo sai lầm khi ta cố sử dụng zero cho false hoặc một số non-zero cho true.

4.1.1.5 Kiểu dữ liệu ký tự

Để trữ trị của một ký tự, C# dùng đến kiểu dữ liệu **char**, chiếm 16-bit (2 byte), chứa ký tự Unicode. Để có thể mã hóa tất cả các ngôn ngữ trên thế giới (chữ Hán chẳng hạn) phải có bộ mã rộng lớn hơn là bộ mã ASCII 8-bit. Do đó, bộ mã Unicode chiếm 2 byte cho mỗi ký tự.

Trực kiện (literal) kiểu char phải được đóng khung trong hai dấu nháy đơn (‘), thí dụ: ‘A’. Nếu bạn cho đóng khung trong hai dấu nháy kép (“) thì trình biên dịch xem như là một chuỗi và sẽ tung ra biệt lệ.

Còn hai kiểu dữ liệu **object** và **string** thuộc kiểu qui chiếu, chúng tôi sẽ đề cập sau.

4.1.1.6 Chọn một kiểu dữ liệu bẩm sinh thế nào?

Điển hình là bạn quyết định chọn kích thước nào dùng cho số nguyên (**short, int** hoặc **long**) dựa trên độ lớn (magnitude) của trị mà bạn muốn trữ. Thí dụ, **ushort** chỉ có thể trữ những trị đi từ 0 đến 65.535, trong khi **ulong** thì có thể trữ trị đi từ 0 đến 4.294.967.295.

Ký ức ngày càng rẻ nhưng thời gian của lập trình viên thì càng mắc, do đó phần lớn thời gian, bạn khai báo biến thuộc kiểu dữ liệu **int,** trừ phi bạn có lý do chính đáng để khai báo khác đi.

Những kiểu dữ liệu có dấu âm dương (signed) thường là kiểu số mà lập trình viên thích chọn sử dụng, ngoại trừ khi họ có lý do sử dụng một trị không dấu âm dương.

Mặc dù có thể bạn khoái dùng một **ushort** (số nguyên không dấu) để tăng gấp đôi trị dương của một **short** có dấu (chuyển trị tối đa từ 32.767 lên 65.535), nhưng xem ra dễ dàng và tiện lợi hơn khi dùng **int,** số nguyên có dấu, với trị tối đa lên đến 2.147.483.647.

Stack và Heap

Một stack là một cấu trúc dữ liệu dùng trữ những mục tin (item) theo kiểu LIFO (last-in-first-out, vào sau ra trước) giống như chồng đĩa trên bếp. **stack** ám chỉ một vùng ký ức hỗ trợ bởi bộ xử lý**,** dùng trữ những biến cục bộ (local variable).

Trên C#, các biến kiểu trị (như số nguyên **int** chẳng hạn) sẽ được cấp phát ký ức trên stack, nghĩa là biến sẽ chiếm dụng một vùng ký ức dành riêng cho biến để chứa trị thực thụ, và vùng ký ức này được mang tên biến.

Còn các biến kiểu qui chiếu (các đối tượng chẳng hạn) thì lại nhận "quyền sử dụng đất" trên một vùng ký ức gọi là heap. Khi một đối tượng được cấp phát ký ức trên heap thì vị chỉ (address) của biến sẽ được trả về, và vị chỉ này được gán cho một biến qui chiếu, và biến qui chiếu này được trữ trên stack.

Để tiết kiệm ký ức, thông thường trình biên dịch sẽ cấp cho mỗi hàm hành sự được triệu gọi một khoảng ký ức heap, được gọi là stack frame, để trữ những biến kiểu trị. Khi hàm hoàn thành công việc rút lui, thì có một dịch vụ gọi là hốt rác (garbage collector, tắt GC) sẽ hủy các đối tượng của hàm bị đánh dấu GC và thu hồi ký ức đã cấp phát.

Còn đối với những đối tượng được trữ trên ký ức heap, thì bộ phận GC sẽ theo định kỳ cho thu hồi ký ức bị chiếm dụng khi thấy đối tượng không còn được "sủng ái" (nghĩa là không còn được qui chiếu).

Tốt hơn là nên dùng một biến không dấu âm dương khi dữ liệu cần biểu diễn không bao giờ âm cả. Thí dụ, bạn muốn dùng một biến để trữ tuổi tác của nhân viên, thì lúc ấy bạn nên dùng một **uint**, một số nguyên không dấu, làm biến vì tuổi không bao giờ âm.

Float, double, và **decimal** cho phép bạn thay đổi mức độ kích thước và độ tinh xác (precision). Đối với những con số nhỏ có số lẻ, thì dùng **float** là tiện nhất. Bạn để ý, trình biên dịch giả định là bất cứ số nào với một dấu chấm thập phân (decimal point) là một **double** trừ khi bạn khai báo khác đi. Muốn gán những trực kiện (literal) **float**, bạn cho theo sau con số một chữ **f**. Chúng tôi sẽ đề cập sau chi tiết về trực kiện:

```
float someFloat = 57f;
```

Kiểu dữ liệu **char** tượng trưng cho một ký tự Unicode. Trực kiện **char** có thể là một ký tự đơn giản, một ký tự Unicode, hoặc một ký tự escape đóng khung trong cặp dấu

nháy đơn (single quote mark). Thí dụ, A là một ký tự đơn giản, trong khi \u0041 là một ký tự Unicode. Còn các ký tự escape là những token gồm hai ký tự theo đấy ký tự đầu là một gạch chéo lui (\, backslash). Thí dụ, **\t** là một dấu canh cột ngang (horizontal tab). Bảng 4-3 cho thấy những ký tự escape thông dụng:

Bảng 4-3: Các ký tự escape thông dụng

Ký tự Escape	Y nghĩa
\’	Single quote - Dấu nháy đơn
\”	Double quote - Dấu nháy kép
\\	Backslash - Gạch chéo lui
\0	Null – Vô nghĩa
\a	Alert – Báo động
\b	Backspace - Nhảy lui một ký tự
\f	Form feed – Sang trang mới
\n	Newline – Sang hàng mới
\r	Carriage Return - Trở về đầu dòng
\t	Horizontal tab – Canh cột ngang
\v	Vertical tab – Canh cột dọc

4.1.1.7 Chuyển đổi các kiểu dữ liệu bẩm sinh

Các đối tượng theo một kiểu dữ liệu nào đó có thể được chuyển đổi (converted) thành một kiểu dữ liệu khác theo hiểu ngầm (implicit) hoặc theo tường minh (explicit). Chuyển đổi ngầm sẽ được thực hiện tự động; trình biên dịch sẽ lo giùm bạn. Chuyển đổi tường minh sẽ xảy ra khi bạn “ép kiểu” (casting) một trị cho về một kiểu dữ liệu khác.

Chuyển đổi ngầm (implicit conversion)

Chuyển đổi ngầm sẽ tự động được thực hiện, và bạn được bảo đảm là không mất thông tin. Thí dụ, bạn có thể “ép kiểu” ngầm từ một số nguyên **short** (2 bytes) qua một số nguyên **int** (4 bytes). Cho dù trị thế nào đi nữa trong **short**, sẽ không bị mất thông tin khi được chuyển đổi qua **int**:

```
short x = 5;
int y = x;    // chuyển đổi ngầm
```

Tuy nhiên, nếu bạn chuyển đổi theo chiều ngược lại, chắc chắn là bạn sẽ mất thông tin. Nếu trị trong biến kiểu **int** lớn hơn 32.767, nó sẽ bị xén (truncated) đi. Trình biên dịch sẽ không chịu thi hành một chuyển đổi ngầm từ **int** qua **short**. Ở đây không có chuyện lấy “ngắn nuôi dài”.

```
short x;
int y = 500;
x = y;    // không chịu biên dịch
```

Bạn phải cho "ép kiểu" một cách rõ ràng ra, nếu bạn chắc cú là không mất thông tin:

```
short x;
int y = 500;
x = (short) y;      // OK, lần này chịu biên dịch
```

Tất cả các kiểu dữ liệu bẩm sinh đều định nghĩa những qui tắc chuyển đổi riêng của kiểu. Chương 5, "Lớp và Đối tượng", sẽ đề cập đến việc định nghĩa những qui tắc chuyển đổi đối với các kiểu dữ liệu tự tạo.

Bảng 4-4 sau đây cho thấy chuyển đổi ngầm mà C# chịu hỗ trợ:

Bảng 4-4: Chuyển đổi ngầm mà C# chịu hỗ trợ

Từ kiểu dữ liệu này..	Qua kiểu dữ liệu …
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Bạn để ý là ta chỉ có thể chuyển đổi ngầm từ một kiểu dữ liệu số nguyên nhỏ qua một kiểu dữ liệu số nguyên lớn hoặc từ một số nguyên không dấu qua một số nguyên có dấu cùng kích thước. Như vậy, ta không thể chuyển đổi ngầm từ một **uint** qua một **int** hoặc ngược lại vì cả hai không cùng miền giá trị (range) và việc chuyển đổi sẽ gây mất dữ liệu. Ta cũng có thể chuyển đổi từ số nguyên qua số floating point, nhưng qui tắc ở đây có hơi khác một chút. Ta có thể chuyển đổi giữa kiểu dữ liệu cùng kích thước chẳng hạn **int/uint** qua **float** (4 byte) và **ulong/long** qua **double** (8 byte), và ta có thể chuyển đổi ngược từ **ulong/long** qua **float**, nhưng việc này có thể gây ra mất 4 byte dữ liệu, có nghĩa là trị của **float** mà ta nhận được sẽ bớt chính xác hơn so với việc dùng một **double**.

Chuyển đổi tường minh (explicit conversion)

Tuy nhiên, vẫn có những chuyển đổi ngầm không thể thực hiện được và trình biên dịch sẽ thông báo sai lầm nếu bạn cố tình vi phạm. Sau đây là một số chuyển đổi ngầm không thể thực hiện được:

- **int** qua **short** - có thể mất dữ liệu
- **int** qua **uint** - có thể mất dữ liệu
- **uint** qua **int** - có thể mất dữ liệu
- **float** qua **int** - sẽ mất dữ liệu sau dấu thập phân
- bất cứ kiểu dữ liệu số qua **char** - sẽ mất dữ liệu
- **decima**l qua bất cứ kiểu dữ liệu số - vì kiểu dữ liệu **decimal** được cấu trúc khác đi so với số nguyên và số di động

Tuy nhiên, ta có thể nếu muốn cho chuyển đổi một cách rõ ra bằng cách dùng **cast** (ép kiểu). Khi ta ép một kiểu dữ liệu này chuyển đổi qua kiểu dữ liệu kia, ta cố tình ép trình biên dịch phải tuân theo, thực hiện việc chuyển đổi. Cú pháp như sau:

```
short x;
int y = 500;
x = (short) y;      // OK, số tối đa là 32.767
```

Kiểu dữ liệu bị ép về sẽ nằm trong cặp dấu ngoặc tròn, trước khi bị chuyển đổi.

Việc chuyển đổi tường minh, sử dụng đến cast có thể là nguy hiểm nếu bạn không biết bạn đang làm gì. Thí dụ một ép kiểu từ long qua int có thể gây nguy hiểm nếu trị ban đầu của long lớn hơn trị tối đa của int. Thí dụ bạn cho chạy các lệnh sau đây:

```
long val = 3000000000;
int i = (int) val;     // Cast sai. Trị int tối đa là 2147483647
```

Trong trường hợp này bạn sẽ nhận: -1294967296 , không phải là số bạn chờ đợi. Để bảo đảm, C# cung cấp cho bạn một tác tử **checked** dùng kiểm tra việc ép kiểu có an toàn hay không. Nếu không an toàn thì nó sẽ tung ra một biệt lệ vào lúc chạy:

```
long val = 3000000000;
int i = checked((int) val); // kiểm tra an toàn ép kiểu
```

Sau đây là một thí dụ sử dụng tác tử **checked** trích từ MSDN xuống. Thí dụ khá đơn giản không có chi rắc rối khỏi cần giải thích:

```
// Các biểu thức non-constant được kiểm tra vào lúc chạy
using System;

class OverFlowTest
{
  static short x = 32767;  // Trị tối đa của short
  static short y = 32767;

  // Dùng kiểm tra một biểu thức
  public static int myMethodCh()
```

```
    {
     int z = 0;

     try
     {
      z = checked((short)(x + y));
     }
     catch (System.OverflowException e)
     {
      System.Console.WriteLine(e.ToString());
     }
     return z;  // tung ra một biệt lệ vượt năng OverflowException
    }

    public static void Main()
    {
     Console.WriteLine("Trị kết xuất được kiểm là: {0}",
                                          myMethodCh());
    }
}
```

Khi bạn chạy chương trình, nó sẽ tung ra biệt lệ. Bạn có thể gỡ rối hoặc bỏ ngang:

```
System.OverflowException: An exception of type
System.OverflowException was thrown.  at OverFlowTest.myMethodCh()
Trị kết xuất được kiểm là: 0
```

Chúng tôi sẽ đề cập sau vấn đề biệt lệ (exception), ở Chương 12, "Thụ lý các biệt lệ", với những khối lệnh **try, catch** và **finally** như bạn có thể thấy trong thí dụ trên.

4.2 Biến và Hằng

Một biến (variable) là một vị trí ký ức trữ một trị mang một kiểu dữ liệu gì đó. Trong các thí dụ đi trước, cả hai **x** và **y** là những biến. Biến có thể mang những trị được gán cho nó, và trị này có thể bị thay đổi theo chương trình. Chắc bạn đã biết ta dùng cú pháp C# sau đây để khai báo một biến:

[*modifier*] *datatype identifier*;

với *modifier* là một trong những từ chốt: public, private, protected, v..v.., còn *datatype* là kiểu dữ liệu (int, long, v.v..) và **identifier** là tên biến. Thí dụ một biến mang tên **i** kiểu số nguyên **int** và có thể được truy cập bởi bất cứ hàm nào:

```
public int i;
```

Hàm WriteLine()

.NET Framework cung cấp cho bạn một hàm hành sự rất hữu ích để kết xuất lên màn hình. Chi tiết về hàm này, **System.Console.WriteLine(),** sẽ được làm rõ về sau khi chúng ta đi lần theo các chương đi sau, nhưng điểm cơ bản thì khá đơn giản. Bạn triệu gọi hàm như theo thí dụ 4-3, trao qua một chuỗi chữ bạn muốn cho hiển thị lên màn hình và, tùy chọn, những thông số cần được thay thế. Trong thí dụ sau đây:

```
System.Console.WriteLine("Sau khi gán, myInt: {0}", myInt);
```

chuỗi "**Sau khi gán, myInt:"** được in ra, theo sau là trị trong biến **myInt**. Vị trí của *thông số thay thế (substitution parameter)* **{0}** cho biết trị của biến kết xuất đầu tiên, **myInt**, sẽ in ra ở đâu, trong trường hợp này là cuối chuỗi. Trong các chương đi sau, chúng ta sẽ làm quen nhiều với **WriteLine()**.

Bạn tạo một biến bằng cách khai báo kiểu dữ liệu của nó rồi đặt cho biến một cái tên "cúng cơm". Bạn có thể khởi gán (initialize) biến khi bạn khai báo biến, và bạn có thể gán lại một trị mới cho biến bất cứ lúc nào, thay đổi trị được trữ trong biến. Thí dụ 4-1 màn hình những điều vừa kể trên:

Thí dụ 4-1: Khởi gán và Gán một trị cho biến

```
class Values
(  static void Main()
   {  int myInt = 7;   // tạo và khởi gán biến
      Console.WriteLine("Khởi gán, myInt: {0}", myInt);
      myInt = 5;    // gán lại. Trị củ bị xoá
      Console.WriteLine("Sau khi gán, myInt: {0}", myInt);
   }
}
```

Kết xuất

```
Khởi gán, myInt: 7
Sau khi gán, myInt: 5
```

Trong thí dụ trên, ta khởi gán biến **myInt** về trị 7, cho hiển thị trị này, rồi sau đó gán lại biến với trị 5, và cho hiển thị lại lần nữa.

Trên một câu lệnh đơn độc, bạn có thể khai báo và khởi gán nhiều biến cùng một lúc, với điều kiện *các biến thuộc cùng một kiểu dữ liệu và cùng một loại modifier*. Thí dụ:

```
public static int x = 10, y = 20; // x và y là biến số nguyên
                                  //public và static
```

và viết như sau đây sẽ là sai vì kiểu dữ liệu hoặc modifier khác nhau:

```
public int x = 10, private byte y = 20; // không thể biên dịch được
```

Cuối cùng, nhiều biến cùng một tên không thể được khai báo trong cùng một phạm vi (scope). Thí dụ ta không thể làm như sau:

```
int x = 20;
// một số câu lệnh ở đây
int x = 30;
```

4.2.1 Gán rõ ràng (definite assignment)

C# yêu cầu biến phải được gán một trị gì đó mới được đem đi dùng. Bạn thử trắc nghiệm qui tắc này, bằng cách thay đổi dòng lệnh khởi gán **myInt** trong thí dụ 4-1 thành: **int myInt;** rồi cất trữ chương trình được chỉnh lại như theo thí dụ 4-2:

Thí dụ 4-2: Sử dụng một biến chưa được khởi gán

```
class Values
(   static void Main()
    {   int myInt;
        Console.WriteLine("Không khởi gán, myInt: {0}", myInt);
        myInt = 5;
        Console.WriteLine("Sau khi gán, myInt: {0}", myInt);
    }
}
```

Khi bạn cho biên dịch lại, trình biên dịch C# sẽ ra thông báo sai lầm như sau:

```
3.1.cs(6,55): error CS0165: Use of unassigned local variable 'myInt'
```

Nói tóm lại, chương trình 4-2 là bất hợp lệ, vì không thể dùng một biến chưa được khởi gán trước. Lý do đơn giản là khi bạn khai báo một biến, thì một vùng ký ức nào đó được cấp phát cho biến, nhưng nội dung vùng ký ức này trước đó đã chứa cái gì đó mà ta cho là "rác rưởi". Do đó, phải khởi gán để xoá đi rác rưởi thay thế bởi trị khởi gán.

Như vậy, có nghĩa là chúng ta phải khởi gán tất cả mọi biến trong chương trình ?. Thật ra, không phải thế. Hiện thời, bạn không nhất thiết phải khởi gán một biến, nhưng bạn phải gán một trị cho biến trước khi muốn sử dụng biến này. Đây được gọi là *definite assignment* (nhất quyết gán rõ ra) và C# đòi hỏi như thế. Thí dụ 4-3, cho thấy chương trình đúng đắn:

Thí dụ 4-3: Gán nhưng khởi khởi gán

```
class Values
(  static void Main()
   {  int myInt;       // khai báo nhưng chưa khởi gán
      myInt = 7;       // bây giờ mới gán
      Console.WriteLine("Gán, myInt: {0}", myInt);
      myInt = 5;
      Console.WriteLine("Gán lại, myInt: {0}", myInt);
   }
}
```

4.2.2 Hằng (constant)

Một *hằng (constant)* là một biến nhưng trị không thể thay đổi được suốt thời gian thi hành chương trình. Đôi lúc ta cũng cần có những trị bao giờ cũng bất biến, không thay đổi. Thí dụ, có thể bạn cần hoạt động với các điểm nước đông lạnh và điểm nước sôi tính theo Fahrenheit trong chương trình của bạn thực hiện một thí nghiệm hóa học. Chương trình của bạn sẽ rõ ràng hơn, nếu bạn đặt tên cho những biến trữ các trị này là **FreezingPoint** (điểm đông) và **BoilingPoint** (điểm sôi), nhưng bạn không muốn trị trong các biến này bị thay đổi. Làm thế nào ngăn ngừa việc gán lại. Câu trả lời là sử dụng một hằng (constant).

Trong định nghĩa lớp mà ta sẽ xem sau, người ta thường định nghĩa những vùng mục tin (field) được gọi là *read-only variable*, nghĩa là những biến chỉ được đọc mà thôi, không được hiệu đính. Các biến này được định nghĩa với từ chốt **static** và **readonly**. Trị ban đầu của các vùng mục tin này sẽ được đặt để (set) khi khai báo biến lần đầu tiên hoặc trên hàm khởi dựng (constructor). Còn trong định nghĩa hàm hành sự mà ta cũng sẽ xem sau, có những biến được định nghĩa trong phạm vi hàm thường được gọi là *biến cục bộ (local variable)*. Bạn xem mục 4.2.6.1 đề cập đến field và biến cục bộ.

Hằng khác biệt so với vùng mục tin read-only ở 4 điểm quan trọng sau đây:

- Các biến cục bộ cũng như biến vùng mục tin có thể được khai báo là hằng, với từ chốt **const**.

- Hằng *bắt buộc* phải được gán vào lúc khai báo – nó không thể được khai báo khi ở cấp lớp rồi sau đó được khởi gán trong hàm constructor. Một khi đã được khởi gán thì không thể bị viết đè chồng lên.

- Trị của hằng phải có thể tính toán được vào lúc biên dịch. Do đó, không thể khởi gán một hằng từ một trị của một biến. Nếu muốn làm thế, thì phải sử dụng đến một read-only field.

- Hằng bao giờ cũng static. Tuy nhiên, ta không thể đưa từ chốt **static** vào khi khai báo hằng.

Có 3 lợi điểm khi sử dụng hằng (hoặc biến readonly) trong chương trình của bạn:

- Hằng làm cho việc đọc chương trình dễ dàng hơn, bằng cách thay thế những con số vô cảm bởi những tên mang đầy ý nghĩa hơn.

- Hằng làm cho dễ sửa chương trình hơn. Thí dụ, trong chương trình của bạn, có một tỉ lệ thuế phải tính là 6% (=.06). Nếu bạn sử dụng đến hằng "SaleTax = 0.06", thì khi có thay đổi tỉ lệ thuế, bạn chỉ cần tìm đến SaleTax mà điều chỉnh thay vì đọc toàn bộ chương trình (có thể là dài lê thê) để tìm câu lệnh nào có con số 0.06 để mà sửa. Và đôi khi lộp chộp bỏ quên một chỗ nào đó, thì sẽ gây sai lầm.

- Hằng làm cho việc tránh lỗi dễ dàng hơn. Nếu bạn gán một trị khác cho một hằng đâu đó trong chương trình sau khi bạn đã gán trị cho hằng, thì trình biên dịch sẽ thông báo sai lầm.

Hằng có 3 loại: ***literals*** (*trực kiện*)[29]***, symbolic constant*** *(hằng tượng trưng)* và ***enumeration*** *(liệt kê)*. Trong câu lệnh gán sau đây:

```
x = 32;
```

trị **32** là một hằng trực kiện (literal constant), bao giờ cũng bằng **32**. ***Trực kiện*** có nghĩa là dữ kiện trực tiếp nằm trong chỉ thị, không nằm trong ký ức. Bạn không thể thay đổi trị này.

Còn ***hằng tượng trưng*** (symbolic constant) là gán một cái tên cho một trị hằng. Bạn khai báo symbolic constant bằng cách dùng từ chốt **const,** và theo cú pháp sau đây:

```
const datatype identifier = value;
```

Một hằng phải được khởi gán khi bạn khai báo nó, và một khi được khởi gán thì không thể thay đổi. Thí dụ:

```
const int FreezingPoint = 32;
```

29. Trực kiện (literal) viết tắt từ: "dữ kiện trực tiếp". Từ này do tác giả tự đặt.

Trong dòng lệnh khai báo trên, **32** là một hằng trực kiện, và **FreezingPoint** là một symbolic constant. Thí dụ 4-4 minh hoạ việc sử dụng hằng tượng trưng:

Thí dụ 4-4: Sử dụng hằng tượng trưng

```
class Values
{   static void Main()
    {   const int FreezingPoint = 32; // degrees Farenheit
        const int BoilingPoint = 212;
        System.Console.WriteLine("Độ nước đông lạnh: {0}",
                                                 FreezingPoint);
        System.Console.WriteLine("Độ nước sôi: {0}", BoilingPoint);
        // BoilingPoint = 21;
    }
}
```

Thí dụ 4-4 tạo hai hằng tượng trưng số nguyên: **FreezingPoint** và **Boiling Point. Bạn** để ý, cách viết các hằng tượng trưng sử dụng đến ký hiệu Pascal, nghĩa là viết hoa ở mỗi đầu chữ. Nhưng qui tắc này không bắt buộc bởi ngôn ngữ.

Những hằng tượng trưng này dùng vào cùng mục đích giống như sử dụng các trực kiện 32 và 212. Thay vì những con số vô cảm không gợi lên điều gì, thì những tên như **FreezingPoint** hoặc **BoilingPoint** sẽ cho bạn (nếu bạn biết tiếng Anh) biết ý nghĩa của những hằng tượng trưng này. Nếu bạn chuyển qua sử dụng Celsius, bạn có thể khởi gán lại các hằng này tương ứng cho về 0 và 100, và phần còn lại chương trình hoạt động bình thường.

Để kiểm chứng rằng hằng không thể gán lại được, bạn thử thêm dòng lệnh viết đậm trên thí dụ 4-4 bỏ đi các dấu //, và cho biên dịch lại bạn sẽ gặp ngay thông báo sai lầm như sau:

```
error CS0131: The lefthand side of the assignment must be
a variable, property or indexer.
```

4.2.2.1 Một lớp hằng

Bạn có thể tạo một lớp tiện ích gồm toàn những hằng dùng mô phỏng một loại hằng toàn cục. Vì C# không cho phép bạn định nghĩa những hằng cấp toàn cục, bạn có thể khắc phục điều này bằng cách tạo một lớp gồm toàn là hằng. Thí dụ 4-5 sau đây là một lớp **MyConstants** gồm toàn hằng:

Thí dụ 4-5: Một lớp hằng

```
class MyConstants
{   // một vài hằng cho vui…
    public const int myIntConst = 5;
    public const string myStringConst = "Tôi là kẻ bất tài!";

    // không để cho người sử dụng dùng lớp này, hàm constructor
    // phải khai báo private
    private MyConstants() {}
}
```

Sau đó, muốn sử dụng các hằng này, chẳng hạn bạn viết:

```
Console.WriteLine("myIntConst = {0}\nmyStringConst = {1}",
                MyConstants.myIntConst, MyConstants.myStringConst);
```

Bạn cũng có thể đi đến kết quả trên bằng cách khai báo lớp gồm toàn là hằng là một lớp trừu tượng (abstract class) với từ chốt **abstract**, như sau:

```
abstract class MyConstants
{   // một vài hằng cho vui…
    public const int myIntConst = 5;
    public const string myStringConst = "Tôi là kẻ bất tài!";
}
```

4.2.3 Enumerations

Phương án thay thế hằng là *enumeration (liệt kê)*. Enumeration là một kiểu trị độc đáo, gồm một tập hợp những hằng được đặt tên, được gọi là *enumerator list*. Enumeration là một kiểu dữ liệu số nguyên tự tạo (user-defined integer type). Trong thí dụ 4-4, bạn đã tạo ra 2 hằng có liên hệ với nhau:

```
const int FreezingPoint = 32;
const int BoilingPoint = 212;
```

Có thể bạn cũng muốn thêm một số hằng hữu dụng khác vào danh sách kể trên, chẳng hạn:

```
const int LightJacketWeather = 60;   // áo jacket cho thời tiết ấm
const int SwimmingWeather = 72;  // áo tắm
const int WickedCold = 0;  // áo cho trời lạnh căm
```

Tiến trình thêm như thế có vẻ ngán ngẩm, và không có liên hệ lô gic giữa các hằng khác nhau này. C# cung cấp *enumeration* để giải quyết vấn đề này:

```
enum Temperatures
{   WickedCold = 0,
    FeezingPoint = 32,
    LightJacketWeather = 60,
    SwimmingWeather = 72,
    BoilingPoint = 212
}
```

Mỗi enumeration có một kiểu dữ liệu nằm ở sau, có thể là bất cứ kiểu dữ liệu số (**int, short, long** v.v..) ngoại trừ **char**. Cú pháp định nghĩa một enumeration như sau:

[*attributes*] [*modifiers*] **enum** *identifier* [*:base-type*] **{***enumerator-list***};**

Chúng ta sẽ xét sau *attribute* và *modifier*. Tạm thời, ta tập trung xét đến phần còn lại của khai báo trên. Enumeration bắt đầu với từ chốt **enum** theo sau là tên nhận diện (identifier, diện từ), chẳng hạn:

```
enum Temperatures
```

Base type (kiểu dữ liệu căn bản) là kiểu dữ liệu nằm đằng sau đối với enumeration. Nếu không chọn mục này (thường là như thế), thì kiểu dữ liệu mặc nhiên sẽ là **int**, nhưng bạn tự do chọn kiểu dữ liệu số (như **ushort, long** chẳng hạn), ngoại trừ **char**. Thí dụ, đoạn mã sau đây khai báo một enumeration những số nguyên không dấu, **uint**.

```
enum ServingSizes: uint
{   Small = 1,
    Regular = 2,
    Large = 3
}
```

Bạn để ý là khai báo **enum** kết thúc bởi một danh sách enumerator chứa những hằng gán cho enumeration, phân cách bởi dấu phẩy, tất cả được bao bởi cặp dấu {}. Thí dụ 4-6 viết lại thí dụ 4-4 sử dụng một enumeration:

Thí dụ 4-6: Sử dụng enumeration để đơn giản hoá đoạn mã

```
class Values
{   enum Temperatures
    {   WickedCold = 0,
        FeezingPoint = 32,
        LightJacketWeather = 60,
        SwimmingWeather = 72,
        BoilingPoint = 212
    }
```

```
    static void Main()
    {   System.Console.WriteLine("Độ nước đông lạnh: {0}",
            Temperatures.FreezingPoint);
        System.Console.WriteLine("Độ nước sôi: {0}",
            Temperatures.BoilingPoint);
    }
}
```

Như bạn có thể thấy, một enum phải được "nêu chính danh một cách trọn vẹn" (fully qualified) bởi kiểu dữ liệu enum (nghĩa là **Temperature.WickedCold** chẳng hạn).

Mỗi hằng trên một enumeration tương ứng với một trị số, trong trường hợp này là một **int**. Nếu bạn không đặc biệt đặt để một trị số, thì enumeration bắt đầu từ zero, và hằng đi sau sẽ tăng 1 so với trị đi trước. Thí dụ, nếu bạn tạo một enumeration sau đây:

```
enum SomeValues
{   First,
    Second,
    Third = 20,
    Fourth
}
```

thì trị của **First** sẽ là 0, **Second** sẽ là 1, **Third** sẽ là 20 và **Fourth** sẽ là 21.

Enumeration không nhất thiết phải theo một trật tự tuần tự nào đó, và cũng không bắt buộc kiểu dữ liệu phải là số nguyên như theo mặc nhiên. Thí dụ, bạn định nghĩa một kiểu **enum** liên quan đến nhân viên, kiểu dữ liệu là một **byte** thay vì **int:**

```
enum EmpType: byte
{   Manager =10,
    Grunt = 1,
    Contractor = 100,
    VP = 9
}
```

Một khi đã tạo ra một enum `EmpType`, bạn có thể sử dụng như theo thí dụ 4-7 sau đây. Bạn định nghĩa một lớp `EnumClass` với thông số duy nhất là `EmpType`.

Thí dụ 4-7: Sử dụng enum

```
using System;

// Đây là enum.EmpType tự tạo
enum EmpType: byte
{   Manager = 10,
    Grunt = 1,
    Contractor = 100,
```

```
    VP = 9
}

class EnumClass
{   // Enums dùng làm thông số đối với hàm AskForBonus (đòi thưởng)
    public static void AskForBonus(EmpType e)
    {   switch(e)
        {   case EmpType.Contractor:
            Console.WriteLine("bạn đã có quá nhiều tiền mặt...");
            break;

            case EmpType.Grunt:
            Console.WriteLine("Anh bạn giỡn mặt hả...");
            break;

            case EmpType.Manager:
            Console.WriteLine("Thị trường chứng khoán sao rồi?");
            break;

            case EmpType.VP:
            Console.WriteLine("VERY GOOD, Sir!");
            break;

            default: break;
        }
    }

  public static void Main()
  {EmpType fred;       // tạo đối tượng nhân viên
       fred = EmpType.VP;  // thuộc loại VP (vice president)
       AskForBonus(fred);  // hàm xin tiền thưởng
    }
}
```

Kết xuất

VERY GOOD Sir!

Nói tóm lại, có 3 lợi điểm khi sử dụng enumeration:

- Như đã nói, enumeration làm cho chương trình của bạn dễ bảo trì bằng cách bảo đảm là các biến của bạn chỉ được gán những trị hợp lệ, những trị đã định trước sẵn.

- Enumeration làm cho chương trình của bạn sáng sủa hơn, thay vì là những con số vô cảm, sẽ là những tên rất gợi cảm.

- Enumeration cũng làm cho dễ khỏ lệnh. Visual Studio .NET IDE, thông qua công cụ Intellsense, sẽ cho hiện lên một trình đơn shortcut cho phép bạn chọn một mục trên enumeration mà bạn đã khai báo, như vậy tiết kiệm cho bạn công khỏ và bảo đảm khỏ không sai.

Các lớp C# trên Visual Studio .NET sử dụng khá nhiều enumeration.

4.2.3.1 Lớp cơ bản System.Enum

C# enumeration được dẫn xuất từ lớp `System.Enum`**.** Lớp căn bản này có một số hàm hành sự cho phép bạn truy tìm một enumeration nào đó. Thí dụ, lớp này có hàm static `Enum.GetUnderlyingType()` trả về kiểu dữ liệu nằm đằng sau của một enumeration chỉ định:

```
// Đi tìm kiểu dữ liệu nằm đằng sau; ở đây là System.Byte
Console.WriteLine(Enum.GetUnderlyingType(typeof(EmpType)));
```

Dòng lệnh trên dùng đến tác tử **typeof** để nhận về đối tượng **System.Type** đối với một kiểu dữ liệu.

Một hàm static khác, `Enum.Format()` cho phép bạn tìm ra chuỗi tương ứng với trị của enumeration được khai báo. Thí dụ:

```
// thí dụ này lôi ra chuỗi "Anh là Contractor" cho hiện lên màn hình
EmpType fred;
fred = EmpType.VP;
Console.WriteLine("Anh là {0}", Enum.Format(typeof(EmpType),
                    fred, "G"));
```

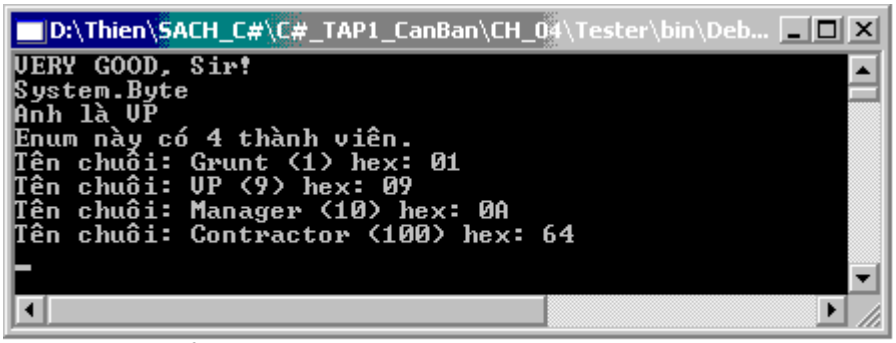
Trên thí dụ trên, "G" đánh dấu là một chuỗi; nếu là trị hexa bạn ghi "x"; nếu là trị thập phân bạn ghi "d".

Một hàm hành sự static khác, `Enum.GetValues()` sẽ trả về một thể hiện của **System.Array**, một bản dãy theo đấy mỗi phần tử tương ứng với một thành viên của enum. Thí dụ:

```
// Lấy thống kê đối với EmpType.
Array obj = Enum.GetValues(typeof(EmpType));
Console.WriteLine("Enum này có {0} thành viên.", obj.Length);

// Bây giờ cho hiển thị tên chuỗi và các trị được gắn liền.
foreach(EmpType e in obj)
{   Console.Write("Tên chuỗi: {0}",
      Enum.Format(typeof(EmpType), e, "G"));
```

```
        Console.Write(" ({0})", Enum.Format(typeof(EmpType), e, "D"));
        Console.Write(" hex: {0}\n", Enum.Format(typeof(EmpType), e,
    }
```

Chúng tôi cho chạy các hàm trên của lớp **System.Enum** trên một ứng dụng console sử dụng Visual Studio .NET IDE, kết quả là hình 4-1.



Hình 4-01: Kết quả sử dụng các hàm của lớp System.Enum

Nói tóm lại lớp **System.Enum** này còn có một số hàm hành sự mà chúng tôi chưa đưa ra ở đây, bạn có thể tham khảo thêm trên MSDN để biết cách sử dụng. Nếu những thí dụ vừa nêu trên có thể đi hơi quá sự hiểu biết của bạn, thì không sao, bạn có thể trở lui lại khi biết cặn kẽ các câu lệnh mà hiện giờ bạn đang mù tịt.

Cuối cùng, enumeration là một kiểu dữ liệu hình thức, do đó đòi hỏi phải có một chuyển đổi tường minh (explicit conversion) để có thể chuyển đổi một enum type qua một integral type.

4.2.4 Các chuỗi chữ

Khó lòng viết một chương trình C# mà không tạo những chuỗi (string). Một đối tượng kiểu **string** thường chứa một chuỗi ký tự. Bạn khai báo một biến chuỗi sử dụng từ chốt **string** giống như bạn tạo một thể hiện của bất cứ đối tượng nào:

```
string myString;
```

Một trực kiện kiểu chuỗi được tạo ra bằng cách bao chuỗi ký tự bởi cặp dấu nháy kép, chẳng hạn: “Xin Chào Bà Con!”.

Thường thì phải khởi gán một biến chuỗi sử dụng đến một trực kiện kiểu string:

```
string myString = “Xin Chào Bà Con!”;
```

Toàn bộ Chương 11, "Chuỗi chữ và biểu thức regular", sẽ dành đề cập đến chuỗi chữ. Bạn sẽ tha hồ mà ngốn.

4.2.5 Các diện từ (identifier)

Identifier (mà chúng tôi tạm dịch là "diện từ" là một từ để nhận diện) là những tên mà lập trình viên chọn đặt cho kiểu dữ liệu tự tạo, hàm hành sự, biến, hằng, đối tượng v.v.. Một identifier phải bắt đầu bởi một chữ (letter) hoặc bởi một dấu gạch dưới (underscore).

Qui ước đặt tên do Microsoft chủ xướng là dùng *ký hiệu lưng lạc đà (camel notation)* đối với tên biến và *ký hiệu Pascal* đối với tên hàm hành sự và phần lớn những identifier khác. Ký hiệu lưng lạc đà là mẫu tự đầu tiên của chữ đầu viết thường, mẫu tự đầu tiên các chữ đi sau viết hoa, thí dụ: **someName**, còn ký hiệu Pascal thì mẫu tự đầu tiên của các chữ đều viết hoa, chẳng hạn **SomeOtherName**.

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

Các diện từ không được đụng độ với các từ chốt (keyword). Do đó, bạn không thể tạo một biến mang tên **int** hoặc **class**. Ngoài ra, các diện từ thường là case-sensitive nghĩa là phân biệt chữ hoa chữ thường. Do đó, **identifier** và **Identifier** là khác nhau. C# có cả thảy 77 từ chốt mà bạn tránh dùng làm diện từ: (xem bảng trên)

Bạn để ý: Microsoft thôi khuyên dùng ký hiệu Hungarian, (chẳng hạn **iSomeInteger**) hoặc dấu gạch dưới (chẳng hạn **SOME_VALUE**).

4.2.6 Phạm vi hoạt động của biến (variable scope)

Trong một phạm vi hoạt động (scope), không thể có hai biến cùng mang một tên trùng nhau. Người ta định nghĩa scope của một biến là vùng đoạn mã từ đấy biến có thể được truy xuất. Ta thử xem thí dụ 4-8 đoạn mã sau đây:

Thí dụ 4-8: Phạm vi hoạt động của biến

```
using System;

public class ScopeTest
{
    public static int Main()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }  // i ra khỏi phạm vi hoạt động ở đây

        // Ta có thể khai báo một biến i một lần nữa, vì không có
        // biến nào mang tên này trong phạm vi
        for (int i = 9; i >= 0; i--)
        {
            Console.WriteLine(i);
        }  // i ra khỏi phạm vi hoạt động ở đây
        return 0;
    }
}
```

Đoạn mã trên đơn giản in ra các con số đi từ 0 đến 9, rồi lộn ngược lại từ 9 đến 0, sử dụng vòng lặp **for**. Chúng tôi sẽ đề cập sau loại vòng lặp này. Điều quan trọng ở đây là chúng tôi khai báo biến **i** hai lần trong cùng một hàm hành sự **ScopeTest.** Lý do vì sao chúng tôi có thể làm được như vậy là vì cả hai lần i được khai báo trong lòng vòng lặp, nghĩa là biến **i** là biến cục bộ (local variable) đối với vòng lặp. Một khi vòng lặp hoàn thành nhiệm vụ thì biến thoát khỏi phạm vi, và không thể truy xuất được nữa.

Ta thử xem một thí dụ 4-9 khác:

Thí dụ 4-9: Trắc nghiệm phạm vi hoạt động của biến

```
using System;

public class ScopeTest
{
    public static int Main()
    {
        int j = 20;
        for (int i = 0; i < 10; i++)
        {
            int j = 30;   // không thể làm như thế - j đang còn trong
                          // phạm vi
            Console.WriteLine(j = i);
        }
        return 0;
```

```
    }
}
```

Nếu bạn cố biên dịch thí dụ trên, bạn sẽ nhận thông điệp sai lầm như sau:

```
ScopeTest.cs(10,14): error CS0136: A local variable named 'j' cannot
be declared in this scope because it would give a different meaning
to 'j', which is already used in a 'parent or current' scope to
denote something else.
```

Lý do sai lầm là vì biến **j** đã được khai báo ở đầu vòng lặp, vẫn đang còn trong phạm vi trong lòng vòng lặp **for** và chỉ thoát khỏi phạm vi khi hàm **Main()** chấm dứt thi hành. Mặc dù **j** thứ nhì (bất hợp lệ) nằm trong phạm vi khác, nhưng phạm vi này lại nằm lòng trong phạm vi của **Main().** Trình biên dịch không có cách chi nhận diện giữa hai biến j, do đó không cho phép biến thứ hai khai báo.

4.2.6.1 Vùng mục tin và Biến cục bộ

C# phân biệt rõ giữa những biến được khai báo ở cấp lớp (class-level variable, nghĩa là trong phần khai báo lớp) mang tên là ***vùng mục tin (field[30])***, và biến được khai báo trong lòng các hàm hành sự được biết đến như là ***biến cục bộ*** (local variable). Bạn thử xem đoạn mã sau đây:

Thí dụ 4-10: Vùng mục tin và biến cục bộ

```
using System;

public class ScopeTest
{
    static int j = 20;  // j là vùng mục tin cấp lớp ScopeTest

    public static int Main()
    {
        int j = 30;  // j là biến cục bộ
        Console.WriteLine(j);
        return 0;
    }
}
```

Nếu cho biên dịch, sẽ không có vấn đề gì, mặc dù ta có hai biến cùng mang tên **j.** Biến j được khai báo trong hàm **Main()** là biến cục bộ, sẽ được trữ trong stack, còn biến j static được khai báo trong lớp **ScopeTest** là vùng mục tin sẽ được trữ trong heap vì được khai báo trong cấp lớp nên sẽ thoát khỏi phạm vi khi nào lớp bị hủy. Như vậy hai biến

[30] Có người dịch là "trường" giống như trường đua ngựa vậy.

này mặc dù trùng tên nhưng được trữ ở hai nơi khác nhau nên không đụng độ nhau. Khi bạn cho chạy đoạn mã này, thì biến j trong hàm **Main()** sẽ che lấp vùng mục tin j và in ra trị 30.

Bây giờ, nếu ta muốn truy xuất vùng mục tin j thì ta làm thế nào? Vì vùng mục tin j trong thí dụ được khai báo là static, ta khỏi phải hiển lộ một thể hiện lớp, nên ta có thể truy xuất vùng mục tin j như sau:

```
Console.WriteLine(ScopeTest.j);  // sẽ in ra 20
```

4.3 Biểu thức (expression)

Các câu lệnh (statement) cho ra một trị được gọi là *biểu thức*. Có thể bạn sẽ ngạc nhiên cần đến bao nhiêu câu lệnh để tính cho ra một trị. Thí dụ, một lệnh gán như sau:

```
myVariable = 57;
```

là một biểu thức; nó tính ra trị được gán, ở đây là 57.

Bạn để ý câu lệnh trên gán trị 57 cho biến **myVariable**. Tác tử gán (=) không trắc nghiệm bằng, mà làm mỗi một việc là lấy gì ở vế phải (57) gán cho những gì ở vế trái (**myVariable**). Chúng tôi sẽ đề cập sau các tác tử (operator) C#.

Vì **myVariable = 57;** là một biểu thức tính ra trị 57, nó có thể đem ra dùng như là thành phần của một tác tử gán khác, chẳng hạn:

```
mySecondVariable = myVariable = 57;
```

Việc xảy ra là trên câu lệnh trên, trị trực kiện 57 được gán cho biến **myVariable**. Trị của việc gán này (57) sau đó lại đem gán cho biến thứ hai **mySecondVariable**. Như vậy, trị 57 được gán cho cả hai biến. Do đó, bạn có thể khởi gán cùng một trị cho bất cứ bao nhiêu biến cũng được chỉ trong một câu lệnh:

```
a = b = c = d = e = 20;
```

4.4 Whitespace

Trên ngôn ngữ C#, các ký tự trắng (space), khoảng trắng canh cột (tab) và sang hàng mới (newline) đều được gọi là whitespace (khoảng trắng). C# phớt lờ những whitespace như thế trong các câu lệnh. Do đó, bạn có thể viết:

```
myVariable = 5;               hoặc
myVariable   =         5;
```

thì trình biên dịch đều xem 2 câu lệnh trên tương tự như nhau. Tuy nhiên, ngoại lệ đối với qui tắc trên là khoảng trắng trong lòng chuỗi chữ không thể bỏ qua. Nếu bạn viết:

```
Console.WriteLine("Xin Chào Bà Con");
```

thì mỗi ký tự trắng nằm giữa "Xin", "Chào", "Bà" và "Con" đều phải tính vào như là một ký tự khác trong chuỗi chữ.

Phần lớn việc sử dụng whitespace trong chương trình mang tính trực giác. Điểm chủ yếu khi sử dụng whitespace là làm thế nào chương trình sáng sủa dễ đọc đối với lập trình viên; còn trình biên dịch thì bất cần.

Tuy nhiên, đôi lúc việc sử dụng whitespace cũng mang ý nghĩa. Mặc dù biểu thức:

```
int x = 5;    // có khoảng trắng ở hai đầu tác tử gán
```

cũng tương tự như:

```
int x=5;  // không khoảng trắng ở hai đầu tác tử gán
```

nhưng lại không giống như:

```
intx=5;
```

Trình biên dịch biết rõ whitespace ở hai đầu tác tử gán là dư (extra), nhưng khoảng trắng giữa **int** và tên biến **x** là bắt buộc. Trình biên dịch có thể phân tích câu lệnh nhờ khoảng trắng để nhận ra từ chốt **int** thay vì một từ vô danh **intx** nào đó. Bạn hoàn toàn tự do thêm ít nhiều whitespace tùy ý giữa **int** và **x**, nhưng ít nhất phải 1 ký tự trắng hoặc một tab.

4.5 Các câu lệnh (statements)

Trên C# một chỉ thị (instruction) chương trình trọn vẹn được gọi là một câu lệnh. Chương trình thường gồm một loạt câu lệnh C# theo một thứ tự nào đó. Mỗi câu lệnh phải được *kết thúc bởi dấu chấm phẩy* (;). Thí dụ:

```
int x;        // một câu lệnh
x = 23;       // một câu lệnh khác
int y = x;    // lại một câu lệnh khác
```

Các câu lệnh C# được định trị (evaluate) theo thứ tự vào. Trình biên dịch bắt đầu từ đầu danh sách các câu lệnh cho đến cuối, hoàn toàn "thẳng tuột"và bị giới hạn khủng khiếp nếu không có rẻ nhánh (branching). Có hai loại rẻ nhánh trên một chương trình C#: ***rẻ nhánh vô điều kiện*** *(unconditional branching)* và ***rẻ nhánh có điều kiện*** *(conditional branching)*.

Dòng chảy chương trình (program flow) cũng bị ảnh hưởng bởi các câu lệnh vòng lặp (loop) và rảo lặp (iteration) qua các từ chốt **for, while, do, in** và **foreach.** Vào cuối chương, chúng tôi sẽ đề cập đến rảo lặp. Tạm thời, chúng ta xét đến những hàm hành sự cơ bản về rẻ nhánh vô điều kiện và có điều kiện.

4.5.1 Các câu lệnh rẽ nhánh vô điều kiện

Rẽ nhánh vô điều kiện được tạo ra theo một trong hai cách. Cách thứ nhất là triệu gọi một hàm hành sự. Khi trình biên dịch gặp phải tên của một hàm hành sự, nó ngưng thi hành đối với hàm hiện hành, và rẻ về hàm mới được triệu gọi. Khi hàm sau trả về một trị, việc thi hành tiếp tục với hàm nguyên thủy ngay ở câu lệnh bỏ dở sau lúc triệu gọi hàm. Thí dụ 4-11, minh hoạ điều vừa kể trên:

Thí dụ 4-11: Triệu gọi một hàm hành sự

```
using System;
class Functions
{   static void SomeMethod()
    {   Console.WriteLine("Xin gởi lời chào từ SomeMethod");
    }

    static void Main()
    {   Console.WriteLine("Trên Main! Triệu gọi SomeMethod()…");
        SomeMethod();
        Console.WriteLine("Trở lui lại Main().");
    }  // end Main
}
```

Kết xuất

```
Trên Main! Triệu gọi SomeMethod()…
Xin gởi lời chào từ SomeMethod
Trở lui lại Main().
```

Chương trình bắt đầu từ **Main()** là ngõ vào duy nhất, và tiến hành cho tới khi hàm **SomeMethod()** được triệu gọi. Tới đây, chương trình chuyển qua thi hành hàm **SomeMethod()**. Sau khi thi hành xong hàm, chương trình trở lui lại nơi bỏ dở nghĩa là câu lệnh kế tiếp nằm sau câu lệnh triệu gọi hàm **SomeMethod()**.

Cách thứ hai là tạo một rẽ nhánh vô điều kiện dùng đến một trong những từ chốt rẽ nhánh vô điều kiện: **goto, break, continue, return** hoặc **statementthrow.** Chúng tôi sẽ đề cập đến các từ chốt này trong các phần kế tiếp trong chương này, ngoại trừ **throw** sẽ được đề cập ở chương 12, "Thụ lý các biệt lệ".

4.5.2 Các câu lệnh rẽ nhánh có điều kiện

Rẽ nhánh có điều kiện sẽ được thực hiện thông qua các từ chốt **if, else,** hoặc **switch**. Rẽ nhánh có điều kiện chỉ xảy ra khi việc định trị biểu thức điều kiện (condition expression) cho ra **true**.

4.5.2.1 Câu lệnh If…else

Các câu lệnh **If…else** cho rẽ nhánh dựa trên một điều kiện (condition). Điều kiện là một biểu thức, được trắc nghiệm ngay ở đầu câu lệnh **if**. Nếu điều kiện suy ra là thỏa (**true**), thì câu lệnh (hoặc khối câu lệnh) trong phần thân của **if** sẽ được cho thi hành.

Câu lệnh **if** có thể có một câu lệnh **else**, nhưng là tùy chọn (optional). Câu lệnh **else** chỉ được thi hành khi điều kiện suy ra là không thỏa (**false**).

if (*condition-expression*)
    statement1
[**else**
    statement2]

Nếu câu lệnh **if** nhận *condition-expression* (nằm trong cặp dấu ngoặc) và định trị ra là **true** thì nó cho thi hành **statement1**. Bạn để ý **statement1** có thể là một khối gồm nhiều câu lệnh bao bởi cặp dấu ngoặc nghéo {}. Bạn cũng thấy là câu lệnh **else** là tùy chọn (nghĩa là nằm trong cặp dấu ngoặc vuông []). Thí dụ 4-12 sau đây minh hoạ việc sử dụng câu lệnh **if…else**:

Thí dụ 4-12: Sử dụng câu lệnh if…else

```
using System;
class Values
{   static void Main()
    {   int triMôt = 10;
        int triHai = 20;

        if (triMôt > triHai)
        {   Console.WriteLine("TriMôt: {0} lớn hơn TriHai: {1}",
                  triMôt, triHai);
```

```
        }
        else
        {   Console.WriteLine("TriHai: {0} lớn hơn TriMôt: {1}",
                triHai, triMôt);
        }

        triMôt = 30;  // cho triMôt trị lớn hơn

        if (triMôt > triHai)
        {   triHai = triMôt++;
            Console.WriteLine("\nCho triHai về trị triMôt, ");
            Console.WriteLine("và tăng 1 đối với triMôt.\n");
            Console.WriteLine("TriMôt: {0} TriHai: {1}",
                triMôt, triHai);
        }
        else
        {   triMôt = triHai;
            Console.WriteLine("Cho 2 biến bằng nhau. ");
            Console.WriteLine("TriMôt: {0} TriHai: {1}",
                triMôt, triHai);
        }
    }
}
```

Trên thí dụ 4-9, câu lệnh **if** đầu tiên trắc nghiệm xem **triMôt** có lớn hơn **triHai** hay không. Những tác tử quan hệ (relational operator) như lớn hơn (>), nhỏ thua (<) và bằng nhau (==) chắc bạn đã hiểu ý nghĩa và sử dụng thế nào, khỏi nhắc lại.

Trắc nghiệm xem trị của **triMôt** (=10) có lớn hơn trị của **triHai** (=20) cho ra **false**, vì **triMôt** không lớn hơn **triHai**, do đó câu lệnh **else** được triệu gọi và in ra:

```
TriHai: 20 lớn hơn TriMôt: 10
```

Lệnh **if** thứ hai cho ra **true** và tất cả các câu lệnh của khối **if** được thi hành, cho in ra hai hàng:

```
Cho triHai về trị triMôt,
và tăng 1 đối với triMôt.

TriMôt: 31 TriHai: 30
```

Khối câu lệnh

Bất cứ nơi nào C# chờ đợi một câu lệnh, bạn có thể thay thế bởi một khối câu lệnh (statement block). Một ***khối câu lệnh*** là một lô các câu lệnh được bao bởi cặp dấu ngoặc nghéo {}. Do đó, nơi nào bạn có thể viết:

```
if (someCondition)
   someStatement;
```

bạn có thể viết thay thế như sau:

```
if (someCondition)
{  statement1;
   statement2;
   statement3;
}
```

4.5.2.2 Các câu lệnh if lồng nhau

Nếu điều kiện khá phức tạp, bạn có thể cho lồng (nested) câu lệnh **if** với nhau, và điều này cũng khá phổ biến. Thí dụ, bạn muốn viết một chương trình định lượng nhiệt độ, và đặc biệt trả về những loại thông tin như sau:

- Nếu nhiệt độ bằng hoặc nhỏ thua 32 độ, chương trình phải báo động bạn biết có nước đá trên đường.
- Nếu nhiệt độ đúng bằng 32 độ, chương trình phải cho bạn biết có thể có vài mảng nước đá trên đường.
- Nếu nhiệt độ lớn hơn 32 độ, chương trình phải bảo đảm bạn biết là không có nước đá trên đường.

Có nhiều cách viết chương trình này. Thí dụ 4-13 minh hoạ một cách tiếp cận vấn đề

Thí dụ 4-13: Câu lệnh if lồng nhau.

```
using System;
class Values
{  static void Main()
   {  int nhiêtDô = 32;
      if (nhiêtDô <= 32)
      {  Console.WriteLine("Coi chừng! Băng đá trên đường!");
         if (nhiêtDô == 32)
         {  Console.WriteLine("Nhiệt đô đang đông lạnh,
                              coi chừng nước");
         }
```

```
            else
            {   Console.WriteLine("Để ý băng đá đen! Nhiệt độ: {0}",
                                                              nhiêtDô);
            }
        }
    }
}
```

Bạn thấy câu lệnh **if** nằm lồng trong lòng câu lệnh **if** thứ nhất, như vậy lô gic của **else** như sau: "vì đã biết nhiệt độ nhỏ thua hoặc bằng 32, và nó không bằng 32, thì là nhỏ thua 32".

4.5.2.3 Câu lệnh switch: một phương án thay thế if nằm lồng

Các câu lệnh **if** nằm lồng rất khó đọc, khó biết là đúng và khó gỡ rối (debug). Khi bạn có một loạt lựa chọn phức tạp, thì nên sử dụng câu lệnh **switch**, rất mạnh. Lôgic của **switch** như sau: "chọn ra một trị khớp với rồi theo đấy mà làm". Cú pháp như sau:

switch (*expression*)
{ **case** *constant-expression:*
statement
jump-statement
[**default:** *statement*]
}

Tất cả các tác tử được tạo ra đều không "bình đẳng"

Khi ta nhìn kỹ câu lệnh **if** thứ hai trên thí dụ 4-10, ta sẽ thấy tiềm tàng một vấn đề khá phổ biến. Câu lệnh **if** này trắc nghiệm xem nhiệt độ có bằng 32 hay không

if (nhiêtDô == 32)

Trên C/C++ có một vấn đề vốn gắn liền với loại câu lệnh này. Việc lập trình viên "tập tễnh" sử dụng tác tử gán (=) thay vì tác tử so sánh bằng (==) không phải là hiếm:

if (nhiêtDô = 32)

Sai lầm trên khó phát hiện, vì kết quả là 32 được gán cho biến **nhiêtDô**, và rồi 32 được trả về như là trị của câu lệnh gán. Vì bất cứ trị nào nonzero sẽ được cho là **true** trên

C và C#, như vậy câu lệnh **if** sẽ trả về **true**. Hiệu ứng phụ sẽ là, **nhiêtDô** sẽ được gán cho con số 32, cho dù nguyên ủy nó có hoặc không mang trị này. Đây là một bug khá phổ biến và hay bị "qua mặt", nếu nhà phát triển C# không tiên liệu.

C# giải quyết vấn đề bằng cách đòi hỏi câu lệnh **if** chỉ chấp nhận những trị bool. (true hoặc false). Con số 32 được trả về không phải là trị bool (nó là một số nguyên) và trên C# không có việc chuyển đổi tự động từ 32 qua **true**. Do đó, lỗi sai trên có thể phát hiện vào lúc biên dịch. Đây là một điều tốt đẹp, một cải tiến đầy ý nghĩa so với C++.

Như bạn có thể thấy, giống như câu lệnh **if**, biểu thức được đặt trong cặp dấu ngoặc ở đầu câu lệnh **switch**. Mỗi câu lệnh **case**, sau đó đòi hỏi một biểu thức hằng (constant expression); nghĩa là một trực kiện hoặc một hằng tượng trưng hoặc một enumeration.

Nếu một trường hợp (case) nào đó khớp, câu lệnh (hoặc khối câu lệnh) được gắn liền với case sẽ được thi hành. Và phải theo sau liền bởi một câu lệnh nhảy (jump); thường là lệnh **break**, cho chuyển việc thi hành liền sau ngoài **switch**. Ta cũng có thể dùng lệnh **goto** thay thế. Lệnh **goto** dùng nhảy về một case khác, như được minh hoạ bởi thí dụ 4-14:

Thí dụ 4-14: Sử dụng câu lệnh switch

```
using System;

class Values
{  static void Main()
   {  const int Democrat = 0;
      const int LiberalRepublican = 1;
      const int Republican = 2;
      const int Libertarian = 3;
      const int NewLeft = 4;
      const int Progressive = 5;

      int myChoice = Libertarian;

      switch (myChoice)
      {  case Democrat:
            Console.WriteLine("Bạn bỏ phiếu cho Democrat.\n");
            break;

         case LiberalRepublican:    // nhảy thẳng tuột xuống
                                    // case Republican
            // Console.WriteLine("LiberalRepublican bỏ phiếu
                                  Republican\n");

         case Republican:
            Console.WriteLine("Bạn bỏ phiếu cho Republican.\n");
            break;
```

```
            case Libertarian:
               Console.WriteLine("Libertarian bỏ phiếu cho
                                  Republican.\n");
               goto case Republican;

            case NewLeft:
               Console.WriteLine("NewLeft bây giờ Progressive.\n");
               goto case Progressive;

            case Progressive:
               Console.WriteLine("Bạn bỏ phiếu cho Progressive.\n");
               break;

            default:
               Console.WriteLine("Bạn chọn không đúng rồi.\n");
         }
         Console.WriteLine("Cảm ơn bạn đã bỏ phiếu.");
      }  // end Main
   }  // end Values
```

Trong thí dụ trên, bạn tạo những hằng cho những đảng chính trị khác nhau ở Mỹ. Tiếp theo, ta gán một trị gì đó (**Libertarian**) cho biến **myChoice** và bật qua trị này. Nếu lựa chọn của bạn là **Democrat**, ta in ra một phát biểu. Bạn để ý case này kết thúc với từ chốt **break**. **break** là lệnh nhảy cho phép bạn nhảy khỏi **switch** xuống dòng lệnh đầu tiên sau **switch**, đấy là lệnh in ra lời cảm ơn.

Case **LiberalRepublican** không có câu lệnh nào, do đó nó nhảy thẳng tuột xuống case kế tiếp là case **Republican**; nghĩa là nếu trị là **LiberalRepublican** hoặc **Republican** thì câu lệnh **Republican** sẽ được thi hành. Bạn chỉ nhảy thẳng tuột xuống như thế khi không có phần thân trong lòng câu lệnh. Nếu bạn cho bỏ dấu // câu lệnh Console.WriteLine nằm dưới case **LiberalRepublican** thì chương trình sẽ không chịu biên dịch.

Nếu bạn không cần một câu lệnh nhưng lại muốn cho thi hành một case khác, bạn có thể dùng **goto**, như trong trường hợp case **NewLeft**:

```
case NewLeft:
   Console.WriteLine("NewLeft bây giờ Progressive.\n");
   goto case Progressive;
```

Không nhất thiết là lệnh **goto** phải cho bạn đi về case nằm ngay liền ở dưới. Bạn thấy trường hợp **Libertarian** cho nhảy lộn ngược lên case **Republican**. Vì ta đã cho **myChoice** về **Libertarian**, nên ta in ra lời phát biểu **Libertarian** rồi sau đó nhảy qua case **Republican,** in ra phát biểu, gặp phải break, nhảy thoát khỏi switch, gặp lệnh in ra lời cảm ơn. Kết xuất như sau:

```
Libertarian bỏ phiếu cho Republican.
Bạn bỏ phiếu cho Republican.

Cảm ơn bạn đã bỏ phiếu.
```

Bạn để ý case Default: mà chúng tôi cho trích ra dưới đây:

```
default:
   Console.WriteLine("bạn chọn không đúng rồi.\n");
```

Nếu không có trường hợp nào khớp cả, thì case **default** sẽ được triệu gọi, báo cho người sử dụng sai lầm xảy ra trong lựa chọn.

4.5.2.4 Switch trên các câu lệnh kiểu chuỗi chữ

Trong thí dụ trước, trị của switch là một hằng số (integral constant). C# cho bạn khả năng chuyển qua dùng một **string**, cho phép viết như sau:

```
case "Libertarian":
```

Nếu chuỗi chữ khớp, thì lệnh **case** sẽ được thi hành.

4.5.3 Các câu lệnh rảo lặp[31] (iteration)

C# cung cấp một lô câu lệnh rảo lặp, bao gồm **for, while, do…while** và **foreach**. Ngoài ra, C# hỗ trợ những lệnh nhảy **goto, break, contine,** và **return**.

4.5.3.1 Lệnh goto

Lệnh **goto** cho phép bạn nhảy trực tiếp về một câu lệnh được đánh dấu bởi một nhãn (label). Nếu không biết sử dụng đúng cách **goto** thì sẽ tạo ra một chương trình "rối nùi" (người Âu Mỹ gọi là chương trình spaghetti, kiểu mì nui của Ý) không biết đâu mà lần. Lập trình viên kinh nghiệm khuyên ta nên tránh xa lệnh này. Tuy nhiên, đã có thì cũng nên giải thích cho bạn. Muốn dùng lệnh **goto**, bạn theo các bước sau đây:

1. Tạo một cái tên dùng làm nhãn (label) cho theo sau bởi dấu hai chấm (:).
2. **goto** về nhãn

[31] Người viết dịch từ "iteration" là "rảo lặp", nghĩa là đi rảo qua mang tính lặp đi lặp lại.

Label là một tên nhận diện câu lệnh mà bạn muốn cho nhảy về. Lệnh **goto** thường được gắn liền với một điều kiện, như được minh hoạ bởi thí dụ 4-15 sau đây:

Thí dụ 4-15: Sử dụng lệnh goto

```
using System;
public class Tester
{   public static int Main()
    {   int i = 0;
        repeat:   // label mang tên repeat
        Console.WriteLine("i: {0}", i);
        i++;
        if (i < 10)
            goto repeat;
            return 0;
    }   // end Main
}   // end Tester
```

Có một số hạn chế khi sử dụng **goto**. Bạn không thể nhảy vào một khối đoạn mã chẳng hạn vòng lặp **for**, và ta cũng không thể nhảy ra khỏi một lớp hoặc một khối **finally** (sau khối **try…catch**).

Bạn chớ nên lạm dụng việc sử dụng lệnh **goto**, vì dùng quá nhiều sẽ gây lúng túng cho người đọc và cũng khó bảo trì. Tuy nhiên, có một nơi mà bạn có thể sử dụng **goto** một cách "ngon lành" là trong câu lệnh **switch**. Bạn xem lại thí dụ 4-14.

4.5.3.2 Vòng lặp while

Y nghĩa vòng lặp **while** là: "Trong khi điều kiện này đang còn hiệu lực (**true**), thì tiếp tục làm việc này". Cú pháp như sau:

while (*expression*)
    statement

Như thường lệ, một *expression* là bất cứ câu lệnh nào trả về **true**. Câu lệnh **while** đòi hỏi một biểu thức sau khi định trị trả về một trị bool (**true** hoặc **false**), và lẽ dĩ nhiên là lệnh này có thể là một khối lệnh. Thí dụ 4-16 nhật tu thí dụ 4-15 dùng đến vòng lặp **while**:

Thí dụ 4-16: Sử dụng vòng lặp while

```
using System;
public class Tester
{   public static int Main()
```

```
    {   int i = 0;
        while (i < 10)
        {   Console.WriteLine("i: {0}", i);
            i++;
        }
        return 0;
    }   // end Main
}   // end Tester
```

Đoạn mã trong thí dụ 4-16 cũng cho ra kết quả giống đúc kết quả của thí dụ 4-15, nhưng lô gic ở đây rõ ràng hơn. Lệnh **while** cho biết "Trong khi (while) i nhỏ thua 10, thì in ra thông điệp và tăng 1 i"

Bạn để ý vòng lặp **while** trắc nghiệm trị của **i** trước khi vào vòng lặp, như vậy bảo đảm vòng lặp sẽ không chạy khi điều kiện trắc nghiệm là **false**; trong thí dụ trên, nếu **i** được khởi gán về 11 thì vòng lặp sẽ không chạy. Do đó, vòng lặp **while** được gọi là pre-test loop, nghĩa là trắc nghiệm trước: nếu trị của điều kiện trắc nghiệm cho ra false thì vòng lặp **while** sẽ không thi hành dù chỉ một lần.

Khác với vòng lặp **for**, vòng lặp **while** thường được dùng để lặp lại một câu lệnh hoặc một khối câu lệnh *theo một số lần không biết trước được.*Thông thường, trong lòng thân của while, sẽ có một câu lệnh cho đặt để một boolean flag về false dựa trên một số lần lặp lại để kích hoạt việc thoát khỏi vòng lặp:

```
bool DieuKien = false;
while (!DieuKien)
{
    // vòng lặp này chạy liên tục khi DieuKien đang còn true
    LamCaiGiDo();
    DieuKien = KiemTraDieuKien();
}
```

Bạn nên cho đặt thân vòng lặp trong một cặp dấu {} cho dù chỉ gồm một câu lệnh duy nhất, vì không biết đâu sau này bạn lại thêm câu lệnh cho thân vòng lặp. Như vậy sẽ tránh sai lầm.

4.5.3.3 Vòng lặp do… while

Đôi lúc vòng lặp **while** không giúp ta trong một mục đích nào đó. Trong vài trường hợp, có thể ta muốn bảo "cho chạy gì đó, trong khi điều kiện đang còn hiệu lực (true)". Nói cách khác, hành động đi cái đã, rồi sau đó khi hoàn thành công việc, cho kiểm tra điều kiện. Nếu còn thoả thì tiếp tục lại, bằng không thì thoát ra. Trong trường hợp này ta dùng vòng lặp **do… while**, mang cú pháp như sau:

do *expression* **while** *statement*

Một biểu thức là bất cứ câu lệnh nào trả về một trị. Thí dụ 4-17 cho thấy một sử dụng vòng lặp **do… while**:

Thí dụ 4-17: Sử dụng vòng lặp do… while

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

Trong thí dụ 4-17, biến **i** được khởi gán về 11 và trắc nghiệm **while** thất bại, nhưng sau khi những gì trong thân vòng lặp đã được thi hành một lần.

Vòng lặp **do... while** được gọi là post-test loop, nghĩa là việc trắc nghiệm điều kiện vòng lặp chỉ được thực hiện sau khi thân vòng lặp đã được thi hành. Như vậy, vòng lặp **do… while** rất hữu ích đối với những trường hợp theo đấy một khối câu lệnh phải được thi hành *ít nhất một lần*:

```
bool DieuKien;
do
{   // vòng lặp này chạy ít nhất một lần cho dù DieuKien là false
    LamCaiGiDo();
    DieuKien = KiemTraDieuKien();
} while (DieuKien);
```

4.5.3.4 Vòng lặp for

Nếu ta quan sát kỹ vòng lặp **while**, như trên thí dụ 4-16, ta thường thấy một mẫu dáng (pattern) diễn ra trong những lệnh lặp lại như sau: (1) khởi gán một biến (i = 0), (2) trắc nghiệm biến (i < 10), (3) thi hành một loạt câu lệnh, (4) tăng trị của biến (i++). Vòng lặp **for** cho phép bạn phối hợp 3 bước kể trên (1-2-4) thành một câu lệnh vòng lặp:

for ([*initializers*]; [*condition*]; [*iterators*])
    statement

theo đấy *initializer* là biểu thức được định trị trước khi vòng lặp được thi hành (thông thường khởi gán một biến cục bộ dùng làm cái đếm); còn *condition* là một biểu thức được trắc nghiệm trước mỗi tua lặp lại (thí dụ trắc nghiệm xem cái đếm vòng lặp nhỏ

thua một trị nào đó hay không); *iterator* là một biểu thức sẽ được định trị sau mỗi tua lặp lại (thí dụ tăng cái đếm vòng lặp).

Vòng lặp **for** cũng được gọi là pre-test loop, vì điều kiện vòng lặp được trắc nghiệm trước khi các câu lệnh thân vòng lặp được thi hành, và sẽ không thi hành chi cả khi điều kiện vòng lặp là false. Vòng lặp **for** rất hữu ích khi ta muốn cho lặp lại một câu lệnh hoặc một khối câu lệnh *theo một số lần mà ta có thể biết được trước*.

Thí dụ 4-18 minh hoạ việc sử dụng vòng lặp **for**

Thí dụ 4-18: Sử dụng vòng lặp for

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

Kết xuất

```
0       0
1 2 3 4 5 6 7 8 9 10  10
11 12 13 14 15 16 17 18 19 20   20
21 22 23 24 25 26 27 28 29 30   30
31 32 33 34 35 36 37 38 39 40   40
41 42 43 44 45 46 47 48 49 50   50
51 52 53 54 55 56 57 58 59 60   60
61 62 63 64 65 66 67 68 69 70   70
71 72 73 74 75 76 77 78 79 80   80
81 82 83 84 85 86 87 88 89 90   90
91 92 93 94 95 96 97 98 99
```

Vòng lặp **for** ở thí dụ trên sử dụng tác tử modulus (%) mà chúng tôi sẽ mô tả ở cuối chương này. Trị của **i** sẽ được in ra khi nào **i** là bội số của 10:

```
if (i % 10 == 0)
```

Một tab được in ra theo sau bởi trị. Do đó các số hàng 10 (20, 30, 40, v.v..) được in ra phía tay phải kết xuất.

Mỗi trị riêng biệt sẽ được in ra dùng đến **Console.Write**, hoạt động giống như **Console.WriteLine** nhưng không có ký tự sang hàng (new line), cho phép in ra nhiều số trên một hàng.

Trong vòng lặp **for**, điều kiện được trắc nghiệm trước khi thi hành các câu lệnh. Do đó, trong thí dụ này, **i** được khởi gán về zero, sau đó trắc nghiệm xem có nhỏ thua 100 hay không. Vì i<100 trả về **true**, nên các câu lệnh trong thân vòng lặp **for** sẽ được thi hành. Sau khi thi hành xong, **i** được tăng 1 (i++).

Bạn để ý biến **i** nằm trong phạm vi (scope) vòng lặp **for** (nghĩa là biến **i** chỉ được nhìn thấy trong lòng vòng lặp **for**). Thí dụ 4-19 sau đây sẽ không biên dịch được:

Thí dụ 4-19: Phạm vi các biến được khai báo trong một vòng lặp for

```
using System;
public class Tester
{  public static int Main()
   {  for (int i = 0; i <100; i++)
      {  Console.Write("{0} ", i);
         if (i%10 == 0)
         {  Console.WriteLine("\t{0}", i);
         }
      }
      Console.WriteLine("\n Trị cuối cùng của i: {0}", i);
      return 0;
   }  // end Main
}  // end Tester
```

Dòng lệnh in đậm sẽ không in ra, vì biến **i** không có sẵn ngoài phạm vi bản thân vòng lặp.

Vòng lặp for lồng nhau

Các vòng lặp **for** có thể nằm lồng nhau (nested), làm thế nào vòng trong (inner loop) phải thi hành trọn vẹn một lần đối với mỗi tua lặp lại của vòng ngoài (outer loop). Cơ chế này thường được dùng để rảo qua một bản dãy nhiều chiều hình chữ nhật (rectangular multidimensional array). Vòng lặp tận ngoài cùng sẽ rảo qua từng hàng một, còn vòng lặp nằm trong cùng sẽ rảo qua từng cột một của một hàng nào đó. Thí dụ 4-20, minh hoạ vòng lặp **for** nằm lồng nhau:

Thí dụ 4-20: Sử dụng vòng lặp for nằm lồng nhau:

```
// vòng lặp này rảo qua từng hàng một
for (int i = 0; i < 100; i++)
{
   // vòng lặp này rảo qua từng cột một
```

```
        for (int j = 0; j < 25; j++)
        {
            a[i, j] = 0;
        }
    }
```

Bạn để ý biến cái đếm vòng lặp nằm trong, ở đây là j, được khai báo lại tuần tự qua mỗi tua lặp lại của vòng lặp nằm ngoài. Thí dụ trên cho xoá trắng một bản dãy.

Whitespace và Dấu ngoặc nghéo {}

Việc sử dụng whitespace (ký tự trắng) trong lập trình đã gây ra ít nhiều tranh cãi. Thí dụ, vòng lặp **for** trên thí dụ 4-14:

```
for (int i = 0; i <100; i++)
{   Console.Write("{0} ", i);
    if (i%10 == 0)
    {   Console.WriteLine("\t{0}", i);
    }
}
```

cũng có thể viết với nhiều ký tự trắng xen kẻ giữa các tác tử:

```
for ( int i = 0; i < 100; i++ )
{   Console.Write("{0} ", i);
    if ( i % 10 == 0)
    {   Console.WriteLine("\t{0}", i);
    }
}
```

Vì các câu lệnh **for** và **if** đơn lẻ không cần đến cặp dấu {}, ta cũng có thể viết lại đoạn mã trên như sau:

```
for ( int i = 0; i < 100; i++ )
    Console.Write("{0} ", i);

    if ( i % 10 == 0)
        Console.WriteLine("\t{0}", i);
```

Tất cả chỉ là mỗi người mỗi sở thích riêng. Mặc dù, chúng tôi cho là ký tự có thể làm cho đoạn mã sáng sủa dễ đọc, nhưng quá nhiều ký tự trắng cũng gây bối rối. Trong tập sách này chúng tôi cố gắng giữ số tối thiểu để tiết kiệm chỗ trên các trang in.

Trong câu lệnh **for**, bạn thường thấy mẫu dáng như sau:

```
for (int i = 0; i <100; i++)
{
    ...
}
```

nghĩa là trước mỗi dấu chấm phẩy chỉ có một câu lệnh: int i; hoặc i < 100;. Tuy nhiên, ta cũng có thể có nhiều câu lệnh trước dấu chấm phẩy, mỗi câu lệnh được phân cách bởi dấu phẩy. Thí dụ sau đây minh hoạ cách sử dụng đến nhiều câu lệnh.

Ta có thể tạo một lớp cho mang tên SumOf (tổng cộng của). SumOf có 3 biến thành viên private:

```
private int startNumber = 1;  // số khởi đi
private int endNumber; // số kết thúc
private int[] theSums; // tổng cộng
```

Biến bản dãy **theSums** tượng trưng trị tổng của tất cả các số từ **startNumber** đến **endNumber**. Do đó, nếu **startNumber** là 1 và **endNumber** là 10, thì bản dãy sẽ có những trị: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55. Mỗi trị là tổng của trị đi trước cộng với trị kế tiếp trên loạt số.

Hàm constructor **SumOf()** nhận hai số nguyên: số khởi đi, **startNumber**, và số kết thúc, **endNumbe**r. Hàm gán các số này cho biến cục bộ và gọi một hàm hỗ trợ, **ComputeSums()** để điền vào nội dung bản dãy bằng cách tính những tổng trong loạt số từ **startNumber** đến **endNumber**:

Thí dụ 4-21: Sử dụng vòng lặp for với nhiều câu lệnh trong phần điều kiện và iterator:

```
public SumOf(int start, int end)
{
    startNumber = start;
    endNumber = end;
    ComputeSums();
}

private void ComputeSums()
{
    int count = endNumber – starNumber + 1; // tính số lần rào qua
    int[] theSums = new int[count];   // tạo một bản dãy
    theSums[0] = startNumber;
    for (int i = 1, j = startNumber + 1; i < count; i++, j++)
    {
        theSums[i] = j + theSums[i-1];
    }
}
```

Bạn thấy vòng lặp `for` sau đây có hơi bất thường so với những vòng lặp `for` trước kia:

```
for (int i = 1, j = startNumber + 1; i < count; i++, j++)
```

Đầu tiên khởi đi bạn có hai câu lệnh "int i = 1" và "j = startNumber"phân cách bởi dấu phẩy, coi như là trị ban đầu khởi động vòng lặp, tiếp theo là một câu lệnh cái đếm bình thường, và cuối cùng có hai biến đếm i++ và j++ cùng tăng 1, và đây cũng là hai câu lệnh phân cách bởi dấu phẩy. Bạn đọc kỷ chương trình. Ở đây chúng tôi có đề cập đến khái niệm về bản dãy (array) hơi sớm một chút. Chương 10, "Bản dãy, Indexers và Collections", sẽ đề cập đến vấn đề bản dãy.

4.5.3.5 Vòng lặp foreach

Câu lệnh **foreach** là khá mới đối với dòng họ ngôn ngữ C, được đưa vào cho phép đi rảo qua các phần tử nằm trong một tập hợp (collection) hoặc một bản dãy. Chương 10. "Bản dãy, Indexers và Collections", sẽ đề cập đến câu lệnh **foreach** này.

4.5.4 Các câu lệnh nhảy: continue, break và return

4.5.4.1 Câu lệnh Continue

Đôi khi bạn muốn khởi động lại một vòng lặp nhưng lại không muốn thi hành phần lệnh còn lại trong vòng lặp, ở một điểm nào đó trong thân vòng lặp. Câu lệnh **continue** làm cho vòng lặp bắt đầu lại từ đầu và tiếp tục thi hành.

Câu lệnh **continue** được dùng trong lòng các vòng lặp **for, foreach, while** hoặc **do..while**. Tuy nhiên, nó chỉ hiện hữu đối với *việc lặp lại hiện hành,* nghĩa là việc thi hành bắt đầu từ đầu của tua vòng lặp kế tiếp.

Bạn để ý, không được dùng **continue** để thoát một khối **finally**, và nếu một câu lệnh **continue** đưa đến kết quả là thoát khỏi một khối **try**, thì bất cứ khối **finally** nào được gắn liền sẽ được thi hành trước khi quyền điều khiển được trả về cho tua lặp lại kế tiếp của vòng lặp.

4.5.4.2 Câu lệnh break

Một khía cạnh khác là có khả năng *ngưng ngang xương việc thi hành và thoát khỏi vòng lặp*. Ta dùng đến câu lệnh **break.** Câu lệnh **break** có thể dùng trên các vòng lặp **for, foreach, while** hoặc **do..while**. Quyền điều khiển sẽ trao qua cho câu lệnh nằm ngay liền ở cuối vòng lặp. Thí dụ 4-22 minh hoạ việc sử dụng câu lệnh **break**:

Thí dụ 4-22: Sử dụng câu lệnh break trong vòng lặp for

```
for (int i = 0; i < 10; i++)
{   Console.Write("Khỏ vào một từ: ");
    string s = Console.ReadLine();    // đọc vào gì vừa khỏ
    if (s == "End")
    {   // thoát khỏi vòng lặp khi người sử dụng khỏ 'End'
        break;
    }
    Console.WriteLine("bạn đã khỏ vào: " + s);
}
// việc thi hành bắt đầu lại sau khi người sử dụng khỏ 'End'
```

Nếu câu lệnh **break** nằm trong một vòng lặp lồng nhau, thì quyền điều khiển sẽ chuyển qua vào cuối vòng lặp nằm sâu nhất. Nếu câu lệnh **break** nằm ngoài một câu lệnh **switch** hoặc một vòng lặp, thì sẽ gây ra sai lầm vào lúc biên dịch.

Sau đây là một thí dụ 4-23 minh hoạ cơ chế làm việc của **continue** và **break.** Đoạn mã này tạo một hệ thống báo hiệu giao thông. Tín hiệu được mô phỏng bằng cách khỏ vào những con số và chữ hoa từ bàn phím, sử dụng **Console.ReadLine** đọc một hàng văn bản từ bàn phím.

Giải thuật khá đơn giản: nhận được zero có nghĩa điều kiện bình thường, không làm gì cả ngoài việc ghi theo dõi biến cố. (Trong trường hợp này, chương trình đơn giản viết ra màn hình một thông điệp; ứng dụng thực thụ có thể nhập một mẫu tin ghi vào căn cứ dữ liệu kèm theo dấu ấn thời gian - timestamp). Khi nhận được một tín hiệu Abort (bỏ ngang, mô phỏng bởi chữ A), vấn đề được ghi theo dõi và tiến trình tạm ngưng thi hành. Cuối cùng, đối với những biến cố khác, một báo động sẽ nổi lên, báo cảnh sát. Nếu tín hiệu là X, báo động sẽ phát ra nhưng vòng lặp **while** cũng ngưng luôn.

Thí dụ 4-23: Sử dụng continue và break

```
using System;

public class Tester
{   public static int Main()
    {   string signal = "0"; // khởi gán "vô thưởng vô phạt"
        while (signal != "X")   // X cho biết ngưng
```

```
        {   Console.Write("Khỏ vào một tín hiệu: ");
            signal = Console.ReadLine();  // đọc tín hiệu

            // làm cái gì ở đây, bất kể nhận được tín hiệu gì
            Console.WriteLine("Tín hiệu nhận được:{0}", signal);

            if (signal = "A")
            {   // bỏ ngang và ghi theo dõi
                Console.WriteLine("Sai! Bỏ ngang\n");
                break;
            }

            if (signal = "0")
            {   // bình thường, ghi theo dõi và tiếp tục
                Console.WriteLine("Mọi việc ổn.\n");
                continue;
            }

            // Có vấn đề - Hành động, ghi theo dõi và tiếp tục
            Console.WriteLine("{0} – báo động!\n", signal);
        }
        return 0;
    }  // end Main
}  // end Tester
```

Kết xuất

Khỏ vào một tín hiệu: 0
Tín hiệu nhận được: 0
Mọi việc ổn.

Khỏ vào một tín hiệu: B
Tín hiệu nhận được: B
B – báo động!

Khỏ vào một tín hiệu: A
Tín hiệu nhận được: A
Sai! Bỏ ngang.

Press any key to continue

Điểm cần nhấn mạnh ở đây là khi nhận được tín hiệu A, hành động trong câu lệnh **if** tiếp tục thi hành rồi sau đó chương trình ngưng ngang ra khỏi vòng lặp không gây ra báo động. Khi nhận được tín hiệu 0 cũng không nên gây ra báo động, do đó chương trình tiếp tục (continue) từ đầu vòng lặp.

Bạn để ý là **continue** và **break** có thể tạo nhiều điểm thoát khỏi vòng lặp làm cho đoạn mã khó hiểu và khó bảo trì. Cho nên, bạn nên sử dụng hai loại lệnh này một cách có chừng mực và có ý thức.

4.5.4.3 Câu lệnh Return

Câu lệnh **return** dùng thoát khỏi một hàm hành sự của một lớp, trả quyền điều khiển về cho phía triệu gọi hàm (caller). Nếu hàm có một kiểu dữ liệu trả về, thì **return** phải trả về một trị kiểu dữ liệu này; bằng không thì câu lệnh được dùng không có biểu thức.

4.6 Các tác tử (operators)

Một *tác tử*[32] (operator) là một ký hiệu yêu cầu C# thực hiện một hành động gì đó. Những kiểu dữ liệu bẩm sinh C# (chẳng hạn **int**) thường hỗ trợ một số tác tử, chẳng hạn tác tử gán (assignment operator), v.v.. Việc sử dụng các tác tử này thường mang tính trực giác, ngoại trừ tác tử gán (=) và tác tử so sánh bằng nhau (equality operator, ==), gây cho người sử dụng một ít bối rối.

4.6.1 Tác tử gán (=)

Trong phần "Biểu thức" (xem mục 4.3) đi trước, chúng tôi đã minh hoạ việc sử dụng tác tử gán (=). Tác tử gán lấy trị của toán hạng ở vế phải dấu (=) ghi lên toán hạng ở vế trái.

Bạn để ý, ta thường hay lẫn lộn tác tử gán (=) với tác tử so sánh bằng nhau (==). Trong câu lệnh if, nếu bạn viết

```
if (x = 3)
```

thay vì:

```
if ( x == 3)
```

thì trên C++, trình biên dịch sẽ không gây ra sai lầm, vì trị số 3 sẽ được gán cho x, và nếu tác vụ thành công thì biểu thức trên sẽ cho ra trị bool là **true.** Kết quả là bao giờ cũng là **true**, và chương trình sẽ hoạt động không như theo ý muốn. Nhưng trên C#, trình biên dịch không cho phép chuyển đổi ngầm thành trị bool, do đó lệnh sẽ gây ra sai lầm.

[32] Chúng tôi không gọi là "toán tử"

4.6.2 Tác tử toán học

C# sử dụng 5 tác tử toán số học, 4 dành cho tính toán thông thường, và tác tử thứ năm trả về số dư (remainder) của bài toán chia số nguyên. Các phần sau đây xét đến việc sử dụng các tác tử này.

*4.6.2.1 Tác tử số học đơn giản (+, -, *, /)*

C# cung cấp những tác tử dùng trong việc tính toán số học đơn giản: **trừ** (-), **cộng** (+), **nhân** (*) và **chia** (/), hoạt động bình thường theo sự chờ đợi của bạn, ngoại trừ biệt lệ chia số nguyên. Nếu nhân hai số hoặc trừ hai số, và nếu biểu thức được kiểm tra phạm trù (checked context) khi kết quả nằm ngoài miền giá trị (range) thì biệt lệ tràn năng **OverFlowException** sẽ được tung ra. Còn khi chia cho một zero thì lại xảy ra biệt lệ **DivideByZeroException**.

Khi bạn chia hai số nguyên, C# chia theo kiểu học sinh tiểu học: nó bỏ đi số lẻ còn lại. Như vậy chia 17 cho 4 sẽ cho ra 4 với số dư là 1. Chia dùng tác tử "/" sẽ bỏ đi số dư. C# cung cấp của bạn một tác tử đặc biệt, gọi là **modulus (%)** cho phép giữ lại số dư khi chia xong.

Tuy nhiên, bạn để ý là C# trả về kết quả số lẻ khi bạn chia float, double và decimal.

4.6.2.2 Tác tử modulus (%) để trả về số dư sau khi chia một số nguyên

Muốn tìm lại số dư sau khi chia số nguyên, bạn sử dụng tác tử modulus (%). Thí dụ, câu lệnh **17%4** sẽ cho ra 1 là số dư sau khi chia số nguyên.

Xem ra modulus % rất hữu ích hơn bạn tưởng. Khi bạn thực hiện modulus **n** trên một số bội số của *n*, kết quả sẽ là zero. Do đó, **80 % 10** sẽ bằng 0, vì 80 là bội số của 10. Điều này cho phép bạn tạo những vòng lặp thi hành một điều gì đó trong vòng lặp cứ theo nhịp độ n lần, bằng cách trắc nghiệm một cái đếm (counter) xem %n có bằng zero hay không. Cách thức này rất tiện lợi khi kết hợp sử dụng với vòng lặp **for**. Thí dụ 4-24 sau đây minh hoạ tác dụng của modulus trên việc chia theo integer, float, double và decimal.

Thí dụ 4-24: Chia và Modulus

```
using System;
```

```
class Values
{   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}\nfloat:\t\t{1}\n",
            i1/i2, f1/f2);
        Console.WriteLine("double:\t\t{0}\ndecimal:\t{1}",
            d1/d2, dec1/dec2);
        Console.WriteLine("\nModulus:\t{0}", i1%i2);
    }
}
```

Kết xuất

Integer: 4
float: 4.25
double: 4.25
decimal: 4.25

Modulus: 1

Bây giờ ta thử xét dòng lệnh sau đây trích từ Thí dụ 4-24 trên

```
Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n", i1/i2, f1/f2);
```

Dòng lệnh bắt đầu với triệu gọi **Console.WriteLine**, trao qua cho phần chuỗi sau đây:

```
"Integer:\t{0}\n
```

để in ra **Integer:** theo sau bởi một canh cột (**\t**), theo sau bởi thông số đầu tiên {0} rồi lại theo sau bởi một ký tự sang hàng mới (\n). Đoạn mã nhỏ kế tiếp là:

```
float:\t\t{1}\n
```

cũng tương tự như thế; nó in ra **float:** theo sau bởi hai canh cột để bảo đảm canh đúng theo chiều đứng, rồi nội dung thông số thứ hai {1} và cuối cùng một ký tự sang hàng mới (\n). Bạn để ý dòng lệnh cuối cùng in modulus:

```
Console.WriteLine("\nModulus:\t{0}", i1%i2);
```

Lần này, chuỗi chữ bắt đầu bởi một ký tự sang hàng mới (\n) làm cho nhảy qua hàng mới trước khi in **Modulus:**. Bạn có thể thấy tác dụng trên kết xuất.

4.6.3 Tác tử tăng giảm (++, --)

Hoạt động phổ biến trong lập trình là cộng thêm một trị vào một biến, hoặc bớt đi một trị khỏi một biến, hoặc nói cách khác thay đổi trị về mặt tính toán, rồi gán trị mới trở lại biến cũ. Bạn cũng có thể gán cho một biến khác. Trong 2 phần kế tiếp chúng tôi sẽ đề cập đến những trường hợp vừa kể trên.

4.6.3.1 Tác tử tính toán và gán trở lại

Giả sử bạn muốn tăng biến **mySalary** (lương căn bản) lên 5000. Bạn có thể làm như sau:

```
mySalary = mySalary + 5000;
```

Phép tính cộng xảy ra trước việc gán; hoàn toàn hợp lệ gán kết quả trở về biến nguyên thủy. Do đó, khi tác vụ này hoàn thành, **mySalary** được tăng 5000. Bạn có thể thực hiện loại gán này với bất cứ tác tử toán học, v.v..:

```
mySalary = mySalary * 5000;   // với tác tử nhân
mySalary = mySalary - 5000;   // với tác tử trừ
```

Nhu cầu tăng và giảm các biến quá phổ biến, nên C# cho bao gồm những tác tử đặc biệt dành cho tự gán (self-assignment). Ta có thể kể: **+=, -=, *=, /=,** và **%=**, phối hợp cộng, trừ, nhân, chia, modulus với tự gán. Do đó, bạn có thể viết lại các dòng lệnh trên:

```
mySalry += 5000;    // tăng thêm 5000
mySalry *= 5000;    // nhân mySalary cho 5000
mySalry -= 5000;    // trừ mySalary bớt đi 5000
```

Vì việc cộng trừ bởi 1 quá phổ biến, C# (giống như C/C++) cũng cung cấp hai tác tử đặc biệt. Để tăng 1, bạn dùng tác tử ++, còn muốn trừ đi 1, bạn dùng tác tử -- .Do đó, muốn tăng **myAge** 1 tuổi, bạn có thể viết: `myAge++;`

4.6.3.2 Tác tử tiền tố và tác tử hậu tố (prefix & postfix operator)

Để cho vấn đề thêm rắc rối, có thể bạn muốn tăng trị của một biến rồi gán trị kết quả cho một biến thứ hai khác:

```
firstValue = secondValue++;
```

Câu hỏi được đặt ra là: bạn muốn gán trước khi tăng trị hay là sau? Nói cách khác, nếu khởi đi **secondValue** mang trị 10**,** bạn muốn kết thúc với cả hai **firstValue** và **secondValue** bằng 11, hoặc là bạn muốn **firstValue** bằng 10 (trị nguyên thủy) và **secondValue** bằng 11?

C# (lại một lần nữa giống như C/C++) cung cấp hai loại "hương vị" đối với tác tử tăng giảm: **prefix** (tiền tố) và **postfix** (hậu tố). Do đó, bạn có thể viết:

```
firstValue = secondValue++; // postfix, vì dấu ++ nằm liền sau biến
```

có nghĩa là gán trước rồi tăng sau (**firstValue** = 10, **secondValue** = 11), hoặc bạn cũng có thể viết:

```
firstValue = ++sceondValue; // prefix,vì dấu ++ nằm liền trước biến
```

có nghĩa là tăng trước rồi gán sau (**firstValue** = 11, **secondValue** = 11). Điểm quan trọng là hiểu tác dụng khác nhau của **prefix** và **postfix**, như được minh hoạ bởi thí dụ 4-25 sau đây:

Thí dụ 4-25: Sử dụng tác tử tăng với prefix và postfix

```
using System;

class Values
{   static void Main()
    {   int triMôt = 10;
        int triHai;
        triHai = triMôt++;
        Console.WriteLine("Sau postfix: {0}, {1}", triMôt, triHai);
        triMôt = 20;
        triHai = ++TriMôt;
        Console.WriteLine("Sau prefix: {0}, {1}", triMôt, triHai);
    }
}
```

Kết xuất
Sau postfix: 11, 10
Sau prefix: 21, 21

4.6.4 Tác tử quan hệ

Tác tử quan hệ (relational operator) dùng so sánh hai trị, để trả về một trị bool (**true** hoặc **false**). Tác tử lớn hơn (>) sẽ trả về true nếu trị ở vế trái tác tử lớn hơn trị ở vế phải. Do đó, **5 > 2** sẽ trả về **true**, trong khi **2 > 5** sẽ trả về **false**.

Bảng 4-4 liệt kê các tác tử quan hệ. Bảng này giả định hai biến **bigValue** mang trị số 100, còn **smallValue** mang trị 50.

Bảng 4-4: Các tác tử quan hệ (giả định bigValue = 100 còn smallValue = 50)

Tên tác tử	Tác tử	Câu lệnh	Kết quả định trị
Equals	==	bigValue == 100	true
		bigValue == 80	false
Not Equals	!=	bigValue != 100	false
		bigValue != 80	true
Greater than	>	bigValue > smallValue	true
Greater than hoặc equals	>=	bigValue >= smallValue	true
		smallValue >= bigValue	false
Less than	<	bigValue < smallValue	false
Less than hoặc equals	<=	smallValue <= bigValue	true
		bigValue <= smallValue	false

Bạn để ý, tác tử so sánh bằng nhau (==) , được tạo với hai dấu (=) không có ký tự trắng ở giữa, dùng so sánh hai đối tượng ở hai đầu tác tử, cho ra trị bool **false** hoặc **true**. Do đó, câu lệnh:

```
myX == 5;
```

định trị cho ra **true** nếu và chỉ nếu **myX** là một biến mang trị 5.

Bạn để ý đừng lẫn lộn tác tử gán (=) với tác tử so sánh bằng nhau (==). Chuyện xảy ra rất thường xuyên.

4.6.5 Sử dụng các tác tử lô gic với điều kiện

Câu lệnh **if** mà ta đã đề cập trước đó thường trắc nghiệm liệu xem một điều kiện có thoả (true) hay không. Thường xuyên bạn phải trắc nghiệm liệu xem 2 điều kiện cả hai thỏa hay không, hoặc chỉ một thỏa, hoặc không điều kiện nào thỏa. C# cung cấp một lô tác tử lô gic cho việc này, như theo bảng 4-5. Bảng này giả định hai biến **x** và **y**, theo đấy **x** mang trị 5 còn **y** mang trị 7.

Bảng 4-5: Các tác tử lô gic của C# (giả định x=5 và y=7)

Tên	**Tác tử**	**Câu lênh**	**Định trị**	**Lô gic**
Conditional AND	&&	(x == 3) && (y == 7)	false	Cả hai phải true
Conditional OR	||	(x == 3) || (y == 7)	true	hoặc mỗi một hoặc cả hai phải true
Conditional NOT	!	! (x == 3)	true	biểu thức phải false

Tác tử **and** trắc nghiệm xem cả hai câu lệnh có true hay không. Cột thứ 3 cho thấy thí dụ minh hoạ việc sử dụng tác tử **and**:

```
(x == 3) && (y ==7)
```

Toàn bộ biểu thức sau khi định trị cho ra false, vì một phía (x == 3) là false.

Với tác tử **or**, hoặc một phía này phía kia, hoặc cả hai là true; biểu thức là false khi cả hai phía đều false. Do đó, trong trường hợp thí dụ trên bảng 4-4:

```
(x == 3) || (y == 7)
```

toàn bộ biểu thức sau định trị cho ra true vì một phía (y == 7) là true.

Với tác tử **not**, câu lệnh là true nếu biểu thức la false, và ngược lại. Trong trường hợp trên bảng:

```
! (x == 3)
```

toàn bộ biểu thức sau định trị cho ra true vì biểu thức được trắc nghiệm (x == 3) là false

4.6.6 Các tác tử lô gic hoặc bitwise operator

C# còn định nghĩa một số tác tử dùng làm việc với những bit của một con số hoặc với một trị bool. Bảng 4-6 cho thấy các tác tử này:

Bảng 4-6: Các tác tử lô gic hoặc bitwise operator

Tên	Tác tử	Ý nghĩa
Bitwise AND	&	Bitwise AND trên hai tác tố
Bitwise OR	|	Bitwise OR trên hai tác tố
Bitwise XOR	^	Bitwise exclusive OR (XOR) trên hai tác tố
Right shift operator	>>	Dịch một số bit qua phải
Left shift operator	<<	Dịch một số bit qua trái
Right shift assignment	>>=	Dịch một số bit qua phải rồi gán
Left shift assignment	<<=	Dịch một số bit qua phải rồi gán

Các tác tử **&, |,** và ^ thường làm việc trên kiểu dữ liệu số hoặc trên kiểu dữ liệu bool, và theo từng bit một (do đó có từ bitwise).

Còn các tác tử >>, <<, >>= và <<= thì thường cho dịch bit qua phải hoặc trái. Khi dịch qua trái (<<, hoặc <<=) thì các bit high-order sẽ bị bỏ rơi, còn các bit low-order sẽ được thêm vào zero. Còn khi dịch qua phải (>>, hoặc >>=) đối với uint và ulong thì các bit low-order sẽ bị bỏ rơi, còn các bit high-order trống sẽ được thêm vào zero, còn đối với int hoặc long thì các bit low-order sẽ bị bỏ rơi, còn các bit high-order trống sẽ được thêm vào zero nếu x là dương, và được thêm vào 1 nếu x là âm.

Chắc bạn đã biết dịch qua phải một bit thì trị sẽ chia 2, nhưng nếu dịch qua trái một bit thì trị sẽ nhân 2, và dịch qua phải 2 bit thì trị bị chia cho 4, còn dịch qua trái 2 bit thì trị lại nhân 4, v.v...

4.6.7 Các tác tử kiểu dữ liệu (Type operator)

4.6.7.1 Tác tử is

Tác tử **is** cho phép bạn kiểm tra xem một đối tượng có tương thích với một kiểu dữ liệu nào đó hay không. Thí dụ muốn kiểm tra xem một biến có tương thích với kiểu dữ liệu **object** hay không:

```
int i = 10;
if (i is object)
{   // Trên C# mọi cái đều là đối tượng
```

```
    Console.WriteLine("i là một đối tượng ");
}
```

int, giống như mọi kiểu dữ liệu C# khác đều được dẫn xuất từ **object**, nên biểu thức **i is object** sẽ cho ra **true**, cho nên thông điệp sẽ in ra.

4.6.7.2 Tác tử sizeof

bạn có thể xác định kích thước (tính theo byte) của một biến kiểu trị (value type) trên stack bằng cách dùng tác tử **sizeof**. Thí dụ:

```
string s = "A string";
unsafe
(
    Console.WriteLine(sizeof(int));
}
```

Bạn để ý, ta chỉ có thể sử dụng tác tử **sizeof** trên đoạn mã unsafe (không an toàn). Theo mặc nhiên, trình biên dịch C# gạt qua bất cứ đoạn mã nào có chứa những khối unsafe, do đó ta dùng mục chọn **/unsafe** trên trình biên dịch command-line hoặc cho đặt để mục chọn `Allow unsafe code blocks` về true trên Visual Studio .NET. Bạn có thể tìm thấy mục chọn này trên trang **Property** bằng cách chọn **Configuration Properties | Build.** Ngoài ra, chỉ sử dụng **sizeof** đối với các biến kiểu trị mà thôi, nghĩa là không sử dụng với dữ liệu kiểu qui chiếu. Tác tử **sizeof** không được nạp chồng (overloaded).

4.6.7.3 Tác tử typeof

Tác tử **typeof** trả về một đối tượng **System.Type** tượng trưng cho một kiểu dữ liệu được chỉ định. Thí dụ, **typeof(string)** sẽ trả về một đối tượng **Type** tượng trưng cho kiểu dữ liệu **System.String**. Tác tử này chỉ hữu ích khi ta muốn có thông tin liên quan đến một đối tượng động. Sau đây là thí dụ sử dụng tác tử typeof , trích từ MSDN.

Thí dụ 4-26: Sử dụng tác tử typeof

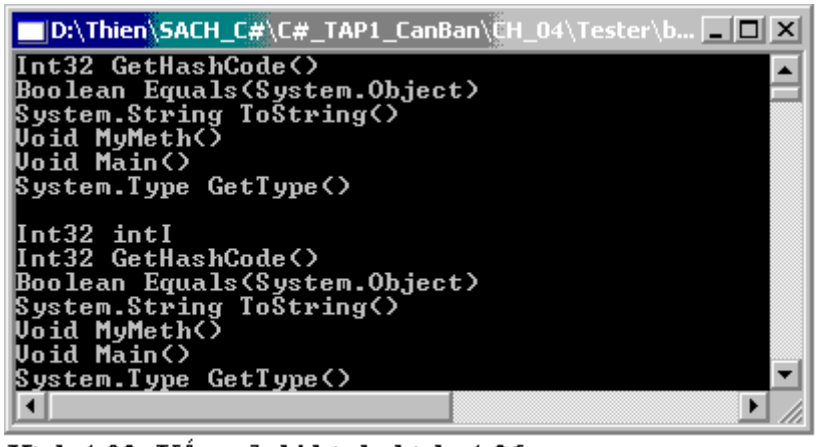
```
using System;
using System.Reflection;

public class MyClass
{
  public int intI;
  public void MyMeth()
  {
  }
```

```
  public static void Main()
  {
   Type t = typeof(MyClass);

   // một phương án khác là bạn có thể dùng các lệnh sau đây
   // MyClass t1 = new MyClass();
   // Type t = t1.GetType();

   MethodInfo[] x = t.GetMethods();
   foreach (MethodInfo xtemp in x)
   {
     Console.WriteLine(xtemp.ToString());
   }

   Console.WriteLine();

   MemberInfo[] x2 = t.GetMembers();
   foreach (MemberInfo xtemp2 in x2)
   {
     Console.WriteLine(xtemp2.ToString());
   }
  }
}
```

Hình 4-2 cho thấy kết quả thi hành thí dụ trên.

4.6.7.4 Tác tử checked và unchecked

Tác tử **checked** và **unchecked** cho phép bạn khai báo cách CLR sẽ xử lý việc tràn năng trên stack (stack overflow) như thế nào, khi bạn thực hiện những tác vụ tính toán trên những kiểu dữ liệu số nguyên. Trong phạm vi **checked,** hiện tượng tràn năng số học sẽ gây ra biệt lệ (exception), còn trong phạm vi unchecked, thì tràn năng số học sẽ bị bỏ qua và kết quả sẽ bị xén bớt (truncated).



Hình 4-02: Kết quả thi hành thí dụ 4-26

Bạn có thể tăng cường việc kiểm tra vượt năng đối với những đoạn mã không đánh dấu trong chương trình bằng cách biên dịch với mục chọn **/checked**.

Từ chốt **checked** có thể được dùng theo dạng câu lệnh hoặc theo dạng tác tử:

checked *block* // dạng câu lệnh
checked (*expression***)** // dạng tác tử

với *block* là khối câu lệnh chứa những biểu thức cần được định trị trong một phạm vi checked, còn *expression* là biểu thức cần được định trị trong một phạm vi checked. Bạn để ý biểu thức phải nằm trong cặp dấu (). Thí dụ dùng checked trên một khối câu lệnh:

```
byte b = 255; // trị tối đa của một byte
checked       // dạng câu lệnh
{
    b++;
}
Console.WriteLine(b.ToString());
```

Khi ta cho biên dịch đoạn mã trên ta sẽ nhận thông điệp sai lầm như sau:

```
Unhandled Exception:System.OverflowException: An exception of type
System.OverflowException was thrown.
  at OverFlowTest.myMethodCh()
```

Sau đây là một thí dụ dùng **checked** như là một tác tử trên một biểu thức, mà ta thấy trước đây:

```
  static short x = 32767;  // Trị tối đa của short
```

```
static short y = 32767;

// Sử dụng một biểu thức checked
public static int myMethodCh()
{
    int z = 0;
    try
    {
       z = checked((short)(x + y));
    }
    catch (System.OverflowException e)
    {
       System.Console.WriteLine(e.ToString());
    }
    return z;  // Một biệt lệ sẽ được tung ra
}
```

Nếu bạn muốn bỏ qua việc kiểm tra vượt năng, bạn có thể đánh dấu đoạn mã là **unchecked** như sau:

```
byte b = 255; // trị tối đa của một byte
unchecked     // dạng câu lệnh
{
   b++;
}
Console.WriteLine(b.ToString());
```

Trong trường hợp này, sẽ không có biệt lệ xảy ra, nhưng chúng ta lại bị mất dữ liệu vì một byte không thể chứa quá 255, nên những gì vượt năng sẽ bị bỏ qua, biến **b** sẽ chứa zero.

4.6.8 Qui tắc "tôn ti trật tự" (Operator Precedence)

Trình biên dịch phải biết thứ tự khi định trị một loạt các tác tử. Thí dụ, khi ta viết:

```
myVariable = 5 + 7 * 3;
```

có 3 tác tử mà trình biên dịch phải định trị (=, +, *). Thí dụ, trình biên dịch có thể hoạt động từ trái qua phải, theo đấy gán trị 5 cho **myVariable**, sau đó cộng thêm 7 (= 12), rồi nhân cho 3 (=36) rồi sau đó vất đi 36. Đây chính không phải ta muốn thế.

Qui tắc "tôn ti trật tự" (rule of precedence) bảo cho trình biên dịch biết tác tử nào phải được định trị trước. Như trong trường hợp đại số, phép nhân có trật tự cao hơn phép

cộng, do đó 5 + 7 * 3 bằng 26, chứ không phải 36. Cả hai nhân và cộng có trật tự cao hơn gán, do đó **myVariable** sẽ nhận 26 khi tính toán xong.

Trên C#, dấu ngoặc () cũng được dùng để thay đổi tôn ti trật tự giống như ở đại số. Do đó, bạn có thể thay đổi kết quả bằng cách dùng () như sau:

```
myVariable = (5 + 7) * 3;     // kết quả bằng 36
```

Bảng 4-7 tóm lược qui tắc tôn ti trật tự các tác tử trên C#:

Bảng 4-7: Qui tắc tôn ti trật tự các tác tử

Loại	**Tác tử**
Primary	(x) x.y f(x) a[x] x++ x-- `new typeof sizeof checked unchecked`
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

Trong vài phương trình phức tạp, có thể phải cho lồng dấu ngoặc bảo đảm đúng trật tự của các tác vụ. Giả định ta muốn biết mất bao nhiêu phí phạm trong gia đình mỗi buổi sáng. Xem ra người lớn mất 20 phút uống cà phê ăn sáng, và 10 phút đọc báo, còn con nít mất 30 phút chơi giởn lông bông và 10 phút gây gổ với nhau. Sau đây là giải thuật:

```
(((minDrinkingCoffee + minReadingNewspaper ) * numAdults) +
((minDawdling + minArguing) * numChildren)) * secondsPerminute.
```

Mặc dù giải thuật chạy đấy, nhưng khó đọc và khó làm cho chạy. Tốt hơn là nên sử dụng những biến trung gian:

```
secondsPerMinute = 60;
wastedByEachAdult = minDrinkingCoffee + minReadingNewspaper;
wastedByAllAdults = wastedByEachAdult * numAdults;
```

```
wastedByEachKid = minDawdling + minArguing;
wastedByAllKids = wastedByEachKid * numChildren;
wastedByFamily = wastedByAllAdults + wastedByAllKids;
totalSeconds = wastedByFamily * secondsPerminute.
```

Thí dụ trên sử dụng nhiều biến trung gian, nhưng dễ đọc dễ hiểu hơn.

4.6.9 Tác tử tam phân (ternary operator)

Mặc dù phần lớn các tác tử chỉ cần đến một toán hạng (term), chẳng hạn **myValue++**, hoặc hai toán hạng, chẳng hạn **a + b**, nhưng có một tác tử có đến 3 toán hạng, ta gọi là ***tác tử tam phân*** (ternary operator), đó là **?:,** mang cú pháp sau đây:

condition-expression **?** *expression1*: *expression2*

Tác tử này cho định trị một biểu thức điều kiện (*condition-expression*), là một biểu thức trả về một trị bool (true/false), rồi sau đó triệu gọi *expression1* nếu trị trả về là true, hoặc *expression2* nếu trị trả về là false. Lôgic của tác tử này là: "nếu là true làm cái đầu tiên, nếu là false thì làm cái sau. Thí dụ 4-27, minh hoạ việc sử dụng loại tác tử này:

Thí dụ 4-27: Sử dụng tác tử tam phân

```
using System;

class Values
(   static void Main()
    {   int triMôt = 10;
        int triHai = 20;
        int maxValue = triMôt > triHai ? triMôt: triHai;
        Console.WriteLine("TriMôt: {0}, TriHai: (1), maxValue: {2}",
            triMôt, triHai, maxValue);
    }   // end Main
}   // end Values
```

Kết xuất
TriMôt: 10, TriHai: 20, maxValue: 20

4.7 Địa bàn hoạt động các tên (Namespaces)

Trong chương 1 & 2, chúng tôi đã đề cập đến lý do đưa namespace vào C# (nghĩa là để tránh đụng độ khi sử dụng các thư viện mua từ nhiều nhà sản xuất phần mềm khác nhau. Ngoài việc dùng những namespace mà .NET Framework cung cấp, hoặc của các nhà cung cấp khác, bạn có thể tự mình tạo namespace riêng cho mình. Muốn thế, bạn dùng từ chốt **namespace** theo sau là tên bạn muốn đặt cho. Bạn cho bao trong cặp dấu {} tất cả các đối tượng thuộc namespace này, như theo thí dụ 4-28 sau đây:

Thí dụ 4-28: Tạo namespaces

```
namespace Prog_CSharp
{   using Systems;
    public class Tester
    {   public static int Main()
        {   for (int i = 0; i < 100; i++)
            {   Console.WriteLine("i: {0}", i);
            }   // end for
            return 0;
        }   // end Main
    }   // end Tester
}   // end Prog_CSharp
```

Thí dụ 4-28 tạo ra một namespace mang tên **Prog_CSharp**, với lớp **Tester** "cư ngụ" trong lòng namespace này. Bạn cũng có thể cho nằm lồng nhau những namespace nếu thấy cần thiết. Có thể bạn muốn những đoạn (segment) nhỏ trên đoạn mã của bạn, tạo những đối tượng trong lòng một namespace được nằm lồng theo đấy các tên sẽ được bảo vệ từ namespace nằm vòng ngoài, như theo thí dụ 4-29 minh hoạ sau đây:

Thí dụ 4-29: Namespaces nằm lồng nhau

```
namespace Prog_CSharp
{
    namespace Prog_CSharp_Test
    {   using Systems;
        public class Tester
        {   public static int Main()
            {   for (int i = 0; i < 100; i++)
                {   Console.WriteLine("i: {0}", i);
                }   // end for
                return 0;
            }   // end Main
        }   // end Tester
```

```
    }   // Prog_CSharp_Test
}   // end Prog_CSharp
```

Đối tượng **Tester** giờ đây được khai báo trong lòng namespace **Prog_CSharp_Test** là:

```
Prog_CSharp.Prog_CSharp_Test.Tester
```

tên này sẽ không xung khắc với một đối tượng **Tester** khác trên bất cứ namespace nào, kể cả namespace Prog_CSharp nằm ngoài.

4.7.1 Namespace Aliases

Một cách sử dụng khác từ chốt **using** là gán những "bí danh" (alias) cho các lớp và namespace. Nếu bạn có một namespace dài lê thê mà bạn muốn qui chiếu nhiều chỗ trên đoạn mã, và bạn lại không muốn bao gồm trong một chỉ thị using (thí dụ để tránh xung đột về tên), bạn có thể gán một alias cho namespace. Cú pháp như sau:

using *alias* = *NamespaceName*;

Thí dụ sau đây gán alias Chapter04 cho Samis.Prog_CSharp, và dùng để hiển lộ một đối tượng NamespaceExample, được định nghĩa trong namespace này. Đối tượng này chỉ có một hàm hành sự **GetNamespace()**, sử dụng đến hàm hành sự **GetType()** mà mọi lớp đều có để truy xuất một đối tượng **Type** tượng trưng cho kiểu dữ liệu của lớp. Chúng tôi dùng đối tượng này để trả về tên namespace của lớp.

```
using System;
using Chapter04 = Samis.Prog_CSharp.Chapter04; // Chapter04 là alias
class Tester
{
    public static int Main()
    {
        Chapter04.NamespaceExample NSEx = new
                                        Chapter04.NamespaceExample();
        Console.WriteLine(NSEx.GetNamespace());
        return 0;
    }
}

namespace Samis.Prog_CSharp.Chapter04
{
    class NamespaceExample
    {
        public string GetNamesapce()
        {
            return this.GetType().Namespace;
        }
```

```
    }
}
```

4.8 Các chỉ thị tiền xử lý (Preprocessor Directives)

Trong các thí dụ mà bạn đã làm quen mãi tới giờ, bạn thường biên dịch toàn bộ chương trình bất cứ lúc nào khi bạn biên dịch. Tuy nhiên, đôi lúc có thể bạn muốn chỉ biên dịch một phần nhỏ chương trình mà thôi tùy theo bạn đang gỡ rối hoặc đang xây dựng đoạn mã sản xuất.

Trước khi bạn biên dịch, có một chương trình khác được gọi là **preprocessor**, *bộ tiền xử lý*, sẽ chạy và lo chuẩn bị chương trình của bạn trước khi đem biên dịch. Bộ preprocessor sẽ tìm xem những chỉ thị tiền xử lý (preprocessor directive) được đánh dấu bởi ký hiệu # (pound sign). Những chỉ thị này cho phép bạn định nghĩa những diện từ (identifier) rồi trắc nghiệm sự hiện diện của chúng.

4.8.1 Định nghĩa những diện từ

#define DEBUG cho định nghĩa một diện từ tiền xử lý (preprocessor identifier) mang tên **DEBUG**. Mặc dù những chỉ thị từ tiền xử lý khác có thể tới bất cứ nơi nào trong đoạn mã của bạn, các diện từ phải được khai báo trước bất cứ đoạn mã khác, kể cả câu lệnh **using**.

Bạn có thể trắc nghiệm liệu xem **DEBUG** đã được định nghĩa hay chưa thông qua câu lệnh **#if**. Do đó, bạn có thể viết:

```
#define DEBUG

    // … vài đoạn mã bình thường - không bị ảnh hưởng bởi preprocessor

#if DEBUG
    // đoạn mã phải bao gồm nếu có gỡ rối (DEBUG )
#else
    // đoạn mã không cho bao gồm vào nếu không có gỡ rối
#endif

    //… vài đoạn mã bình thường - không bị ảnh hưởng bởi preprocessor
```

Khi preprocessor chạy, nó thấy câu lệnh **#define** và ghi nhận diện từ **DEBUG**. Preprocessor nhảy bỏ đoạn mã bình thường cho đến khi gặp khối **#if - #else - #endif**.

Câu lệnh **#if** trắc nghiệm tìm diện từ **DEBUG**, hiện có mặt, do đó đoạn mã nằm giữa **#if** và **#else** sẽ được biên dịch trong chương trình của bạn, nhưng đoạn mã giữa **#else** và **#endif** thì lại *không* được biên dịch. Đoạn mã này sẽ không xuất hiện trong assembly của bạn, như là bị bỏ ngoài đoạn mã của bạn.

Nếu **#if** thất bại trong việc trắc nghiệm sự hiện diện của **DEBUG**, nghĩa là bạn trắc nghiệm tìm một diện từ không có, thì đoạn mã nằm **#if** và **#else** sẽ không được biên dịch, nhưng đoạn mã giữa **#else** và **#endif** thì lại được biên dịch.

4.8.2 Không định nghĩa những diện từ

Bạn không định nghĩa một diện từ thông qua **#undef**. Preprocessor hoạt động xuyên qua chương trình của bạn từ đầu đến cuối, như vậy diện từ được định nghĩa từ **#define** cho tới khi gặp phải **#undef** hoặc cho tới cuối chương trình. Như vậy, khi bạn viết:

```
#define DEBUG

#if DEBUG
   // đoạn mã này sẽ được biên dịch
#endif

#undef DEBUG

#if DEBUG
   // đoạn mã này sẽ không được biên dịch
#endif
```

#if đầu tiên sẽ thành công (DEBUG được định nghĩa), nhưng **#if** thứ hai thất bại (DEBUG đã bị hoá giải định nghĩa).

4.8.3 Các chỉ thị #if, #elif, #else, và #endif

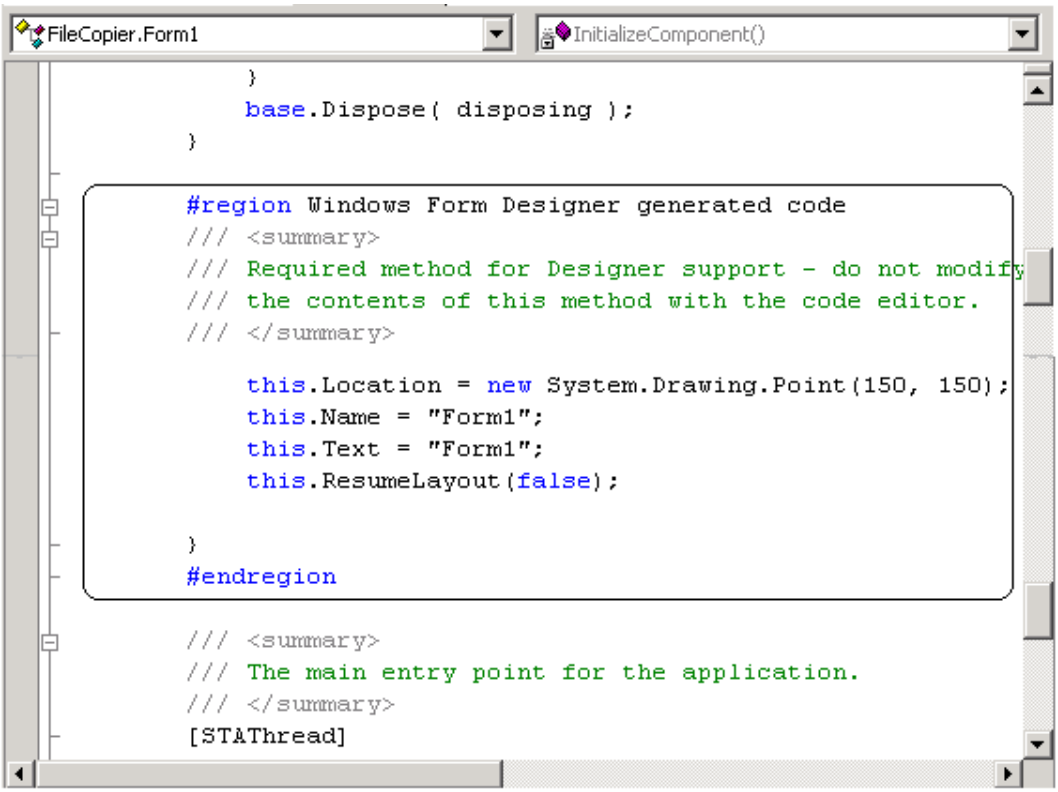
Không có lệnh **switch** đối với preprocessor, nhưng các chỉ thị **#elif** và **#else** cung cấp mức độ uyển chuyển khá lớn. Chỉ thị **#elif** cho phép lôgic else-if "nếu DEBUG thì hành động số 1, bằng không nếu TEST thì hành động số 2, bằng không thì hành động số 3":

```
#if DEBUG
   // biên dịch đoạn mã này nếu debug được định nghĩa
#elif TEST
   // biên dịch đoạn mã này nếu debug không được định nghĩa
   // nhưng TEST được định nghĩa
#else
   // biên dịch đoạn mã này nếu cả debug lẫn test
   // không được định nghĩa
#endif
```

Trong thí dụ này, preprocessor trước tiên trắc nghiệm xem DEBUG đã được định nghĩa chưa, nếu rồi thì đoạn mã giữa **#if** và **#elif** sẽ được biên dịch, và phần còn lại cho tới **#endif** sẽ không được biên dịch.

Nếu (và chỉ nếu) DEBUG không được định nghĩa, preprocessor cho kiểm tra kế tiếp xem TEST có được định nghĩa hay không. Bạn để ý là preprocessor sẽ không kiểm tra TEST trừ khi DEBUG không được định nghĩa. Nếu TEST đã được định nghĩa, thì đoạn mã nằm giữa **#elif** và **#else** sẽ được biên dịch. Nếu xem cả DEBUG lẫn TEST không được định nghĩa, thì đoạn mã nằm giữa **#else** và **#endif** sẽ được biên dịch.

4.8.4 Chỉ thị #region

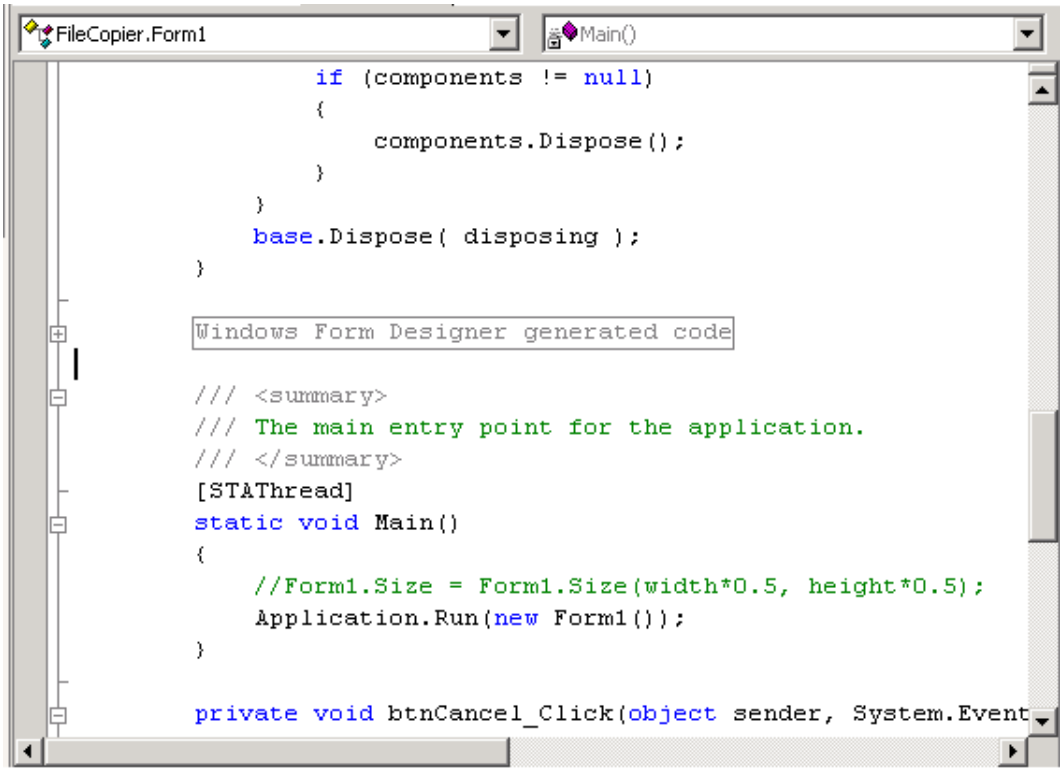


Hình 4-3: Bung Visual Studio .NET code region

Chỉ thị preprocessor **#region** đánh dấu một vùng văn bản với một chú thích (comment). Công dụng chủ yếu của chỉ thị này cho phép những công cụ chẳng hạn

Visual Studio .NET đánh dấu một vùng trên đoạn mã và cho teo lại (collapse) trên editor chỉ còn lại một chú thích liên quan đến vùng.

Thí dụ, khi bạn tạo một ứng dụng Windows (sẽ được đề cập ở tập 2, chương 2 bộ sách này), Visual Studio .NET sẽ tạo một vùng dành cho đoạn mã được kết sinh (generated) bởi trình designer. Khi vùng này được bành trướng (expand) ra, nó giống như trên hình 4-3. Bạn có thể thấy vùng này được đánh dấu bởi **#region** và **#endregion**. Khi vùng bị teo lại (collapsed) thì bạn thấy chú thích (**Windows Form Designer generated code**) như theo Hình 4-4.



Hình 4-4 : Code region bị teo lại

Chương 5

Lớp và Đối tượng

Chương 4 đi trước đã đề cập đến vô số kiểu dữ liệu "bẩm sinh" (built-in type) mà C# có thể cung cấp cho bạn, chẳng hạn **int, long** và **char**. Tuy nhiên, "linh hồn" của C# là khả năng tạo ra được những kiểu dữ liệu tự tạo (user-defined type - UDT) mới, phức tạp cho phép áp (map) một cách "sạch sẽ" hơn lên các đối tượng thuộc vấn đề bạn đang giải quyết.

Chính khả năng tạo ra những kiểu dữ liệu mới là đặc tính của một ngôn ngữ thiên đối tượng. Bạn khai báo những kiểu dữ liệu mới trên C# bằng cách khai báo và định nghĩa những **lớp** (class). Bạn cũng có thể sử dụng giao diện (interface) để định nghĩa kiểu dữ liệu, như bạn sẽ thấy ở chương 9. Những thể hiện (instance) của một lớp được gọi là ***đối tượng***. Các đối tượng chỉ sẽ được tạo ra trong ký ức, khi chương trình của bạn thi hành.

Khác biệt giữa một lớp với một đối tượng của nó cũng giống như khác biệt giữa khái niệm (concept) của một lớp **Dog** và một con chó đặc biệt nào đó đang nằm cạnh chân bạn. Bạn không thể chơi trò tìm bắt với định nghĩa của một **Dog**, mà chỉ với một thể hiện.

Một lớp **Dog** mô tả loại chó giống như thế nào: chó thường có một trọng lượng, một chiều cao, một màu mắt, một màu lông, một cái mõm, v.v..Ngoài ra, chó cũng có những hành động như ăn uống, sủa, chạy nhảy, và ngủ. Một con chó đặc biệt (như con Titi của bạn chẳng hạn), có trọng lượng 62 pounds, chiều cao 22 inches, màu mắt đen, màu lông vàng, v.v.. Con chó này còn có những hành động thông thường của bất cứ con chó nào, nhưng đặc biệt nó có thể làm xiếc chẳng hạn.

Lợi điểm to lớn của lớp trong lập trình thiên đối tượng là lớp cho "đóng gói" hoặc "gói ghém" (encapsulate) những đặc tính và khả năng của một thực thể (entity) vào trong một ***đơn vị đoạn mã*** (unit of code) đơn lẻ và "tự cung tự cấp" (self-contained). Thí dụ, khi bạn muốn cho sắp xếp nội dung của một thể hiện (đối tượng) của một ô liệt kê (list box) Windows, bạn bảo ô liệt kê tự mình sắp xếp lấy. Ô này làm thế nào, bạn bất cần biết đến, miễn là nó làm đúng theo sự chờ đợi của bạn. Khả năng "gói ghém", cùng với khả năng ***đa hình*** (polymorphisme) và ***tính thừa kế*** (inheritance), là một trong 3 hòn đá tảng của lập trình thiên đối tượng.

Chương này cố gắng giải thích những chức năng của ngôn ngữ C# mà bạn sẽ dùng đến để khai báo những lớp mới. Những phần tử cấu thành của một lớp - lối hành xử (behavior) và những thuộc tính (property) - được gọi là những ***thành viên lớp*** (class

member). Chương này sẽ chỉ cho bạn làm thế nào các hàm hành sự (method) được dùng để mô tả cách hành xử của lớp, và làm thế nào trạng thái (hoặc tình trạng, state) của lớp sẽ được duy trì thông qua những ***biến thành viên*** (member variable), thường được gọi là ***vùng mục tin*** (field[33]). Ngoài ra, chương này sẽ dẫn nhập vào những ***thuộc tính***, hoạt động tương tự như những hàm hành sự đối với người tạo ra lớp nhưng lại giống như những vùng mục tin đối với người sử dụng lớp.

5.1 Định nghĩa lớp

Muốn định nghĩa một kiểu dữ liệu (type) hoặc lớp mới, trước tiên bạn phải khai báo lớp rồi sau đó định nghĩa những hàm hành sự và các vùng mục tin. Bạn dùng từ chốt **class** để khai báo một lớp mới, theo cú pháp sau đây:

[***attributes***] [***access-modifiers***] **class** *identifier* [**:*base-class***]
{***class-body***}

Attributes sẽ được đề cập ở Chương 4 tập 2 bộ sách này: *access-modifiers* sẽ được đề cập trong phần kế tiếp. Phần lớn lớp của bạn sẽ dùng từ chốt **public** như là một access-modifier (truy xuất lớp thế nào). Mục *identifier* là tên lớp do bạn đặt cho. Mục tự chọn (optional) *base-class*, cho biết lớp cơ bản, sẽ được đề cập ở chương 6, "Tính kế thừa và tính đa hình". Còn những định nghĩa các thành viên lớp hình thành thân lớp *class-body* sẽ được đặt nằm trong cặp dấu ngoặc nghéo **{}** (curly brace).

Bạn nhớ cho là trên C#, tất cả mọi việc đều diễn ra trong các lớp. Dân lập trình viên C# phải quen bị ám ảnh bởi lớp, ăn với lớp, ngủ với lớp cũng như mơ với lớp. Thí dụ, trong chương đi trước, bạn đã làm quen với lớp **Tester** dùng trắc nghiệm chương trình:

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

Tới lúc này, ta vẫn chưa thực hiện bất cứ thể hiện nào của lớp **Tester** này: nghĩa là ta chưa tạo ra bất cứ đối tượng **Tester** nào. Khác biệt giữa một lớp và một thể hiện lớp này là gì?. Để trả lời câu hỏi trên, ta thử phân biệt giữa *kiểu dữ liệu* (type) **int** và một *biến* kiểu dữ liệu **int**. Do đó, trong khi bạn có thể viết:

[33] Chúng tôi không dịch là 'trường' như trường đua ngựa Phú Thọ, như nhiều tác giả đã dịch.

```
int myInteger = 5;
```

thì bạn không thể viết được:

```
int = 5;
```

Nói cách khác, bạn không thể gán một trị cho một kiểu dữ liệu: bạn *chỉ có thể gán một trị cho một đối tượng thuộc kiểu dữ liệu này*, ở đây là một biến kiểu **int**.

Khi bạn khai báo "đẻ ra" một lớp mới, bạn sẽ định nghĩa những thuộc tính của tất cả các đối tượng thuộc lớp này, cũng như những cách hành xử của các đối tượng này. Thí dụ, nếu bạn đang tạo ra một môi trường window, có thể bạn muốn tạo những ô điều khiển (control), giúp người sử dụng dễ dàng tương tác với chương trình. Một trong những ô control này là listbox control (ô liệt kê), một loại bảng liệt kê kê ra một danh sách những chọn lựa cho phép người sử dụng chọn ra từ bảng này.

Listbox control này có vô số đặc tính: kích thước (chiều cao, chiều dài), vị trí và màu sắc phần văn bản (text color). Ngoài ra, lập trình viên cũng chờ đợi một số hành xử (behavior) từ phía ô liệt kê: nghĩa là ta có thể mở, đóng, sắp xếp một ô liệt kê v.v..

Lập trình thiên đối tượng cho phép bạn tạo ra một kiểu dữ liệu mới, đặt tên là **ListBox** (hoặc **OLietKe** nếu bạn muốn viết theo tiếng Việt), cho gói ghém lại những khả năng và những đặc tính. Một lớp **ListBox** như thế có thể có những biến thành viên mang tên **height, width, location** và **textColor** và những hàm hành sự mang tên **add(), remove()** và **sort()** để thêm, gỡ bỏ và sắp xếp các mục tin trên ô liệt kê.

Bạn không thể gán dữ liệu cho kiểu dữ liệu **ListBox**. Mà trước tiên, bạn phải tạo ra một đối tượng kiểu **ListBox**, như theo dòng lệnh sau đây:

```
ListBox myListBox;      // myListBox là thể hiện một ô liệt kê
```

Lệnh trên tạo một *thể hiện* của lớp **ListBox.** Một khi đã tạo xong thể hiện này, bạn mới có thể gán dữ liệu cho những vùng mục tin.

Bây giờ ta thử xét đến một lớp lo theo dõi thời gian trong ngày và cho in ra. Các vùng mục tin nội bộ của lớp này có khả năng biểu diễn ngày, tháng, năm, giờ, phút và giây. Ngoài ra, bạn cũng muốn lớp cho hiển thị thời gian theo nhiều dạng thức (format) khác nhau. Bạn có thể thiết đặt một lớp như thế, bằng cách định nghĩa một hàm hành sự duy nhất với 6 biến thành viên như sau:

Thí dụ 5-1: Lớp Time đơn giản

```
using System;
```

```
public class Time
{
    // các hàm hành sự public
    public void DisplayCurrentTime()
    {   Console.WriteLine("Phần dành cho DisplayCurrentTime,
                          sẽ viết sau");
    }

    // các biến private
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;
}   // end Time

public class Tester
{
    static void Main()
    {
        Time t = new Time();    // t là đối tượng kiểu Time
        t.DisplayCurrentTime();
    }   // end Main
}   // end Tester
```

Hàm hành sự duy nhất trong phần định nghĩa của lớp **Time** là hàm **DisplayCurrent Time**, lo hiển thị thời gian. Thân hàm hành sự được định nghĩa ngay trong lòng định nghĩa của lớp. Khác với các ngôn ngữ khác, chẳng hạn C++, C# không đòi hỏi được định nghĩa trước, cũng không đòi hỏi ngôn ngữ hỗ trợ phải đặt những khai báo hàm trong một tập tin, và đoạn mã của hàm trong một tập tin khác: nghĩa là C# không có tập tin tiêu đề (header file) giống như trên C++. Tất cả các hàm hành sự C# đều được định nghĩa giống như **DisplayCurrentTime** trên thí dụ 5-1.

DisplayCurrentTime được định nghĩa trả về **void**, nghĩa là nó sẽ không trả về một trị cho hàm đã triệu gọi nó. Hiện thời thân hàm chỉ vỏn vẹn in ra dòng chữ: `Phần dành cho DisplayCurrentTime, sẽ viết sau.`

Phần định nghĩa lớp **Time** kết thúc bởi việc khai báo một số biến thành viên: **Year, Month, Date, Hour, Minute** và **Second**.

Sau dấu } khép lại , một lớp thứ hai được khai báo, **Tester**. Lớp này chứa hàm **Main()** khá quen thuộc. Trong **Main()**, một thể hiện lớp **Time** được tạo ra và vị chỉ của nó được gán cho đối tượng **t**. Vì **t** là một thể hiện của **Time**, **Main()** có thể sử dụng hàm hành sự **DisplayCurrentTime** có sẵn đối với những đối tượng kiểu này, và triệu gọi hàm này để cho hiển thị thời gian:

```
t.DisplayCurrentTime();
```

5.1.1 Từ chốt hướng dẫn truy xuất (Access Modifiers)

Một ***từ chốt hướng dẫn truy xuất*** (*access modifier)* xác định cho biết những hàm hành sự nào trong lớp - kể cả những hàm hành sự của các lớp khác - có thể thấy và sử dụng một biến thành viên hoặc hàm hành sự trong lòng một lớp. Bảng 5-1 sau đây tóm lược những access modifier của C#.

Bảng 5-1: Các từ chốt access modifier

Access Modifier	Hạn chế
public	*Công cộng*. Không có giới hạn. Các thành viên được đánh dấu **public** có thể được nhìn thấy (visible) bởi bất cứ hàm hành sự nào của bất cứ lớp nào.
private	*Riêng tư*. Các thành viên nào trên lớp A được đánh dấu **private** thì chỉ có thể được truy xuất bởi những hàm hành sự của lớp A mà thôi. Các hàm trên các lớp khác không thể truy cập được.
protected	*Được bảo vệ*. Các thành viên nào trên lớp A được đánh dấu **protected** chỉ có thể được truy xuất bởi những hàm hành sự của lớp A, cũng như bởi những hàm hành sự của những lớp được *dẫn xuất* (derived) từ lớp A, mà thôi. Thật tình, từ protected không nói lên điều gì.
internal	Các thành viên nào trên lớp A được đánh dấu **internal** chỉ có thể được truy xuất bởi những hàm hành sự của bất cứ lớp nào trong assembly của A mà thôi.
protected internal	Các thành viên nào trên lớp A được đánh dấu **protected internal** chỉ có thể được truy xuất bởi những hàm hành sự của lớp A, cũng như bởi những hàm hành sự của những lớp được *dẫn xuất* (derived) từ lớp A, cũng như bởi bất cứ lớp nào trong assembly của A mà thôi. Thật ra đây là **protected** OR **internal**.

Thông thường, người ta đề nghị nên cho các biến thành viên của một lớp về **private**. Đây có nghĩa là chỉ những hàm thành viên thuộc lớp mới có thể truy xuất trị các biến này. Vì **private** là cấp truy xuất mặc nhiên (default), nên bạn có thể không viết ra cũng được, nhưng theo tập quán lập trình tốt, ta nên viết rõ ra **private**. Do đó, trong thí dụ 5-1, những khai báo biến thành viên phải được viết như sau:

```
// các biến private
private   int Year;
private   int Month;
private   int Date;
private   int Hour;
```

```
private   int Minute;
private   int Second;
```

Lớp **Tester** và hàm **DisplayCurrentTime** cả hai đều được khai báo là **public** do đó bất cứ lớp nào khác cũng có thể sử dụng chúng.

5.1.2 Các đối mục hàm hành sự (Method Arguments)

Các hàm hành sự có thể nhận bao nhiêu thông số (parameter) hoặc đối mục (argument) cũng được. Từ parameter và argument[34] có thể dùng lẫn lộn không phân biệt. Danh sách các thông số (parameter list) theo sau tên hàm và được đặt nằm trong cặp dấu ngoặc (parentheses), (): mỗi thông số có kèm theo một kiểu dữ liệu nằm trước. Thí dụ, khai báo sau đây định nghĩa một hàm hành sự mang tên **MyMethod()** trả về **void** (nghĩa là gì đó được trả về không có ý nghĩa gì cả) và nhận hai thông số: một kiểu **int**, một kiểu **button**:

```
void MyMethod (int FirstParam, button secondParam)
{
    // …
}
```

Trong lòng thân hàm, các thông số hoạt động như là những biến cục bộ (local variable), xem như chúng được khai báo trong thân của hàm, và cho khởi gán (initialized) chúng bởi những trị mà ta chuyển qua. Thí dụ 5-2, minh hoạ cho thấy cách chuyển các trị cho một hàm hành sự: trong trường hợp này trị có kiểu **int** và **float.**

Thí dụ 5-2: Chuyển trị vào hàm nào đó

```
using System;

public class MyClass
(
    public void SomeMethod(int firstParam, float secondParam)
    {
        Console.WriteLine("Sau đây là các thông số nhận được:
                           {0}, {1}", firstParam, secondParam);
    }
}   // end MyClass

public class Tester
{
```

[34] Tuy nhiên cũng có người thích phân biệt: argument trong khai báo hàm, còn parameter trong khi trao cho triệu gọi hàm.

```
    static void Main()
    {
        int howManyPeople = 5;
        float pi = 3.14;
        MyClass mc = new MyClass();
        mc.SomeMethod(howManyPeople, pi);
    }  // end Main
}  // end Tester
```

Hàm hành sự **SomeMethod()** nhận hai thông số một **int** và một **float** và cho hiển thị nội dung của chúng thông qua **Console.WriteLine()**. Các thông số **firstParam** và **secondParam** được xem như là những biến cục bộ trong lòng **SomeMethod()**.

Trong hàm phía triệu gọi (calling method) **Main()**. 2 biến cục bộ, **howManyPeople** và **pi** được tạo và được khởi gán. Các biến này được chuyển cho **SomeMethod()** như là những thông số. Trình biên dịch sẽ ánh xạ (map) **howManyPeople** với **firstParam** và **pi** với **secondParam** dựa trên vị trí tương đối của chúng trên danh sách các thông số.

5.2 Tạo các đối tượng

Trong chương 4, ta đã phân biệt dữ liệu kiểu trị và kiểu qui chiếu. Những kiểu dữ liệu bẩm sinh C#, như **int, char** chẳng hạn, thuộc kiểu trị, nên được tạo trên ký ức stack. Tuy nhiên, những đối tượng lại thuộc kiểu qui chiếu, nên được tạo trên ký ức heap, và sử dụng từ chốt **new,** như theo dòng lệnh sau đây:

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

t, thường được gọi là *biến đối tượng* (object variable), hiện không chứa nội dung của đối tượng **Time**, mà là vị chỉ của đối tượng không tên được tạo ra trên ký ức heap. **t** đơn giản chỉ là một qui chiếu chĩa về đối tượng này.

5.2.1 Hàm khởi dựng (constructor)

Trong thí dụ 5-1, bạn để ý lệnh tạo đối tượng **Time**, hình như đang triệu gọi một hàm hành sự:

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

Đúng thế, một hàm hành sự *được* triệu gọi bất cứ lúc nào bạn cho thể hiện một đối tượng. Hàm này mang một tên đặc biệt là ***hàm khởi dựng*** *(constructor)*, và bạn phải hoặc định nghĩa một như là thành phần của định nghĩa lớp, hoặc để cho Common Language Runtime (CLR) cung cấp một mặc nhiên (gọi là default constructor) nhân danh bạn.

Công năng của hàm constructor là tạo một đối tượng được ấn định bởi lớp và cho đối tượng này ở tình trạng *hợp lệ*, "sẵn sàng tác chiến". Trước khi hàm constructor chạy, đối tượng nằm trong ký ức "bất định", còn sau khi hàm constructor chạy xong, ký ức trữ một thể hiện hợp lệ của lớp **Time**.

Bạn để ý: Đối với những bạn lần đầu tiên mới làm quen với lập trình thiên đối tượng (OOP, object oriented programming), từ ngữ "hàm khởi dựng"có vẻ rất lạ lẫm. Để dễ hình dung công năng của hàm này, bạn có thể tưởng tượng bạn mở một quán cà phê nhỏ trước nhà để kiếm đôi chút tiền còm. Sáng ra, trước khi khách kéo đến, bạn phải quét sân, dọn bàn ghế ra, nấu sẵn một bình nước sôi, và những việc linh tinh khác như thắp một nén nhang vái tứ phương cầu mong nhiều khách trong ngày. Tất cả những việc này, bạn thực hiện một lần thôi vào đầu ngày, mà ta thường gọi là "mở cửa hàng". Cuối ngày bạn làm những động tác ngược lại được gọi là "dọn đóng cửa hàng". Trong OOP, thì hàm constructor làm công việc "mở cửa hàng", còn hàm destructor (chúng tôi dịch là hàm hủy) lại làm công việc "dọn đóng cửa hàng". Thật ra, khái niệm hàm constructor và destructor chả có chi là mới mẻ. Trong lập trình, ta có hai từ tương đương là setup và clean up. Nhưng vì là OOP, để nghe cho oai nên người ta mới cho ra hai từ constructor và destructor. Đúng là bình mới rượu cũ.

Lớp **Time**, như theo thí dụ 5-1, không có định nghĩa một hàm constructor. Nếu hàm constructor không được khai báo, trình biên dịch sẽ tự động cung cấp một hàm constructor mặc nhiên cho bạn. Hàm constructor mặc nhiên tạo một đối tượng, nhưng sẽ không hành động gì cả. Các biến thành viên sẽ được khởi gán về những trị thích ứng với kiểu dữ liệu (số nguyên cho về zero, chuỗi cho về chuỗi rỗng, v.v..). Bảng 5-2 cho liệt kê những trị mặc nhiên được gán cho những kiểu dữ liệu bẩm sinh:

Kiểu dữ liệu	**Trị mặc nhiên**
numeric (int, long, v.v.)	0 (zero)
bool	false
char	'\0' (null)
enum	0
reference	null

Điển hình là bạn muốn định nghĩa hàm constructor riêng của bạn và cung cấp cho hàm những đối mục để hàm có thể đặt để trạng thái ban đầu (initial state) đối với đối tượng của bạn. Trong thí dụ 5-1, giả sử bạn muốn trao dữ liệu hiện hành về năm, tháng, ngày v..v.., như vậy đối tượng được tạo ra với dữ liệu có ý nghĩa.

Muốn định nghĩa một hàm constructor, bạn phải khai báo một *hàm hành sự cùng tên với lớp*. Hàm constructor không có trị trả về và thường được khai báo là **public**. Nếu có đối mục phải trao qua, bạn định nghĩa một danh sách các đối mục giống như bạn khai báo các hàm khác. Thí dụ 5-3 sau đây khai báo một hàm constructor đối với lớp **Time** chấp nhận một đối mục đơn lẻ, một đối tượng kiểu **DateTime**.

Thí dụ 5-3: Khai báo một hàm constructor

```
using System;

public class Time
{
    // hàm constructor
    public Time(System.DateTime dt)
    {   Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
    }

    // các hàm hành sự public khác thuộc lớp Time
    public void DisplayCurrentTime()
    {
        Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second);
    }

    // các biến thành viên private
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;
}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
        t.DisplayCurrentTime();
    }  // end Main
}  // end Tester
```

Kết xuất

11/15/2002 16:21:40

Trong thí dụ này, hàm constructor lấy một đối tượng **DateTime** và cho khởi gán tất cả các biến thành viên dựa trên trị của đối tượng này. Khi hàm constructor xong việc, thì đối tượng **Time** hiện hữu và những biến được khởi gán. Khi hàm **DisplayCurrentTime()** được triệu gọi trong **Main()**, các trị được hiển thị.

Bạn thử cho một lệnh gán biến thành viên thành chú giải (comment out, bằng cách thêm ở đầu hai dấu //), bạn sẽ thấy trình biên dịch cho khởi gán biến thành viên về zero. Biến thành viên kiểu **int** sẽ được cho về zero nếu bạn khởi gán nó khác đi. Bạn nên nhớ dữ liệu kiểu trị không thể không được khởi gán (uninitialized); nếu bạn không bảo hàm constructor phải làm gì, thì nó sẽ làm một cái gì đó vô thưởng vô phạt.

Trong thí dụ 5-3, đối tượng **DateTime** được tạo trong hàm **Main()** của lớp **Tester**. Đối tượng này, cung cấp bởi thư viện **System**, cung cấp một số trị public – **Year, Month, Day, Hour, Minute** và **Second** - trực tiếp tương ứng với các biến thành viên của đối tượng **Time.** Ngoài ra, đối tượng **DateTime** còn cung cấp một hàm hành sự static **Now** trả về một qui chiếu về một thể hiện của đối tượng **DateTime** được khởi gán với thời gian hiện hành. Qui chiếu này được gán cho **currentTime**, được khai báo là một qui chiếu chĩa về đối tượng **DateTime**. Sau đó, **currentTime**, được trao cho hàm constructor của **Time** như là thông số. Thông số **dt** trên hàm constructor **Time** cũng là một qui chiếu về đối tượng **DateTime**; thật ra **dt** giờ đây cũng qui chiếu về cùng đối tượng **DateTime** như với **currentTime**. Do đó, hàm constructor **Time** có thể truy xuất các biến thành viên public của đối tượng **DateTime** được tạo ra trong **Tester.Main()**.

Lý do mà một đối tượng **DateTime** được qui chiếu về trong một hàm constructor là cùng đối tượng được qui chiếu về trong **Main()** là vì các đối tượng thuộc loại reference type. Do đó, khi bạn trao một đối tượng như là một thông số, nó sẽ được trao theo qui chiếu (*by reference*), nghĩa là một con trỏ (pointer) được trao qua chứ không phải một bản sao đối tượng được trao qua.

5.2.2 Bộ khởi gán (Initializers)

Ta có khả năng khởi gán các biến thành viên trong một *initializer* (bộ khởi gán), thay vì khởi gán trong mọi hàm constructor. Bạn tạo ra một initializer bằng cách gán một trị ban đầu (initial value) cho một thành viên lớp:

```
private int Second = 30;   // initializer
```

Giả sử ý nghĩa của đối tượng **Time** là bất luận thời gian được đặt để thế nào đi nữa, số giây bao giờ cũng được cho về 30. Có thể ta phải viết lại lớp **Time** dùng đến một initializer để cuối cùng bất cứ hàm constructor nào được triệu gọi thì trị của **Second** bao gồm cũng được khởi gán, hoặc rõ ra bởi hàm constructor hoặc ngầm bởi initializer, như được minh hoạ bởi thí dụ 5-4 sau đây:

Thí dụ 5-4: Sử dụng một initializer

```
using System;

    public class Time
    {
        // hàm constructor thứ nhất có đủ thông số
        public Time(System.DateTime dt)
        {
            Year =    dt.Year;
            Month =   dt.Month;
            Date =    dt.Day;
            Hour =    dt.Hour;
            Minute =  dt.Minute;
            Second =  dt.Second;    // explicit assigment
        }

        // hàm constructor thứ hai này không có thông số Second
        public Time(int Year, int Month, int Date, int Hour, int Minute)
        {
            this.Year =      Year;
            this.Month =     Month;
            this.Date =      Date;
            this.Hour =      Hour;
            this.Minute =    Minute;
        }

        // hàm hành sự public
        public void DisplayCurrentTime()
        {
            System.DateTime homNay = System.DateTime.Now;
            System.Console.WriteLine("\nDebug\t: {0}/{1}/{2} {3}:{4}:{5}",
                 homNay.Month, homNay.Day, homNay.Year,
         homNay.Hour, homNay.Minute, homNay.Second);
            System.Console.WriteLine("Time\t: {0}/{1}/{2} {3}:{4}:{5}",
               Month, Date, Year, Hour, Minute, Second);
        }

        // các biến thành viên private
        private int Year;
        private int Month;
        private int Date;
        private int Hour;
        private int Minute;
        private int Second = 30;   // initializer
    }

    public class Tester
    {
        static void Main()
        {
            System.DateTime currentTime = System.DateTime.Now;
            Time t = new Time(currentTime);
            t.DisplayCurrentTime();

            Time t2 = new Time(2000, 11, 18, 11, 45);
```

```
            t2.DisplayCurrentTime();

        }   // end Main
    }   // end Tester
```

Kết xuất

Debug: 11/27/2000 7:52:54
Time: 11/27/2000 7:52:54

Debug: 11/27/2000 7:52:54
Time: 11/18/2000 11:45:30

Nếu bạn không cung cấp một initializer đặc thù, thì hàm constructor sẽ cho khởi gán từng biến thành viên số nguyên về zero. Tuy nhiên, trong trường hợp ở đây, biến **Second** được khởi gán về 30:

```
private int Second = 30;   // initializer
```

Nếu một trị không được chuyển cho biến **Second**, thì trị của biến này sẽ được cho về 30 khi **t2** được tạo ra:

```
Time t2 = new Time(2000, 11, 18, 11, 45);
t2.DisplayCurrentTime();
```

Tuy nhiên, nếu một trị được gán cho biến **Second**, như được thấy trong hàm constructor (lấy một đối tượng **DateTime**, được in đậm) thì trị này sẽ đè chồng lên trị của initializer.

Lần đầu tiên thông qua chương trình ta triệu gọi hàm constructor nhận một đối tượng **DateTime**, và biến **Second** được khởi gán về **54**. Lần 2, ta ghi rõ ra thời gian là **11:45** (giây không được đặt để) và initializer lần này làm chủ. Nếu chương trình không có initializer và cũng không gán một trị cho biến **Second** thì lúc ấy trình biên dịch sẽ gán zero cho biến **Second.**

5.2.3 Copy Constructors

Một *copy constructor* sẽ tạo ra một đối tượng mới bằng cách sao qua những biến từ một đối tượng hiện hữu cùng kiểu dữ liệu. Copy constructor cho phép bạn khởi gán một thể hiện lớp (hoặc thể hiện struct) bằng cách chép một thể hiện khác cùng thuộc một lớp. Thí dụ, có thể bạn muốn chuyển một đối tượng **Time** cho một hàm constructor **Time**, như vậy đối tượng **Time** mới sẽ mang cùng trị với đối tượng cũ.

C# không cung cấp một copy constructor, do đó nếu bạn muốn có một copy constructor thì bạn phải tự cung cấp lấy. Một copy constructor như thế sẽ sao những phần tử từ đối tượng nguyên thủy về một đối tượng mới:

```
public Time(Time existingTimeObject) // copy constructor
{
    Year = existingTimeObject.Year;
    Month = existingTimeObject.Month;
    Date = existingTimeObject.Date;
    Hour = existingTimeObject.Hour;
    Minute = existingTimeObject.Minute;
    Second = existingTimeObject.Second;
}
```

Một copy constructor được triệu gọi bằng cách cho thể hiện một đối tượng kiểu **Time** chẳng hạn, rồi chuyển cho nó tên của đối tượng **Time** cần được sao:

```
Time t3 = new Time(t2);
```

Ở đây, một **existingTimeObject** (**t2)** được trao như là một thông số cho copy constructor, và hàm này sẽ tạo một đối tượng **Time** mới, **t3**. Ta dùng thể hiện **t2** để khởi gán **t3**. **t2** và **t3** cùng thuộc lớp **Time**.

5.2.4 Từ chốt *this*

Từ chốt **this** ám chỉ thể hiện hiện hành của một đối tượng. Qui chiếu **this (**thỉnh thoảng còn được gọi là *con trỏ this*, this pointer) là một con trỏ nằm ẩn đối với mọi hàm hành sự static của một lớp. Mỗi hàm hành sự có thể qui chiếu về các hàm và biến khác của một đối tượng thông qua qui chiếu **this** này.

Điển hình, có 3 cách sử dụng qui chiếu **this**. Cách thứ nhất, là "chính danh hoá" (qualify) các thành viên thể hiện (instance member) bằng không bị cho nằm ẩn bởi các thông số, như sau:

```
private int hour; // biến thành viên

public void SomeMethod(int hour)
{
    this.hour = hour;
}
```

Trong thí dụ này, hàm **SomeMethod()** nhận một thông số, **hour** mang cùng tên với biến thành viên của lớp. Qui chiếu **this** này giải quyết sự nhập nhằng về tên biến. Trong khi **this.hour** ám chỉ biến thành viên, trong khi **hour** thì lại ám chỉ thông số.

Bênh vực cho ý kiến này là ta chọn đúng tên biến và dùng nó vừa là biến thành viên vừa là thông số. Ngược lại cũng có người cho rằng, dùng một tên cho hai mục đích (biến thành viên và thông số) sẽ gây lúng túng cho người đọc và hiểu chương trình.

Cách dùng thứ hai qui chiếu **this** là trao đối tượng hiện hành như là thông số cho một hàm hành sự khác. Thí dụ đoạn mã sau đây:

```
public void FirstMethod(OtherClass otherObject)
{
   otherObject.SecondMethod(this);
}
```

Đoạn mã trên thiết lập 2 lớp, lớp đầu tiên với hàm **FirstMethod()**, và lớp thỨ hai **OtherClass** với hàm hành sự **SecondMethod()**. Trong lòng hàm **FirstMethod()** ta muốn triệu gọi hàm hành sự **SecondMethod()** với việc trao qua đối tượng hiện hành như là thông số để xử lý**.**

Sử dụng thứ ba qui chiếu **this** là với indexer, mà chúng tôi sẽ đề cập ở chương 10, "Bản dãy, Indexer và Collection".

5.3 Sử dụng các thành viên static

Các thuộc tính và hàm hành sự của một lớp có thể hoặc là ***instance member*** (thành viên khả dĩ thể hiện được) hoặc là ***static member*** (thành viên tĩnh, không thể thể hiện được). Các *thành viên thể hiện* thường được gắn liền với những thể hiện của một kiểu dữ liệu, trong khi static member sẽ được xem như là thành phần của lớp. Bạn truy xuất một static member thông qua tên lớp theo đấy thành viên được khai báo. Thí dụ, giả sử bạn có một lớp mang tên **Button**, và đã cho thể hiện những đối tượng thuộc lớp này, **btnUpdate** và **btnDelete**. Giả sử lớp **Button** có một hàm hành sự static mang tên **SomeMethod()**. Muốn truy xuất hàm static này, bạn viết:

```
Button.SomeMethod();
```

thay vì viết;

```
btnUpdate.SomeMethod();
```

Bạn nhớ cho, trên C#, là *bất hợp lệ khi truy xuất một thành viên static của một lớp nào đó thông qua một thể hiện lớp* này, và nếu vô tình vi phạm thì trình biên dịch sẽ "la làng".

Trong vài ngôn ngữ lập trình, người ta phân biệt các hàm hành sự lớp và các hàm hành sự toàn cục (global) khác có sẵn nằm ngoài phạm vi của bất cứ lớp nào. Trên C#, không có global method, mà chỉ có class method, nhưng bạn có thể đi đến kết quả tương tự bằng cách định nghĩa những static method trong lòng lớp của bạn.

Hàm static hoạt động ít nhiều tương tự như hàm toàn cục, theo đấy bạn có thể *triệu gọi hàm static mà khỏi phải cho thể hiện đối tượng* ngay trong tầm tay. Tuy nhiên, lợi thế của hàm static trên hàm global, là tên được đặt nằm trong phạm vi của lớp xảy ra hàm, và do đó, bạn không nên dồn cục vào global namespace vô số tên hàm. Điều này có thể giúp quản lý các chương trình phức tạp, và tên lớp hành động tương tự như một namespace đối với những hàm static nằm trong lòng lớp.

Bạn để ý: bạn nên từ bỏ cám dỗ tạo một lớp duy nhất trên một chương trình trong ấy bạn dồn tất cả các hàm hành sự tạp nham lại với nhau. Việc này thì làm được, nhưng như vậy sẽ phá vỡ khái niệm gói ghém (encapsulation) trong thiết kế chương trình theo thiên đối tượng.

5.3.1 Triệu gọi các hàm static

Hàm **Main()** là một hàm static. Như đã nói, hàm static hoạt động trên lớp thay vì trên một thể hiện của lớp. Như vậy, hàm static sẽ không có một qui chiếu **this** vì đâu có một thể hiện để mà chĩa về.

Các *hàm static không thể trực tiếp truy xuất các thành viên không static*. Đối với **Main()** muốn triệu gọi một hàm nonstatic, nó phải cho thể hiện một đối tượng. Ta thử xem lại thí dụ 5-2 mà chúng tôi cho sao lại đây cho tiện bề giải thích:

```
using System;

public class MyClass
(   public void SomeMethod(int firstParam, float secondParam)
    {   Console.WriteLine("Sau đây là các thông số nhận được:
                              {0}, {1}", firstParam, secondParam);
    }
}

public class Tester
{
    static void Main()
    {
        int howManyPeople = 5;
        float pi = 3.14;
        MyClass mc = new MyClass();
        mc.SomeMethod(howManyPeople, pi);
    }   // end Main
```

```
}   // end Tester
```

SomeMethod() là một hàm nonstatic thuộc lớp **MyClass**. Đối với **Main()**, muốn truy xuất hàm này, nó phải cho thể hiện một đối tượng kiểu MyClass, ở đây là **mc**; rồi sau đó mới có thể triệu gọi **SomeMethod()** thông qua đối tượng này. Chúng tôi in đậm hai dòng lệnh thực hiện việc này.

5.3.2 Sử dụng hàm static constructor

Nếu lớp của bạn có khai báo một hàm static constructor, bạn sẽ được bảo đảm rằng *hàm constructor static này sẽ được triệu gọi trước khi bất cứ thể hiện nào của lớp được tạo ra.*

Thí dụ, có thể bạn cho thêm hàm static constructor sau đây vào lớp **Time:**

```
static Time()
{
   Name = "Time";
}
```

Bạn để ý, trước *hàm static constructor không có access modifier* (nghĩa là **public** chẳng hạn). *Access modifier không được xuất hiện trên hàm static constructor.* Ngoài ra, vì đây là một hàm static constructor, bạn không thể truy xuất một biến thành viên nonstatic, do đó biến **Name** phải được khai báo là một biến thành viên static:

```
private static string Name;
```

Thay đổi cuối cùng là thêm một dòng lệnh vào **DisplayCurrentTime()** như sau:

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

Khi những thay đổi được thực hiện, kết xuất sẽ như sau:

```
Name: Time
11/20/2000 14:39:8
Name: Time
11/18/2000 11:3:30
Name: Time
11/18/2000 11:3:30
```

Mặc dù đoạn mã này chạy được, không nhất thiết bạn phải tạo một hàm static constructor để thực hiện mục tiêu trên. Thay vào đó, bạn sử dụng một initializer:

```
private static string Name = "Time";
```

cho ra kết quả như nhau. Tuy nhiên, hàm static constructor chỉ hữu ích trong công tác "dàn dựng" (setup) mà một initializer không thể thực hiện được và chỉ cần thực hiện một lần thôi.

Thí dụ, giả sử bạn có một đoạn mã "unmanaged" được thừa hưởng từ một COM.dll. Bạn muốn cung cấp một lớp dùng làm vỏ bọc (wrapper) đối với đoạn mã này. Bạn có thể triệu gọi nạp thư viện vào hàm static constructor và khởi gán "jump table" trong hàm static constructor. Thụ lý đoạn mã và cho liên hoạt động với đoạn mã "unmanaged" sẽ được đề cập ở chương 7, "Tương tác với unmanaged code" tập 2 bộ sách này.

5.3.3 Sử dụng Private Constructor

Trên C#, không có hàm hành sự toàn cục (global) hoặc hằng toàn cục. Có thể bạn phải ra tay tạo những lớp tiện ích nho nhỏ chỉ hiện hữu để trữ những thành viên static. Để qua một bên liệu xem là một cách thiết kế tốt hay không, nếu bạn tạo ra một lớp như thế, bạn sẽ không tạo bất cứ thể hiện của lớp này. Bạn có thể ngăn ngừa việc tạo bất cứ thể hiện nào bằng cách tạo một hàm constructor mặc nhiên (hàm không thông số) không làm gì cả, và được khai báo là **private**. Vì không có hàm constructor public, như vậy không thể tạo một thể hiện trên lớp của bạn.

5.3.4 Sử dụng các vùng mục tin static

Sử dụng thông dụng đối với các biến thành viên static là cho theo dõi số lần thể hiện hiện hữu đối với lớp của bạn. Thí dụ 5-5 minh hoạ việc làm này:

Thí dụ 5-5: Sử dụng vùng mục tin static để đếm những lần thể hiện

```
using System;

public class Meo
{   public Meo()
    {
        soLanTheHien++;
    }

    public static void BaoNhieuMeo()
    {
        Console.WriteLine("{0} con mèo nuôi trong nhà.", soLanTheHien);
    }
```

```
    private static int soLanTheHien = 0; // explicit initializer
}   // end Meo

public class Tester
{   static void Main()
    {   Meo.BaoNhieuMeo();
        Meo frisky = new Meo();
        Meo.BaoNhieuMeo();
        Meo whisker = new Meo();
        Meo.BaoNhieuMeo();
    }   // end Main
}   // end Tester
```

Kết xuất
0 con mèo nuôi trong nhà
1 con mèo nuôi trong nhà
2 con mèo nuôi trong nhà

Lớp **Meo** đã được thu gọn lại đến mức cần thiết. Một biến thành viên static, mang tên **soLanTheHien** được tạo ra và được khởi gán về zero. Bạn để ý thành viên static này được xem là thành phần của lớp, chứ không phải là thành viên của thể hiện, do đó không thể được khởi gán bởi trình biên dịch khi tạo ra một thể hiện. Do đó, *cần phải có* một explicit initializer đối với biến thành viên static. Mỗi lần một thể hiện lớp **Meo** được bổ sung, thì biến **soLanTheHien** tăng 1.

Bạn để ý: **Hàm static truy xuất các vùng mục tin static.** Người ta khuyên không nên biến các thành viên dữ liệu thành **public**. Điều này cũng áp dụng đối với các biến thành viên static. Giải pháp cho vấn đề là biến các thành viên static thành **private**, như chúng tôi đã làm với biến **soLanTheHien**. Chúng tôi đã tạo một hàm truy xuất public, **BaoNhieuMeo()** lo truy xuất thành viên **soLanTheHien** private này. Vì **BaoNhieuMeo** cũng là static, nên nó có thể truy xuất biến thành viên static **soLanTheHien**.

5.4 Hủy các đối tượng

C# cung cấp một dịch vụ gọi là "thu lượm rác" (garbage collection, tắt la GC), do đó không cần rõ ra một *hàm hủy (destructor)*. Tuy nhiên, nếu bạn triệu gọi một đoạn mã "unmanaged" có sử dụng nguồn lực (resource), bạn sẽ phải ra lệnh một cách tường minh (explicit) yêu cầu giải phóng ký ức bị chiếm dụng bởi nguồn lực khi bạn làm việc xong với đoạn mã. Việc điều khiển ngầm (implicit) đối với nguồn lực sẽ do hàm hành sự **Finalize()** (thường được gọi là *finalizer*) lo. Hàm này sẽ được dịch vụ hốt rác triệu gọi khi đối tượng của bạn bị hủy.

Bộ phận finalizer chỉ lo giải phóng ký ức nguồn lực mà đối tượng của bạn mượn dùng trong thời gian "tại chức". Bạn để ý là nếu bạn chỉ có những qui chiếu bị quản lý (managed reference), bạn khỏi phải làm việc này; bạn chỉ làm khi bạn phải xử lý những nguồn lực không bị quản lý. Vì thiết đặt finalizer sẽ tốn hao một số ký ức, nên chỉ thiết đặt finalizer trên những hàm hành sự đòi hỏi phải có.

Bạn không bao giờ được triệu gọi trực tiếp hàm hành sự **Finalize()** của đối tượng. Dịch vụ hốt rác sẽ gọi giùm bạn ở hậu trường.

***Bạn để ý*: Hàm Finalize hoạt động thế nào?** Bộ phận hốt rác duy trì một danh sách những đối tượng có hàm **Finalize()**. Danh sách này sẽ được nhật tu khi một đối tượng có hàm **Finalize()** được tạo ra hoặc bị hủy. Khi một đối tượng nằm trên danh sách này được thu gom lần đầu tiên, nó được đặt trên một hàng nối đuôi (queue) với những đối tượng khác chờ được hốt. Sau khi hàm **Finalize()** được thi hành, dịch vụ hốt rác sẽ đi thu lượm đối tượng và nhật tu hàng nối đuôi, luôn cả danh sách các đối tượng cần thu gom.

5.4.1 C# Destructor

Về mặt cú pháp, hàm destructor của C# cũng giống như hàm destructor của C++, nhưng lại hoạt động khác đi. Bạn khai báo một hàm destructor C# dùng đến dấu ngã ~ (tilde) đi trước tên lớp, như sau:

```
~MyClass()
{
    // làm gì ở đây
}
```

cũng tương tự như viết:

```
MyClass.Finalize()
{
    // làm gì ở đây
    base.Finalize();
}
```

Vì có khả năng nhập nhằng gây hiểu lầm, người ta khuyên bạn tránh dùng hàm destructor mà viết ra một explicit finalizer nếu thấy cần.

5.4.2 Hàm Finalize() đối nghịch với hàm Dispose()

Triệu gọi rõ ra một finalizer là bất hợp lệ. Hàm **Finalize()** sẽ được triệu gọi bởi dịch vụ hốt rác (garbage collector). Nếu bạn thụ lý những nguồn lực qúy báu nhưng

unmanaged (chẳng hạn những file handle, "mục quản tập tin") mà bạn muốn đóng lại và giải phóng càng nhanh càng tốt, thì bạn nên thiết đặt giao diện **IDisposable** (chúng tôi sẽ đề cập đến giao diện ở chương 9). Giao diện **IDisposable** đòi hỏi thiết đặt để định nghĩa một hàm hành sự mang tên **Dispose()**, lo thực hiện việc dọn dẹp (cleanup) mà bạn thấy là bức xúc. Sư hiện diện của **Dispose()** là cách khách hàng của bạn bảo rằng "đừng chờ **Finalize()** được triệu gọi, làm ngay bây giờ".

Nếu bạn cung cấp một hàm **Dispose()**, thì bạn yêu cầu garbage collector ngưng triệu gọi **Finalize()** đối với đối tượng của bạn**.** Muốn thế, bạn cho triệu gọi hàm static **GC.SuppressFinalize()**, trao thông số là **this** chĩa về đối tượng của bạn. Hàm **Finalize()** có thể sau đó triệu gọi **Dispose()**. Như vậy bạn có thể viết:

```
public void Dispose()
{   // thực hiện việc dọn dẹp
    // yêu cầu GC (garbage collector) đừng gọi Finalize()
    GC.SuppressFinalize(this);
}

public override void Finalize()
{   Dispose();
    base.Finalize();
}
```

5.4.3 Thiết đặt hàm hành sự *Close*

Đối với vài đối tượng, khách hàng của bạn nên triệu gọi hàm **Close()**. Thí dụ **Close()** mang nhiều ý nghĩa hơn là **Dispose()** nhất là đối với những đối tượng file. Bạn có thể thiết đặt điều này bằng cách tạo ra một hàm hành sự **Dispose()** private và một hàm hành sự **Close()** public, và cho **Close()** triệu gọi **Dispose()**.

5.4.4 Sử dụng lệnh *using*

Vì bạn không biết chắc người sử dụng sẽ triệu gọi **Dispose()**, và vì không đoán trước được hàm **Finalize()** sẽ được triệu gọi lúc nào (nghĩa là bạn không biết lúc nào GC sẽ chạy), C# cung cấp một lệnh **using** bảo đảm là **Dispose()** sẽ được triệu gọi vào lúc sớm nhất, nghĩa là bạn tạo một thể hiện trong lệnh using để bảo đảm là hàm Dispose() sẽ được triệu gọi đối với đối tượng khi lệnh using thoát ly. Một lệnh using có thể được thoát ly hoặc khi đạt đến cuối lệnh using hoặc khi một biệt lệ được tung ra và quyền điều khiển thoát khỏi khối câu lệnh trước cuối lệnh using. Nguyên tắc là khai báo những đối tượng nào bạn dùng, rồi sau đó tạo một phạm vi (scope) cho những đối tượng này sử dụng cặp dấu ngoặc nghéo {}. Khi đạt đến dấu } thì hàm **Dispose()** được triệu gọi đối với đối tượng một cách tự động. Thí dụ 5-6 minh hoạ điều vừa nói trên:

Thí dụ 5-6: Sử dụng lệnh using

```
using System.Drawing;
class Tester
{  public static void Main()
   {  using (Font theFont = new Font("Arial", 10.0f))
      {
         // sử dụng theFont ở đây...
      }  // trình biên dịch sẽ triệu gọi Dispose() đối với theFont

      Font anotherFont = new Font("Courier", 12.0f));
      using (anotherFont)
      {
         // sử dụng anotherFont ở đây...
      }  // trình biên dịch sẽ gọi Dispose() đối với anotherFont
   }
}
```

Trong phần đầu của thí dụ, đối tượng **Font**, **theFont**, được tạo ra trong lòng lệnh using. Khi lệnh using kết thúc, hàm **Dispose()** được triệu gọi đối với **theFont**. Trong phần thứ hai, một đối tượng **Font**, **anotherFont**, được tạo ra ngoài lệnh using. Khi ta quyết định sử dụng **anotherFont**, ta đặt đối tượng này trong lòng lệnh using. Và khi lệnh using kết thúc, hàm **Dispose()** lại được triệu gọi đối với **anotherFont.**

Ngoài ra, lệnh using bảo vệ bạn khỏi những biệt lệ (exception) không tiên liệu trước. Không cần biết đến quyền điều khiển rời khỏi lệnh using thế nào, hàm **Dispose()** được triệu gọi, giống như là có ngầm một khối *try-catch-finally* ở đâu đó. Đề nghị bạn xem chi tiết tại mục "Các đối tượng biệt lệ" ở chương 12. "Thụ lý các biệt lệ".

5.5 Trao thông số cho hàm

Theo mặc nhiên, thông số kiểu trị sẽ được trao qua cho hàm theo trị (xem lại mục "Các đối mục hàm hành sự", 5.1.2 ở đầu chương). Đây có nghĩa là khi một đối tượng kiểu trị được trao qua cho hàm hành sự, thì một bản sao tạm của đối tượng này được tạo ra trong lòng hàm này. Hàm làm gì đó trên bản sao này. Bản nguyên thủy không hề hấn gì. Một khi hàm hoàn tất công việc thì bản sao sẽ bị "vất vào sọt rác". Mặc dù, trao theo trị là trường hợp bình thường và phổ biến, nhưng nhiều lúc, bạn lại muốn trao những đối tượng kiểu trị theo qui chiếu (by reference) để có thể cho thay đổi trực tiếp lên trị này. C# cung cấp từ chốt **ref** (thường được gọi là parameter modifier) để báo cho biết là trao những đối tượng kiểu trị cho hàm theo qui chiếu, và từ chốt **out** (cũng là một parameter modifier) đối với những trường hợp bạn trao một biến **ref** nhưng lại chưa khởi gán trị ban đầu. Ngoài ra, C# cũng hỗ trợ một từ chốt parameter modifier khác, **params** cho phép một hàm hành sự chấp nhận một số lượng thông số không cố định. Chương 10, "Bản dãy, Collections và Indexers", sẽ đề cập đến từ chốt **params**.

5.5.1 Trao thông số theo qui chiếu

Các hàm chỉ có thể trả về cho phía triệu gọi một trị duy nhất (mặc dù trị này có thể là một collection). Ta thử trở lại thí dụ lớp **Time**, và thêm một hàm hành sự **GetTime()**; hàm này trả về giờ, phút và giây. Vì ta không thể trả về 3 trị, ta có thể trao 3 thông số rồi để cho hàm thay đổi những thông số, và quan sát kết quả trên hàm triệu gọi (caller). Thí dụ 5-7 minh hoạ cố gắng đầu tiên của chúng ta:

Thí dụ 5-7: Trả về trị trong dấu vòng ngoặc

```
public class Time
    {   // các hàm hành sự public
        public void DisplayCurrentTime()
        {   Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
                Month, Date, 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;
        }

        // các 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 = 0;
            int theMinute = 0;
            int theSecond = 0;
            t.GetTime(theHour, theMinute, theSecond);
            System.Console.WriteLine("Current time: {0}:{1}:{2}",
                theHour, theMinute, theSecond);

        }  // end Main
    }  // end Tester
```

Kết xuất
11/17/2000 13:41:18
Current Time: 0:0:0

Bạn để ý "Current Time" trên phần kết xuất là 0:0:0. Rõ ràng, cố gắng đầu tiên của chúng ta không thành công. Vấn đề là ở các thông số. Ta trao 3 thông số số nguyên cho **GetTime()**, rồi ta thay đổi các thông số trong **GetTime()**, nhưng khi những trị này được truy xuất trở lại trong **Main()**, chúng không bị thay đổi. Lý do là vì số nguyên thuộc value type, do đó được trao qua theo value type; một bản sao được thực hiện trong **GetTime()**. Điều ta cần là trao trị theo qui chiếu.

Hai thay đổi nhỏ cần phải làm; đầu tiên thay đổi những thông số của hàm **GetTime()** cho biết các thông số thuộc loại **ref**:

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

Kế đến, cho thay đổi triệu gọi hàm **GetTime()** để trao các đối mục như là những qui chiếu;

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

Nếu bạn quên thực hiện bước thứ hai thêm **ref** cho các đối mục, thì trình biên dịch sẽ la làng bảo rằng đối mục không thể chuyển đổi từ **int** qua **ref int**.

Sau khi chỉnh lại hai điểm vừa nêu trên, kết quả sẽ đúng như ý muốn. Bằng cách khai báo các thông số là những thông số **ref**, bạn yêu cầu trình biên dịch trao các thông số theo qui chiếu. Thay vì một bản sao được thực hiện, thì thông số trên **GetTime()** là một qui chiếu về cùng biến (**theHour**) được tạo ra trong **Main().** Khi bạn thay đổi những trị này trong **GetTime(),** thì thay đổi được phản ảnh trong **Main(). Bạn** nhớ cho là thông số **ref** là qui chiếu chĩa về trị nguyên thủy hiện thời, giống như bạn bảo "đây này, làm gì thì làm

ở đây trên bản nguyên thủy". Còn thông số kiểu trị là những bản sao, giống như bạn bảo "đây này bản sao, làm gì thì làm, nhưng chớ đụng đến bản nguyên thủy".

5.5.2 Trao các thông số out với cách gán rõ ràng

C# bắt buộc *definite assignment,* nghĩa là đòi hỏi tất cả các biến phải được gán trị trước khi được sử dụng. Trong thí dụ 5-7, trình biên dịch sẽ la làng nếu bạn không khởi gán **theHour, theMinute** và **theSecond** trước khi trao chúng cho **GetTime()** như là thông số; cho dù trị khởi gán toàn là zero:

```
int theHour = 0;
int theMinute = 0;
int theSecond = 0;
t.GetTime(theHour, theMinute, theSecond);
```

Có vẻ hơi điên điên khi khởi gán những trị như trên vì lập tức bạn lại trao chúng theo qui chiếu cho **GetTime()** để lại bị thay đổi, nhưng nếu không làm thế, thì trình biên dịch sẽ cự nự như sau:

Use of unassigned local variable 'theHour'
Use of unassigned local variable 'theMinute'
Use of unassigned local variable 'theSecond'

C# cung cấp từ chốt **out** (được gọi là parameter modifier) để giải quyết trường hợp này. **out** gỡ bỏ yêu cầu một thông số qui chiếu phải được khởi gán. Thí dụ, những thông số trao qua cho **GetTime()** không cung cấp thông tin cho hàm, các thông số này chỉ là cơ chế để lấy thông tin từ chúng mà ra. Do đó, đánh dấu tất cả 3 là thông số **out** bạn loại bỏ việc khởi gán chúng ở ngoài hàm. Trong lòng hàm hành sự được triệu gọi, các thông số **out** phải được gán một trị trước khi hàm trở về. Sau đây là những khai báo các thông số đã được sửa đổi:

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

và sau đây là dòng lệnh mới triệu gọi hàm **GetTime()** trên **Main()**:

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

Để tóm lược, các biến kiểu trị sẽ được trao qua cho các hàm theo trị. Thông số **ref** được dùng để trao các biến value type cho các hàm theo qui chiếu, cho phép bạn tìm lại

trị bị thay đổi của chúng trên hàm triệu gọi. Còn thông số **out** được dùng để nhận thông tin từ một hàm. Thí dụ 5-8 sau đây viết lại thí dụ 5-7 sử dụng cả 3 loại thông số:

Thí dụ 5-8: Sử dụng các thông số in, out và ref

```
public class Time
    {   // các hàm hành sự public
        public void DisplayCurrentTime()
        {   Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
                Month, Date, Year, Hour, Minute, Second);
        }

        public int GetHour()
        {   return Hour;
        }

        public void SetTime(int hr, out int min, ref int sec)
        {   // nếu số giây trao qua >= 30, thì tăng phút lên 1
            // và cho giây về zero, bằng không để yên cả hai
            if (sec >= 30)
            {   Minute++;
                Second = 0;
            }
            Hour = hr; // cho Hour về trị được trao qua
            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;
        }

        // các 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; // trao theo trị
```

```
            int theMinute;   // không khởi gán
            int theSecond = 20; // trao theo qui chiếu
            t.SetTime(theHour, out theMinute, ref theSecond);
            System.Console.WriteLine("the Minute is now: {0}
                  and {1}seconds", theMinute, theSecond);

            theSecond = 40;
            t.SetTime(theHour, out theMinute, ref theSecond);
            System.Console.WriteLine("the Minute is now: {0}
                  and {1}seconds", theMinute, theSecond);

      }  // end Main
}  // end Tester
```

Kết xuất
11/17/2000 14:6:24
theMinute is now: 6 and 24 seconds
theMinute is now: 7 and 0 seconds

Thí dụ **SetTime()** có hơi "khiêng cưỡng", nhưng minh hoạ cho thấy 3 loại thông số: **theHour** được chuyển giao như là một biến kiểu trị; nhiệm vụ của nó là đặt để biến thành viên **Hour** và không trị nào được trả về sử dụng thông số này. Thông số **ref theSecond** được dùng đặt để một trị trong hàm. Nếu **theSecond** lớn hơn hoặc bằng 30, thì biến thành viên **Second** được cho về zero và tăng 1 đối với biến thành viên Minute. Cuối cùng, thông số **out theMinute** được trao qua cho hàm chỉ để nhận lại trị của biến thành viên **Minute**, do đó thông số **theMinute** được đánh dấu là **out**.

5.6 Nạp chồng các hàm hành sự và hàm constructor

Thường xuyên bạn cần đến nhiều hơn một hàm mang cùng tên. Thí dụ phổ biến mà bạn có thể tìm thấy là trong một lớp bạn có nhiều hơn một hàm constructor. Trong những thí dụ mà bạn đã làm quen mãi tới đây, hàm constructor chỉ nhận một thông số duy nhất là đối tượng **DateTime**. Bây giờ ta muốn có khả năng cho các đối tượng **Time** về bất cứ thời gian nào bằng cách chuyển qua những trị year, month, date, hour, minute và second. Nếu một vài khách hàng sử dụng một hàm constructor, và một số khác lại sử dụng hàm constructor khác. ***Nạp chồng các hàm*** (function overloading) sẽ giúp bạn giải quyết vấn đề kể trên.

Mỗi hàm đều mang một ***dấu ấn*** *(signature)*; dấu ấn được định nghĩa bởi tên hàm và bởi danh sách các thông số (parameter list). Hai hàm khác nhau do dấu ấn mà ra, nghĩa là chúng có tên khác nhau hoặc danh sách thông số khác nhau. Danh sách thông số có thể khác nhau do số lượng thông số khác nhau hoặc kiểu dữ liệu khác nhau. Thí dụ sau đây

cho thấy hàm thứ nhất khác hàm thứ hai dựa trên số lượng thông số, và hàm thứ hai khác hàm thứ ba dựa trên kiểu dữ liệu thông số:

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

Một lớp có thể có bao nhiêu hàm hành sự cũng được, miễn là hàm này khác hàm kia dựa trên dấu ấn.

Thí dụ 5-9 minh hoạ việc lớp **Time** có hai hàm constructor, một hàm nhận đối tượng **DateTime**, hàm kia nhận 6 số nguyên.

Thí dụ 5-9: Nạp chồng hàm constructor

```
public class Time
    {   // hàm hành sự public
        public void DisplayCurrentTime()
        {   Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
                Month, Date, Year, Hour, Minute, Second);
        }

        // hàm constructor thứ nhất
        public Time(System.DateTime dt)
        {   Year =    dt.Year;
            Month =   dt.Month;
            Date =    dt.Day;
            Hour =    dt.Hour;
            Minute =  dt.Minute;
            Second =  dt.Second;
        }

        // hàm constructor thứ hai
        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;
        }

        // các 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(2000, 11, 18, 11, 02, 30)
            t2.DisplayCurrentTime();
        }  // end Main
    }  // end Tester
```

Như bạn có thể thấy, lớp **Time** trong thí dụ 5-9 có hai hàm constructor. Nếu dấu ấn của hàm chỉ vỏn vẹn một cái tên, trình biên dịch sẽ không tài nào phân biệt hàm constructor nào được triệu gọi. khi xây dựng các đối tượng **t1** và **t2**. Tuy nhiên, vì dấu ấn bao gồm các kiểu dữ liệu các đối mục, nên trình biên dịch có thể so khớp triệu gọi tạo **t1** với hàm constructor đòi hỏi một đối tượng **DateTime**. Cũng giống như thế, trình biên dịch có thể gắn hàm triệu gọi tạo **t2** với hàm constructor có 6 số nguyên làm đối mục.

Khi bạn nạp chồng một hàm hành sự, bạn phải thay đổi dấu ấn (nghĩa là tên, số lượng, và kiểu dữ liệu của các đối mục). Bạn cũng có thể thay đổi kiểu dữ liệu trị trả về, nhưng đây là tùy chọn. Nếu chỉ thay đổi kiểu dữ liệu trị trả về không thể là nạp chồng hàm hành sự, và việc tạo ra hai hàm hành sự cùng mang một dấu ấn nhưng lại khác trị trả về sẽ gây ra sai lầm biên dịch. Thí dụ 5-10 minh hoạ điều này:

Thí dụ 5-10: Thay đổi trị trả về trên các hàm được nạp chồng

```
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);
        System,Console.WriteLine("x: {0} y: {1}", x, y);
        long lx = 10;
        long ly = Triple(lx);
        System,Console.WriteLine("lx: {0} ly: {1}", lx, ly);
    }

    static void Main()
    {   Tester t = new Tester();
```

```
        t.Test();
    }   // end Main
}   // end Tester
```

Trong thí dụ trên, lớp **Tester** nạp chồng hàm hành sự **Triple**, hàm thứ nhất nhận một số nguyên còn hàm thứ hai thì lại nhận một số kiểu long. Trị trả về của hai hàm **Triple()** cũng thay đổi. Mặc dù không bắt buộc, nhưng lại thích hợp trong trường hợp này.

5.7 Gói ghém dữ liệu thông qua các thuộc tính

Phần lớn các biến thành viên trong một hàm là những vùng mục tin dành riêng cho hàm trong khi xử lý**,** do đó thường mang tính private. Như vậy, người sử dụng ở ngoài không "vọc" được các biến riêng tư của hàm. Đây là nguyên tắc "che dấu thông tin" (information hiding) của lập trình thiên đối tượng. Nhưng đôi khi người ta vẫn muốn biết tình trạng (state) của các biến này. Do đó, người ta đưa ra những **thuộc tính** (properties) cho phép người sử dụng truy xuất tình trạng của lớp xem như truy xuất trực tiếp các vùng mục tin. Người ta cho thiết đặt thuộc tính này thông qua một hàm hành sự lớp. Người sử dụng muốn truy xuất trực tiếp tình trạng của một đối tượng, nhưng lại không muốn làm việc với hàm hành sự. Tuy nhiên, nhà thiết kế lớp thì lại muốn che dấu tình trạng nội tại của lớp này trong các biến thành viên, và chỉ gián tiếp cho phép truy xuất thông qua một hàm hành sự.

Bằng cách tách rời (decoupling) tình trạng của lớp khỏi hàm hành sự truy xuất tình trạng này, nhà thiết kế tha hồ thay đổi tình trạng nội tại của đối tượng khi thấy cần thiết. Khi lần đầu tiên lớp **Time** được tạo ra, trị của **Hour** có thể được trữ trên một biến thành viên. Khi lớp được thiết kế lại, trị của **Hour** có thể do tính toán mà ra, hoặc lấy từ một căn cứ dữ liệu. Nếu người sử dụng đi trực tiếp vào biến thành viên nguyên thủy của **Hour**, thì việc thay đổi qua tính toán đối với trị này sẽ làm cho người sử dụng "hụt giò". Bằng cách tách rời và ép người sử dụng dùng qua một hàm hành sự (hoặc thuộc tính), lớp **Time** mới có thể thay đổi việc quản lý tình trạng nội tại thế nào mà không phá vỡ chương trình của người sử dụng.

Khái niệm thuộc tính đạt được hai mục tiêu: thuộc tính cung cấp một giao diện đơn giản đối với người sử dụng, xuất hiện như là một biến thành viên lớp. Nhưng lại được thiết kế như là những hàm hành sự, cho phép thực hiện việc che dấu thông tin, một đòi hỏi của lập trình thiên đối tượng. Thí dụ 5-11 minh hoạ việc sử dụng thuộc tính;

Thí dụ 5-11: Sử dụng thuộc tính

```
public class Time
    {   // các hàm hành sự public
        public void DisplayCurrentTime()
        {   Console.WriteLine("Time:\t {0}/{1}/{2} {3}:{4}:{5}",
                month, date, year, hour, minute, second);
        }

        // hàm constructor
        public Time(System.DateTime dt)
        {   year =    dt.Year;
            month =   dt.Month;
            date =    dt.Day;
            hour =    dt.Hour;
            minute =  dt.Minute;
            second =  dt.Second;
        }

        // tạo một thuộc tính
        public int Hour
        {   get
            {   return hour;
            }
            set
            {   hour = value;
            }
        }

        // các 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 = t.Hour;
            System.Console.WriteLine("\nTìm đoc lại hour: {0}\n",
                                     theHour);
            theHour++;
            t.Hour = theHour;
            System.Console.WriteLine("\nNhật tu hour: {0}\n", theHour);
        }  // end Main
    }  // end Tester
```

Muốn khai báo một thuộc tính, bạn viết ra kiểu dữ liệu và tên thuộc tính theo sau là cặp dấu {}. Trong lòng cặp dấu này, bạn có thể khai báo get accessor, **get()** và set

accessor, **set()**, cả hai không có thông số đi kèm, mặc dù hàm **set()** có một thông số hiểu ngầm là **value**, như ta sẽ thấy.

Trong thí dụ 5-11, **Hour** là một thuộc tính. Phần khai báo của thuộc tính này tạo ra 2 accessor (hàm truy xuất thuộc tính): **get()** và **set()**:

```
// tạo một thuộc tính Hour
public int Hour
{   get
    {   return hour;
    }

    set
    {   hour = value;
    }
}
```

Mỗi accessor có một phần thân lo truy tìm và đặt để trị thuộc tính. Trị thuộc tính có thể được trữ trên một căn cứ dữ liệu, hoặc có thể được trữ trên một biến thành viên private:

```
private int hour;
```

5.7.1 *get* Accessor

Phần thân của **get** accessor giống như một hàm hành sự lớp trả về một đối tượng kiểu dữ liệu của thuộc tính. Trong thí dụ trên, **get** accessor của thuộc tính **Hour** giống như một hàm hành sự trả về một **int**. Nó trả về trị của một biến thành viên private theo đấy trị của thuộc tính được trữ.

```
get
{
    return hour;
}
```

Trong thí dụ trên, một biến thành viên cục bộ **int** được trả về, nhưng bạn cũng có thể tìm lại dễ dàng trị số nguyên này trên một căn cứ dữ liệu, hoặc cho tính toán ra ngay tại chỗ.

Bất cứ lúc nào bạn qui chiếu thuộc tính (thay vì gán), thì **get** accessor sẽ được triệu gọi để đọc trị của thuộc tính:

```
Time t = new Time(currentTime);
int theHour = t.Hour;
```

Trong thí dụ trên, trị của thuộc tính **Hour** của đối tượng **Time**, **t** được tìm lại, gọi **get** accessor để trích thuộc tính này ra, rồi sau đó đem gán cho biến cục bộ, **theHour**.

5.7.2 *set* Accessor

set accessor cho đặt để trị của một thuộc tính và cũng giống như một hàm hành sự trả về **void**. Khi bạn định nghĩa một **set** accessor bạn phải sử dụng từ chốt **value** để tượng trưng cho đối mục theo đấy trị được trao qua và được trữ bởi thuộc tính.

```
set
{
   hour = value;
}
```

Một lần nữa, một biến thành viên private được dùng để trữ trị của thuộc tính, nhưng **set** accessor cũng có thể viết lên căn cứ dữ liệu hoặc nhật tu các biến thành viên khác nếu thấy cần thiết.

Khi bạn gán một trị cho thuộc tính, thì tự động **set** accessor được triệu gọi, và trị của thông số hiểu ngầm **value** được đặt để trị bạn gán:

```
theHour++;
t.Hour = theHour;
```

Lợi điểm của cách tiếp cận này là người sử dụng có thể tương tác trực tiếp với thuộc tính, mà khỏi vi phạm nguyên tắc cất giấu thông tin và gói ghém, là những nguyên lý rất cơ bản của lập trình thiên đối tượng.

5.8 Các vùng mục tin read-only

Có thể bạn cũng muốn tạo một phiên bản lớp **Time** chịu trách nhiệm cung cấp những trị static và public tượng trưng cho thời gian và ngày hiện hành. Thí dụ 5-12 minh hoạ cách tiếp cận đơn giản đối với vấn đề.

Thí dụ 5-12: Sử dụng static public constants

```
public class RightNow
{   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;
    }

    // các biến thành viên
    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()
    {   System.Console.WriteLine("This year: {0}",
                                  RightNow.Year.ToString());
        RightNow.Year = 2002;
        System.Console.WriteLine("This year: {0}",
                                  RightNow.Year.ToString());
    }   // end Main
}   // end Tester
```

Kết xuất
This year: 2000
This year: 2002

Chương trình này chạy tốt, cho tới khi một người nào đó thay đổi một trong những trị này. Như thí dụ trên cho thấy, trị **RightNow.Year** có thể bị thay đổi, chẳng hạn cho về 2002. Rõ ràng đây là điều bạn không muốn thế.

Chúng tôi muốn đánh dấu những trị static là những hằng (constant), nhưng không thể được vì ta không khởi gán chúng cho tới khi hàm static constructor được thi hành. C# cung cấp từ chốt **readonly** đúng dành cho mục đích này. Nếu bạn cho thay đổi khai báo các biến thành viên lớp như sau:

```
    // các biến thành viên
    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;
```

sau đó biến thành chú giải (comment out) lệnh gán lại trên **Main()**:

```
// RightNow.Year = 2002;   // sai!
```

chương trình sẽ biên dịch lại và chạy đúng như theo ý muốn.

5.9 Cuộc sống bên trong của các đối tượng

Trong các phần đi trước của chương này, bạn đã biết qua các lớp được định nghĩa thế nào, và các thuộc tính (còn gọi là vùng mục tin) và các hàm hành sự hoạt động thế nào. Và chắc chắn những gì bạn biết qua đều đang còn mù mờ. Chắc đã đến lúc bạn nên biết qua lớp được thiết đặt và hoạt động thế nào ở sau hậu trường, nghĩa là cuộc sống đằng sau của các đối tượng mà bạn đã tạo ra là thế nào, "tâm tư nguyện vọng" của chúng ra sao.

5.9.1 Thật sự một biến đối tượng (object variable) là gì?

Bây giờ ta thử định nghĩa một lớp mang tên **Authenticator** (kiểm chứng thực) dùng để đặt một mật khẩu (do hàm **ChangePassword()** lo) và để kiểm tra mật khẩu có hợp lệ hay không (do hàm **IsPasswordCorrect()** đảm nhiệm):

```
public class Authenticator
{
    private string Password;

    public bool IsPasswordCorrect(string password)
    {
        return (password == Password) ? true; false;
    }

    public bool ChangePassword(string oldPassword, string newPassword)
    {
        if (oldPassword == Password)
        {
            Password = newPassword;
            return true;
        }
        else
            return false;
    }
}
```

Có trong tay lớp **Authenticator**, bạn viết như sau để tạo hai đối tượng, một cho mang tên là Bush, còn đối tượng kia cho mang tên là Blair, bằng cách sử dụng từ chốt **new**, đồng thời bạn tạo thêm một đối tượng "ảo" thứ ba, mang tên Putin không sử dụng đến tác tử **new:**

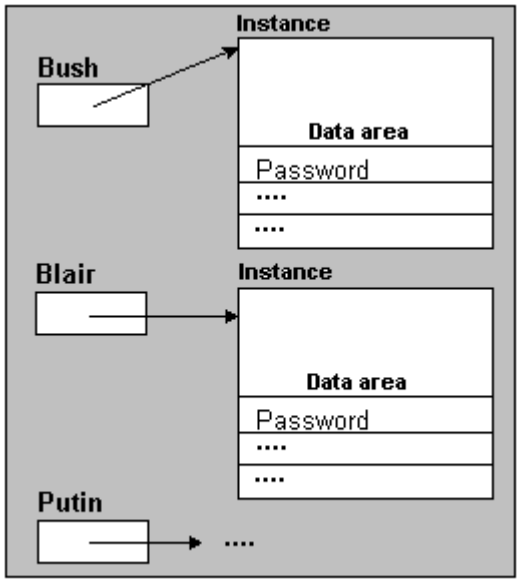
```
Authenticator Bush = new Authenticator();
Authenticator Blair = new Authenticator();
Authenticator Putin;
```

Bush được gọi là một *biến đối tượng* (object variable), nghĩa là một biến chỉ về một đối tượng. Blair cũng là một biến đối tượng khác. Hai biến đối tượng này là hai thể hiện khác nhau riêng biệt của một lớp **Authenticator**. Còn Putin cũng là một biến đối tượng, nhưng hiện chưa có đối tượng nào để chĩa về. Nói tóm lại, 3 câu lệnh trên tạo 3 biến đối tượng, 2 đã có "ý trung nhân" còn một thì chưa.

Thế thì biến đối tượng là cái quái gì thế? Câu hỏi có vẻ hơi vô duyên, vì bạn "bộp chộp" sẽ nghĩ ngay rằng: một biến đối tượng là một vùng ký ức cầm giữ dữ liệu của đối tượng. Có thể bạn liên tưởng đến định nghĩa của những đối tượng kiểu cấu trúc UDT (user-defined type), nhưng rất tiếc câu trả lời trên hoàn toàn sai. Đây là hai khái niệm (lớp và UDT) hoàn toàn riêng biệt. Bush, Blair và Putin là ba biến đối tượng được qui chiếu về cùng một loại đối tượng, nhưng hiện thời chỉ có Putin thì đang chĩa về null (nghĩa là chưa có "ý trung nhân").

Chắc bạn đã biết từ lớp người ta cho hiển lộ (instantiate) ra đối tượng. Nếu đem so sánh với ngành kiến trúc, thì lớp tương đương với một bản vẽ (blueprint), còn căn nhà được xây dựa trên bản vẽ là một đối tượng. Cùng một bản vẽ, nếu có tiền bạn có thể xây bao nhiêu nhà giống y chang cũng được. Tương tự như thế, cũng một lớp bạn có thể hiển lộ ra bao nhiêu đối tượng cũng được.

Tiếp theo, bạn cũng đã biết lớp thường bao gồm những thuộc tính và những hàm hành sự. Phần lớn những thuộc tính này là các vùng mục tin (field) mang tính private chỉ dành riêng cho lớp, người sử dụng không được rờ mó tới để bảo đảm nguyên tắc "cất dấu tập tin" (information hiding). Như vậy, theo cấu trúc, một lớp thường gồm một bộ (set) các thuộc tính và một bộ các hàm hành sự. Ngoài ra, trong mỗi hàm hành sự lại có một số biến dynamic dành riêng cho hàm, mà ta gọi là biến cục bộ thường được cấp phát ký ức trên stack khi hàm được triệu gọi vào làm việc.



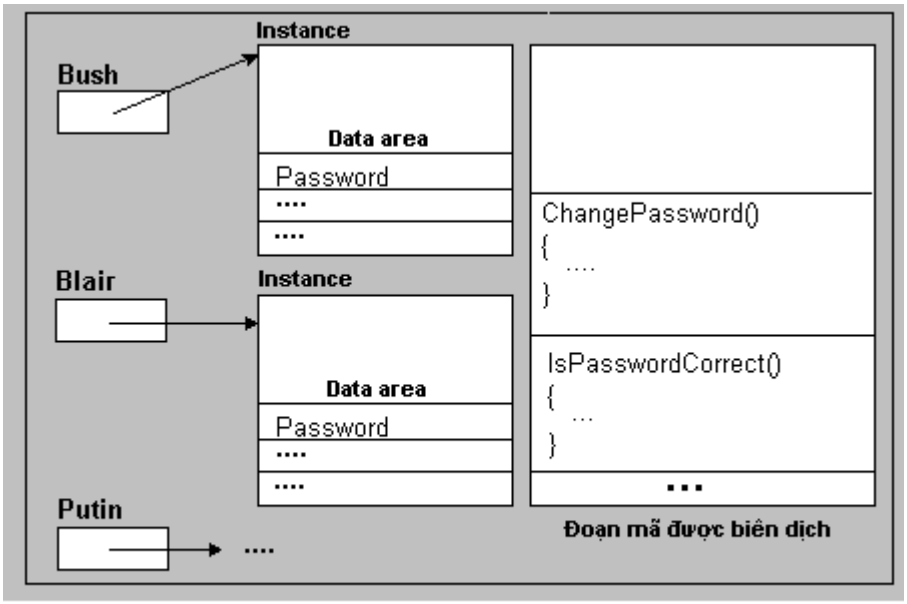
Hình 5-1: Vùng mục tin và biến đối tượng

Khi bạn sử dụng từ chốt **new** để cho thể hiện một đối tượng, thì mỗi đối tượng sẽ có riêng cho mình một lô các thuộc tính, nghĩa là một lô biến vùng mục tin, để đối tượng làm việc. Còn lô hàm hành sự thì chỉ có một dành dùng chung cho tất cả các đối tượng được thể hiện từ lớp. Như vậy, khi bạn tạo hai biến đối tượng Bush và Blair, thì hai biến đối tượng này dùng chung lô hàm hành sự, còn mỗi anh chàng đều có riêng cho mình lô vùng mục tin. Còn anh chàng Putin thì "đang phòng không chiếc bóng". Ta có thể vẽ lô vùng mục tin của hai đối tượng Bush và Blair như sau, hình 5-1.

Bạn thấy lớp **Authenticator** hiện chỉ có một vùng mục tin là **Password**. Do đó, biến đối tượng Bush và Blair, mỗi đối tượng có riêng một chuỗi ký tự mang tên Password, dùng chứa mật khẩu. Thay đổi mật khẩu trên Bush sẽ không ảnh hưởng gì đến trên mật khẩu của Blair. Đây là khái niệm "hồn ai nấy giữ". Ngoài ra, trên hình 5-1, chúng tôi có ghi ở góc trên tay trái chữ "Instance" cho biết vùng mục tin này được tạo trong một thể hiện của lớp. Còn các biến đối tượng được tượng trưng bởi những ô hình chữ nhật có mũi tên chĩa về khối dữ liệu (data area). Bạn thấy trên hình 5-1, biến đối tượng Putin hiện chỉ về hư vô.

Nói tóm lại, mỗi lần bạn khai báo một biến đối tượng thuộc kiểu dữ liệu nào đó, thì theo mặc nhiên biến đối tượng này sẽ nhận được một bản sao tất cả các vùng mục tin thuộc lớp. Như vậy, mỗi thể hiện khác nhau sẽ có những lô vùng mục tin riêng biệt cho thể hiện. Điểm này bạn phải nắm thật chắc.

Bây giờ ta xem một điểm khác. Như bạn đã biết, lớp còn có một số hàm hành sự. Thí dụ lớp **Authenticator** có hai hàm hành sự: **IsPasswordCorrect()** và **ChangePass word()**. Hai hàm này khi biên dịch sẽ chiếm một khoảng ký ức nào đó. Khi lớp được gọi vào, thì ngoài các vùng mục tin vừa kể trên phải cấp phát cho các đối tượng, ta còn phải có chỗ trữ lô hàm hành sự của lớp. Và *chỉ trữ một bản mà thôi.* Ký ức dùng cấp phát cho vùng mục tin của mỗi đối tượng cũng như cho khối hàm hành sự sẽ được lấy từ heap. Nói tóm lại ta có hình 5-2 như sau, với khối các hàm hành sự được biên dịch nằm ở phía tay phải:



Hình 5-2 : Vùng mục tin và đoạn mã các hàm hành sự

Bây giờ làm sao liên lạc với những vùng mục tin và với các hàm hành sự. Vấn đề hơi phức tạp, nhưng chúng ta sẽ phăng lần ra để tìm hiểu. Trong tạm thời, bạn biết cho biến đối tượng, như Bush hoặc Blair chẳng hạn, chỉ là một ***con trỏ*** (pointer) có phận sự chĩa về vùng ký ức chứa dữ liệu vùng mục tin. Nếu biến đối tượng nào chưa có đối tượng cụ thể, Putin chẳng hạn, thì con trỏ chỉ về null. Trên hình 5-1 và 2, ô chữ nhật nhỏ tượng trưng cho biến đối tượng chứa vị chỉ đích, còn mũi tên tượng trưng cho một con trỏ. Đây là một khái niệm rất quan trọng, khá lý thú giúp bạn hiểu vấn đề về sau.

Mỗi biến đối tượng bao giờ cũng chỉ chiếm 4 bytes ký ức (dùng để chứa vị chỉ ký ức - memory address - của khối dữ liệu các vùng mục tin), không cần biết đến kích thước hoặc sự phức tạp của đối tượng mà nó qui chiếu về. Và biến đối tượng (chẳng qua chỉ là một con trỏ) này được cấp phát ký ức trên stack.

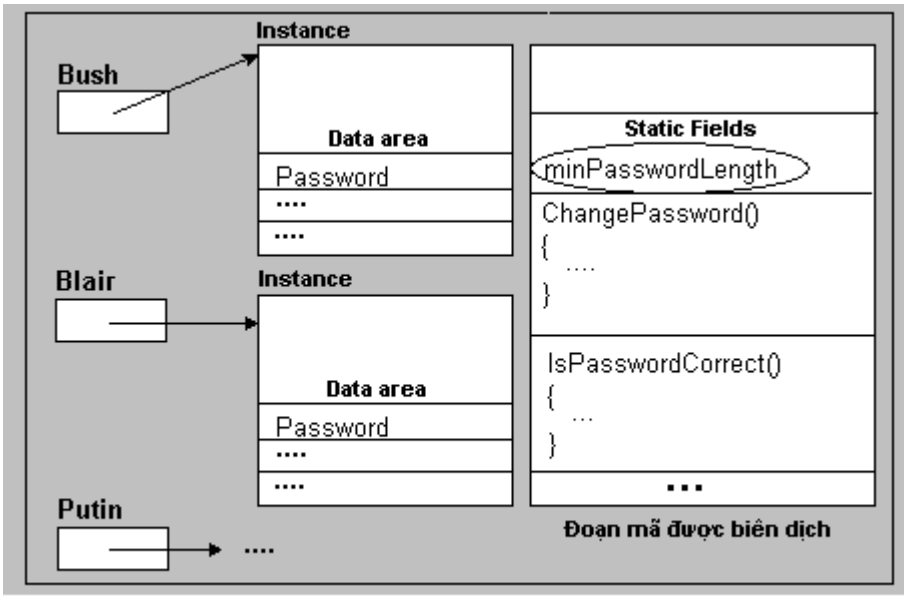
Để cho vấn đề phức tạp thêm, ta giả sử đối với lớp **Authenticator** của ta, ta muốn giới hạn bề dài tối thiểu của mật khẩu là 6 ký tự mà thôi. Đây là bề dài áp dụng cho tất cả các thể hiện của lớp **Authenticator.** Ở đây, ta không muốn mỗi thể hiện mang một bề dài tối thiểu, có nghĩa là bề dài tối thiểu chỉ được trữ một lần thôi trong ký ức, không cần biết đến bao nhiêu thể hiện sẽ được tạo ra. Rắc rối phải không bạn, giống như sự đời.

Muốn cho biết một vùng mục tin chỉ được trữ một lần thôi, dùng chung cho mọi thể hiện của lớp, ta sẽ dùng đến từ chốt **static** đặt nằm đầu khai báo vùng mục tin. Thí dụ

```
public class Authenticator
{
```

```
        private static uint minPasswordLength = 6; // mới thêm vào
        private string Password;
    …
```

Cho trữ một bản sao **minPasswordLength** đối với mỗi thể hiện của lớp **Authenticator** là một phí phạm ký ức. Cho nên cho trữ một lần thôi bằng cách khai báo là **static**, đồng thời khởi gán luôn trị, bằng 6. Như vậy có hai loại biến vùng mục tin; loại **static** như **minPasswordLength**, còn gọi là **static field** hoặc **static data**, còn loại kia như **Password**, không phải static được gọi là **instance field** (vùng mục tin thể hiện). Một cách nhìn khác là: static field thuộc lớp, còn instance field thì thuộc đối tượng, hai cấp khác nhau. Bây giờ ta phải nhật tu lại hình số 5-2 thành hình 5-3, để ý đến việc thêm vào vùng mục tin static **minPasswordLength**, nằm ở phía tay phải cùng với khối các hàm hành sự.



Hình 5-3 : Thêm vùng mục tin static

Sự phân biệt giữa static field và instance field rất quan trọng. Khi chương trình bạn đang chạy và lớp được nạp vào thì static field đã hiện hữu. Còn chỉ khi nào một thể hiện lớp được khai báo, nghĩa là khi biến đối tượng (như Bush hoặc Blair chẳng hạn) được khai báo với từ chốt **new** thì lúc ấy instance field mới hiện hũu.

Từ chốt **static** hoàn toàn độc lập so với việc truy xuất các thành viên lớp, nghĩa là một thành viên lớp có thể là **public static** hoặc **private static**.

Đối với các hàm hành sự, ta cũng có thể sử dụng từ chốt **static**. Nếu không có từ chốt **static**, hàm hành sự được triệu gọi đối với một đối tượng đặc biệt (một thể hiện của một

lớp) nào đó. Hàm này được hiểu ngầm là có thể truy xuất tất cả các vùng mục tin thành viên (kể cả các hàm hành sự, indexer, v.v.) của đối tượng đặc biệt này.

5.9.2 Hàm hành sự instance và static

Trong chương này, ở mục 5.3 chúng tôi đã đề cập qua việc sử dụng các thành viên static. Giống như với các vùng mục tin, ta cũng có khả năng khai báo một hàm hành sự là static, với điều kiện là *hàm static này không được truy xuất bất cứ instance data hoặc instance method nào*. Thí dụ, có thể ta muốn có một hàm hành sự lo đọc xem bề dài tối thiểu của mật khẩu.

```
public class Authenticator
{
   private static uint minPasswordLength = 6;
   private string Password;

   public static uint GetMinPasswordLength()
   {
      return minPasswordLength;
   }
…
```

Nếu một hàm nào không khai báo rõ ra là static, thì nó thuộc instance method, và nó được gắn liền với một biến đặc biệt - một thể hiện đặc biệt của lớp, giống như với instance field.

Tới đây, thiết nghĩ bạn đã phân biệt thế nào instance field, static field, instance method và static method. Bạn có thể hình dung static field & static method giống như món ăn làm sẵn ăn liền, còn instance field & instance method giống như món ăn phải đặt trước theo đúng khẩu vị người đặt.

5.9.3 Truy xuất các thành viên static và instance

Việc các hàm static cũng như vùng mục tin static chỉ gắn bó với một lớp thay vì với một đối tượng sẽ phản ảnh cách bạn truy xuất chúng. Thay vì khai báo tên một biến đối tượng trước tác tử dấu chấm (.), bạn cho khai báo tên của lớp như sau:

```
Console.WriteLine(Authenticator.GetMinPasswordLength()); // OK
Console.WriteLine(Bush.GetMinPasswordLength());   // Trật lất rồi !
```

vì **GetMinPasswordLength()** là một hàm static, nên câu lệnh thứ hai không thể gắn với đối tượng Bush. Ngược lại, khi bạn viết:

```
bool bCorrect = Authenticator.IsPasswordCorrect("Hi"); // Sai rồi
bool bCorrect = Bush.IsPasswordCorrect("Hi");     // OK
```

thì hàm **IsPasswordCorrect()** không phải là hàm static nên câu lệnh thứ nhất không thể gắn liền với lớp.

Do đó**, qui tắc truy xuất sẽ là bao giờ bạn triệu gọi thành viên static bằng cách khai báo tên lớp, còn triệu gọi thành viên instance thì phải khai báo tên thể hiện.**

Bạn để ý: Nếu việc truy xuất các thành viên hàm hành sự hoặc vùng mục tin xảy ra ngay lòng lớp, thì bạn dùng trực tiếp tên của thành viên lớp:

```
// đoạn mã này nằm trong một hàm trong lòng lớp Authenticator
string Passwd = Password;  // instance field trên đối tượng này
int MinLen = minPasswdLength; // static field trong lớp này
```

5.9.4 Các instance và static method được thiết đặt thế nào

Hình số 5-3, cho bạn thấy các hàm hành sự được thiết đặt thế nào trên ký ức. Các instance method, giống như các vùng mục tin static chỉ được trữ một lần thôi, và được gắn liền với lớp như là thành phần trọn vẹn

Như vậy, nếu các hàm hành sự thể hiện chỉ trữ được một lần thôi, làm thế nào một hàm hành sự có khả năng truy xuất đúng bản sao của mỗi vùng mục tin. Nói cách khác, khi bạn viết:

```
Bush.ChangePassword("OldBushPassword", "NewBushPassword");
Blair.ChangePassword("OldBlairPassword", "NewBlairPassword");
```

Làm thế nào trình biên dịch có khả năng kết sinh đoạn mã truy xuất mật khẩu của Bush theo triệu gọi hàm thứ nhất, và mật khẩu của Blair theo triệu gọi hàm thứ hai. Câu trả lời là các hàm hành sự instance nhận thêm một thông số hiểu ngầm, và thông số này qui chiếu về vùng ký ức mà thể hiện lớp được trữ. Bạn có thể hình dung hai câu lệnh trên thực thụ như sau khi biên dịch

```
ChangePassword(Bush,"OldBushPassword", "NewBushPassword");
ChangePassword(Blair,"OldBlairPassword", "NewBlairPassword");
```

Thật ra câu trả lời trên cũng chưa giải tỏa mơ hồ về việc này. Chúng tôi sẽ đi sâu vào vấn đề sau, ở mục "Dưới căn lều thiên đối tượng". Tạm thời ta tiếp tục vời các hàm hành sự static.

Việc khai báo một hàm hành sự là **static** làm cho việc triệu gọi hàm hữu hiệu hơn, vì ta khỏi trao qua một thông số extra hiểu ngầm. Mặt khác, nếu một hàm hành sự được khai

báo là static, nhưng khi cố truy xuất bất cứ vùng mục tin instance nào thì trình biên dịch sẽ la làng, với lý do rõ ràng là bạn không thể truy xuất data instance, trừ khi bạn có vị chỉ của một lớp thể hiện. Nói tóm lại: **bạn không thể truy xuất bất cứ vùng mục tin thể hiện nào từ một hàm hành sự được khai báo là static.**

Đây có nghĩa là trên thí dụ **Authenticator**, bạn không thể khai báo các hàm **Change Password()** và **IsPasswordCorrect()** là **static** vì rằng cả hai truy xuất vùng mục tin **Password** là nonstatic.

Một điểm khá lý thú, mặc dù thông số bị ẩn dấu kèm theo với các hàm hành sự instance không bao giờ được khai báo, bạn hiện lại truy xuất thông số này trong đoạn mã của bạn. Bạn có thể nhận lấy thông số này bằng cách dùng từ chốt **this**. Để lấy một thí dụ, ta có thể viết lại đoạn mã liên quan đến hàm hành sự **ChangePassword()**.

```
public bool ChangePassword(string oldPassword, string newPassword)
{
    if (oldPassword == this.Password)
    {
        this.Password = newPassword;
        return true;
    }
    else
        return false;
}
```

Thông thường, bạn không viết như thế, bạn chỉ tốn thời gian viết dài ra và hơi khó hiểu. Trình biên dịch sẽ tự động suy diễn bất cứ qui chiếu nào về vùng mục tin thành viên không ở trong phạm vi với tên của biến sẽ được xem như là **this.<vùng mục tin>**.

Một điểm khác mà chúng tôi muốn đề cập đến là khi một hàm hành sự instance hoặc static có định nghĩa những biến cục bộ riêng cho mình hoặc nhận thông số, thì những biến này được gắn liền với hàm hành sự và như thế sẽ được trữ riêng biệt mỗi lần hàm hành sự được triệu gọi.

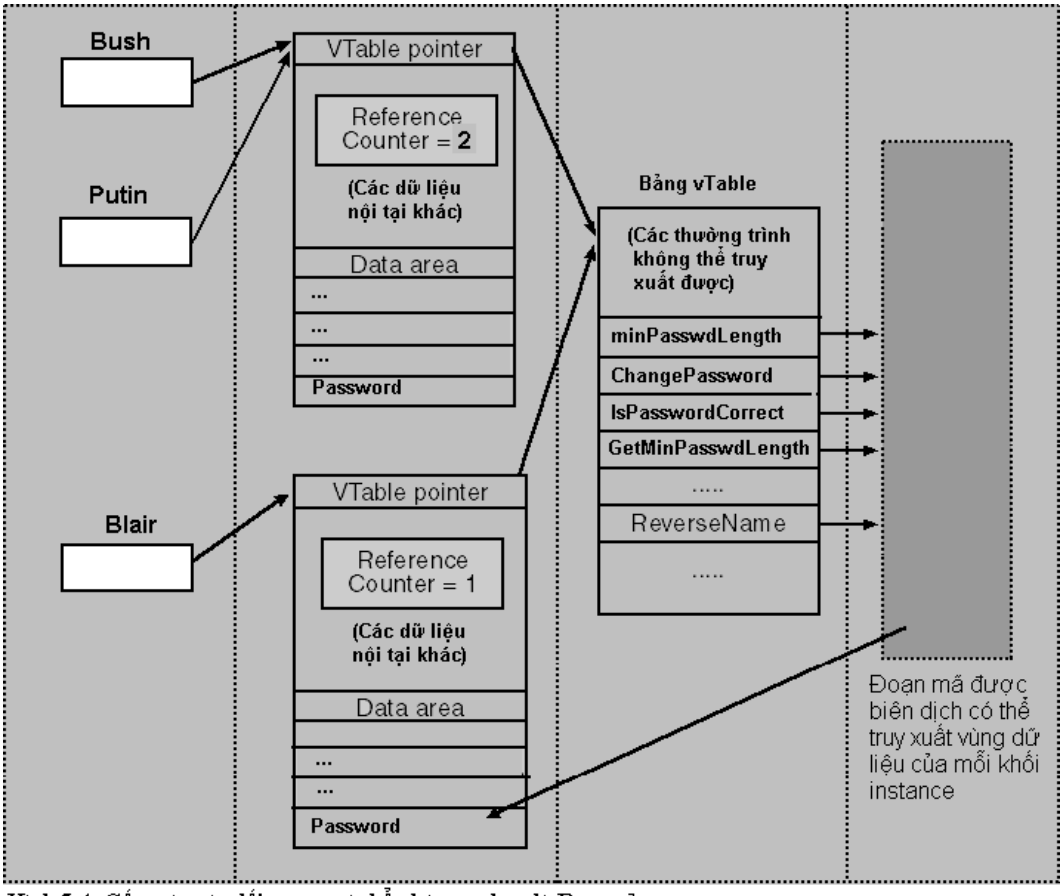
Bây giờ ta đi sâu vào vấn đề mà ta bỏ dở trước đó.

5.9.5 Dưới căn lều thiên đối tượng

Bây giờ ta thêm một câu lệnh gán như sau:

```
Putin = Bush;
```

Như vậy, sau câu lệnh này biến đối tượng Putin không còn cô đơn, mà chia sẽ cùng một "ý trung nhân" với Bush. Hình 5-4 cho thấy cách một đối tượng điển hình được bố trí thế nào trên ký ức.



Hình 5-4: Cấu trúc các đối tượng có thể phức tạp hơn là Bạn tưởng..

Trong thí dụ lớp **Authenticator**, bạn có hai biến đối tượng Bush và Blair, mỗi biến chĩa về một thể hiện khác nhau của lớp **Authenticator,** và lại có hai biến đối tượng Bush và Putin cùng chĩa về một thể hiện. Bất cứ lúc nào bạn dùng từ chốt **new** để tạo một thể hiện mới của lớp, thì C# sẽ cấp phát một vùng ký ức riêng biệt được phân định rõ ràng để chứa những vùng mục tin thể hiện (instance field). Cấu trúc và kích thước của vùng ký ức dành riêng cho từng thể hiện tùy thuộc lớp nào đó cũng như tùy thuộc có bao nhiêu thuộc tính mà lớp chịu trưng ra, và kiểu dữ liệu của thuộc tính v.v..

Tuy nhiên, có một mẫu thông tin đặc biệt quan trọng đối với dân lập trình thiên đối tượng, đó là ***cái đếm số lần qui chiếu*** (reference counter). Đây là một vùng ký ức dài 4 bytes (32 bit) bao giờ cũng chứa số biến đối tượng chĩa về khối dữ liệu instance. Trong lúc này, đối tượng Bush reference counter ghi con số 2, vì có hai qui chiếu là Bush và Putin, còn Blair reference counter chỉ ghi 1. Cái đếm này không thể nào chứa một trị nhỏ thua 1 vì rằng nếu không có nghĩa là không biến đối tượng nào chĩa về đối tượng đặc biệt

này, và lập tức đối tượng phải bị hủy. Dù gì đi nữa, thì đối với bạn là lập trình viên, reference counter cũng chỉ là một thực thể trừu tượng vì bạn không thể đọc được hoặc cho thay đổi trị cái đếm này. Thay đổi duy nhất mà bạn có thể làm một cách hợp lệ và gián tiếp trên cái đếm là tăng hoặc giảm bằng cách sử dụng lệnh gán:

```
Authenticator Chirac;
Authenticator Bush = new Authenticator();   // Tạo một đối tượng
                                            // tăng cái đếm lên 1
Authenticator Blair = new Authenticator(); // Tạo một đối tượng
                                            // tăng cái đếm lên 1
Chirac = Blair;  // tăng cái đếm lên thành 2
Blair = null;    // giảm cái đếm xuống còn 1
Chirac = null;   // giảm cái đếm xuống còn zero, đối tượng sẽ bị hủy
```

Vào cuối khối dữ liệu instance là trị của tất cả các biến vùng mục tin instance, không bao gồm vùng mục tin static. Các biến cục bộ thuộc các hàm hành sự cũng sẽ không bao gồm vì chúng được cấp phát ký ức ở stack mỗi lần được triệu gọi. Lẽ dĩ nhiên, những trị này thay đổi theo từng thể hiện, nhưng cách bố trí thì giống nhau đối với tất cả các thể hiện thuộc lớp.

Một mẫu thông tin khác trên khối dữ liệu instance rất quan trọng nhưng lại không được sưu liệu: đó là con trỏ **vTable pointer**. Đây là một vị trí ký ức dài 32 bit nằm ở trên đầu khối dữ liệu instance, dùng làm con trỏ chĩa về một vùng ký ức chủ chốt khác mang tên là **vTable**. *Tất cả các đối tượng cùng thuộc một lớp đều chĩa về cùng vTable*, như vậy 4 bytes đầu tiên của khối dữ liệu instance đều khớp nhau. Lẽ dĩ nhiên là 4 bytes đầu tiên đối với những đối tượng được hiển lộ bởi những lớp khác nhau sẽ không giống nhau.

Chính **vTable** đem lại cách hành xử khác nhau của một lớp. Thật ra, **vTable** chẳng qua chỉ là một loại bảng, được gọi là **jump table** (bảng nhảy), nghĩa là gồm một loạt con trỏ kiểu Long chĩa về đoạn mã hiện hành được biên dịch. Mỗi con trỏ chĩa về byte đầu tiên của một hàm hành sự được biên dịch. Như bạn đã biết các thuộc tính mà người sử dụng có thể tương tác với lớp cũng thuộc một loại hàm (get/set) đặc biệt. Những thuộc tính Read/Write sẽ có hai mục vào (entry) trên bảng **vTable**. Vì khó lòng biết trước vào lúc biên dịch, ứng dụng sẽ tìm thấy đâu khối ký ức còn rãnh để nạp vào đoạn mã được biên dịch, nên vị chỉ nạp chỉ được biết vào lúc chạy (run-time). Do đó, cấu trúc của **vTable** sẽ được tạo một cách linh động vào lúc chạy chương trình mà thôi

5.9.5.1 Hiển lộ[35] một đối tượng (Object instantiation)

Lần đầu tiên khi C# tạo một đối tượng từ một lớp nào đó, CLR sẽ cho thi hành một loạt tác vụ (đã được đơn giản hoá) như sau:

[35] Chúng tôi dịch từ "Instantiate, hoặc instantiation" bởi từ "hiển lộ". Đây là một từ ngữ lập trình thiên đối tượng với khái niệm rất mới. Đừng lầm lẫn với hiển thị là "display".

1. Cấp phát một khối ký ức đối với đoạn mã được biên dịch từ lớp, rồi nạp đoạn mã này từ đĩa lên (trên hình 5-4, đó là khối hình chữ nhật nằm tận cùng phía tay phải). Ta gọi khối này là module lớp.

2. Cấp phát một khối ký ức nhỏ dành cho bảng **vTable**, rồi điền vào mỗi mục vào vị chỉ điểm nhập của mỗi hàm hành sự thuộc module lớp. Bạn thấy trên hình 5-4, bảng **vTable** nằm ở phía tay trái module lớp. Trên bảng **vTable**, bạn thấy những mục vào (ChangePassword, IsPasswordCorrect, v.v..) có những mũi tên chĩa qua khối module lớp. Mỗi mũi tên là một con trỏ chĩa về vị chỉ đầu tiên của mỗi hàm đã được biên dịch.

3. Cấp phát một khối ký ức dành cho dữ liệu instance của đối tượng (nằm ở phía tay trái bảng **vTable** trên hình 5-4), đồng thời triệu gọi hàm constructor mặc nhiên của lớp để khởi gán những vùng mục tin này về trị mặc nhiên tùy theo kiểu dữ liệu của vùng mục tin. Ngoài ra, CLR cho đặt để 4 bytes đầu tiên của khối rồi cho điền vị chỉ của vTable. Bạn thấy mũi tên của hai khối dữ liệu instance đều chĩa về **vTable.**

4. Cuối cùng, cho trữ vị chỉ cũng khối ký ức chứa dữ liệu instance lên biến đối tượng đích (Bush, Blair, Putin v.v..). Tới lúc này, đoạn mã người sử dụng có thể làm việc với đối tượng.

Loạt tác vụ kể trên sẽ được thực hiện lần đầu tiên khi tạo đối tượng của một lớp nào đó. Qua những lần sau khi tạo những đối tượng khác cùng thuộc lớp này, thì CLR sẽ bỏ qua các bước 1 và 2 vì bảng **vTable** đã hiện diện trong ký ức rồi. Và khi bạn gán các biến đối tượng (sử dụng lệnh gán), thì bước 3 cũng sẽ bỏ qua và toàn bộ công việc chỉ là việc gán trị 32 bit của biến đối tượng.

Thiết nghĩ, tới đây bạn đã hiểu ra cơ chế của việc tạo đối tượng từ một lớp nào đó.

5.9.5.2 Sử dụng đối tượng

Bây giờ ta thử xem việc gì xảy ra khi chương trình người sử dụng triệu gọi một hàm nào đó. Trước khi đi sâu vào vấn đề, chúng tôi muốn cùng bạn xem qua một khái niệm nhỏ là **offset** mà chúng tôi tạm dịch là **di số.** Khi một người nào đó hỏi bạn nhà ông A ở đâu, nếu bạn biết bạn trả lời liền: "Dạ, cách đây năm căn", nghĩa là từ chỗ bạn đang đứng đếm đủ 5 căn thì tới nơi. Theo thuật ngữ ngôn ngữ assembly, "cách đây 5" là một di số (offset), nghĩa là từ đây di về 5 vị trí là đến nơi.

Giả sử một lớp **Class1**, gồm 3 hàm hành sự, f1, f2, và f3. Khi biên dịch f1 chiếm 500 bytes, f2 chiếm 1000 bytes và f3 chiếm 1500 bytes. Chúng tôi dùng những con số nguyên để giải thích cho dễ nhớ. Và 3 hàm này nằm theo trật tự f1, f2 rồi f3. Như vậy khi biên

dịch xong lớp **Class1** chiếm 3000 bytes. Ở đây, để đơn giản hoá, chúng tôi tạm bỏ qua phần dữ liệu của các biến thành viên. Khi biên dịch, và khi chưa nạp xuống ký ức thì vị chỉ khởi đi của lớp **Class1** bao giờ cũng là zero. Về offset, thì hàm f1 mang di số 0, hàm f2 mang di số 500, còn hàm f3 lại có di số 1500. Chắc bạn biết vì sao tính ra di số này.

Như vậy, trình biên dịch biết được các hàm hành sự của lớp **Class1** được bố trí thế nào, offset ra sao của từng hàm hành sự và thế là nó có thể dịch qui chiếu về một hàm hành sự hoặc thuộc tính (bạn nhớ cho thuộc tính cũng là một hoặc hai hàm) thành một offset cho ghi lên một bảng gọi là **vTable**.

Trên bảng **vTable**, có 7 mục vào (mỗi mục dài 4 bytes, mà chúng tôi ghi "những thường trình không thể truy xuất được") được dùng vào việc chi đó không liên quan đến chúng ta, nên mục vào đầu tiên của các hàm hành sự (hoặc thuộc tính) sẽ bắt đầu từ di số 28 (7 x 4 = 28) trở đi.

Bây giờ ta thử xem các lệnh sau đây đối với lớp Class1:

```
Class1 Toto = new Class1():   // tạo một đối tượng toto
Toto.f2();    // triệu gọi hàm f2()
```

Sau đây là những gì xảy ra ở hậu trường:

1. CLR sẽ tìm thấy trị 32 bit trên biến đối tượng Toto, do đó nó có thể truy xuất khối dữ liệu mà biến đối tượng Toto chĩa về. Nó gặp phải **vTable pointer** ở đầu khối data instance này.

2. Ở đầu khối instance data, CLR tìm thấy vị chỉ của bảng **vTable**. Vì trình biên dịch biết ta muốn thi hành triệu gọi hàm f2(), nó thêm 28 bytes vào trị này, đạt đến mục vào tương ứng của f2() và tìm thấy vị chỉ khởi đi của hàm hành sự f2() mà ta muốn triệu gọi (trên khối hàm nằm ở tận cùng phía tay phải trên hình 5-4).

3. Cuối cùng, chương trình sẽ triệu gọi đoạn mã được biên dịch của hàm, đồng thời trao cho nó nội dung nguyên thủy của biến đối tượng Toto (vị chỉ của khối data instance). Vì đoạn mã được biên dịch biết cấu trúc của khối data instance đối với lớp Class1, nên nó có thể truy xuất tất cả các biến private, xử lý chúng và trả về một kết quả có ý nghĩa đối với phía hàm gọi (caller).

Thật vất vả phải không bạn, chỉ để tìm một hàm, nhưng đây là những gì thế giới đối tượng đang hoạt động. Bạn nên đọc đi đọc lại phần kể trên để có thể hiểu những gì bạn đang viết những câu lệnh lập trình thiên đối tượng. Chúng tôi chắc chắn trong tương lại bạn sẽ cần đến những thông tin trên.

5.9.5.3 Trở lại vấn đề dữ liệu kiểu trị và kiểu qui chiếu

Rất quan trọng tìm hiểu sự phân biệt trong C# giữa dữ liệu kiểu trị (value type) và kiểu qui chiếu (reference type).

Khi bạn khai báo một biến thuộc kiểu dữ liệu bẩm sinh, **int, uint, double** chẳng hạn, bạn hiển lộ một biến kiểu trị. Đây có nghĩa là mỗi biến sẽ chứa một bản sao dữ liệu nó chứa. Thí dụ, bạn xem đoạn mã con con sau đây:

```
int I = 10;
int J, K;
J = I;
K = 10;
```

Sau khi cho chạy đoạn mã kể trên, ta có 3 biến: I, J và K tất cả đều mang trị 10. Bạn để ý là mặc dù cả 3 biến đều mang cùng trị 10, nhưng trong ký ức có 3 trị 10. Nếu bạn thêm:

```
++J;
```

thì J sẽ tăng lên 11, trong khi ấy I và K vẫn ở trị 10. Có thể bạn cho là quá đương nhiên sao lại đem ra bàn nữa. Lý do, là khi bạn tạo một biến kiểu dữ liệu do một lớp đem lại thì sự việc không còn đương nhiên như thế.nữa.

Tất cả các lớp đều thuộc kiểu dữ liệu qui chiếu, có nghĩa là mỗi biến đối tượng được tạo ra từ lớp đều có một qui chiếu về thể hiện lớp được trữ trong ký ức. Trong các phần đi trước chúng tôi đã đề cập dông dài về việc này.

Khi bạn viết như sau đối với lớp **Authenticator**:

```
Authenticator User1;
```

Bạn chỉ làm mỗi một việc là tạo một biến đối tượng với qui chiếu là null (hư vô) nghĩa là chưa có một thể hiện nào để chĩa về. Lúc này lớp **Authenticator** (với tất cả bộ sậu được biên dịch) các được gọi nạp vào ký ức nằm chơi xơi nước. Và khi bạn khai báo tiếp theo sau với tác tử **new**:

```
Authenticator User2 = new Authenticator();
```

thì lúc này bạn hiển lộ một thể hiện bằng cách tạo ra một biến đối tượng User2 qui chiếu về vTable pointer như ta đã thấy ở hình 5-4.

Việc các lớp thuộc kiểu qui chiếu có thể có những tác dụng không chờ đợi khi ta cho so sánh các thể hiện lớp để tìm sự bằng nhau, và cho đặt để những thể hiện lớp cho bằng nhau. Thí dụ, bạn thử xem;

```
Authenticator User1;                            // (1)
Authenticator User2 = new Authenticator();      // (2)
Authenticator User3 = new Authenticator();      // (3)
User1 = User2;                                  // (4)
User2.ChangePassword("", "Chirac");             // (5)
User3.ChangePassword("", "Chirac");             // (6)
if (User2 == User3)
{
   // khối câu lệnh này sẽ không được thi hành mặc dù mật khẩu
   // có giống nhau, nhưng biến đối tượng của hai thể hiện
   // chĩa về các đối tượng khác nhau
}
if (User2 == User1)
{
   // khối câu lệnh này sẽ được thi hành vì biến đối tượng
   // User2 và User1 chĩa về cùng một thể hiện
}
```

Nếu bạn đã thấm nhuần những gì đã giải thích theo hình 5-4, thì bạn biết các biến đối tượng User1, User2 và User3 là những biến qui chiếu về vị chĩ các **vTable pointer**. Do đó câu lệnh số (4):

```
User1 = User2;
```

làm cho cả hai đều chĩa về cùng một vị chỉ của **vTable**, nghĩa là về cùng một đối tượng. Đây có nghĩa là những gì bạn thay đổi trên User2 cũng sẽ ảnh hưởng lên User1. Câu lệnh số (5) cho mật khẩu của User2 về Chirac thì coi như ngầm hiểu ta cho mật khẩu của User1 về Chirac. Chính điều này khác cách hành xử của dữ liệu kiểu trị.

Các lệnh so sánh trên liên quan đến kiểu qui chiếu nên không so sánh nội dung dữ liệu mà chỉ so sánh hai biến qui chiếu có chĩa về cùng một vị chỉ hay không. Do đó bạn thấy kết quả trong các lời chú giải.

Chương 6

Kế thừa và Đa hình

Trong chương trước, ta đã biết qua cách tạo ra những kiểu dữ liệu mới bằng cách khai báo những lớp. Chương này sẽ khám phá những mối quan hệ (relationship) giữa các đối tượng trong một thế giới thực, và làm thế nào để mô hình hóa những mối quan hệ này trong các đoạn mã chương trình. Chương này tập trung vào *specialization (chuyên hoá)* được thiết đặt trên C# thông qua *inheritance (kế thừa)*. Ngoài ra, chương này cũng giải thích làm thế nào những lớp chuyên hóa hơn có thể xem chúng như là những thể hiện của những lớp tổng quát (general) hơn, một tiến trình được gọi là *polymorphisme (tính đa hình)*. Chúng tôi kết thúc chương này bằng cách xem qua các lớp "vô sinh" (sealed) không thể cho chuyên hóa, và những lớp "trừu tượng" (abstract class) chỉ có thể hiện diện ở dạng được chuyên hóa, và một đề cập đến nguồn gốc của tất cả các lớp, đó là lớp **Object**.

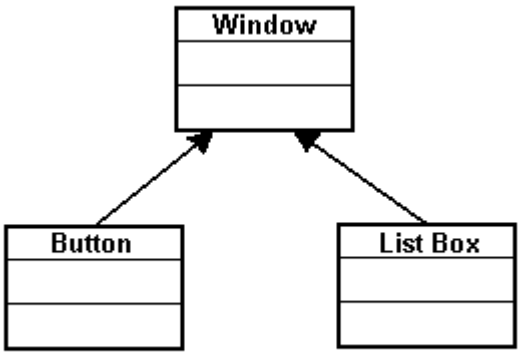
6.1 Chuyên hóa (specialization) và Tổng quát (generalization)

Lớp cũng như những thể hiện của lớp (những đối tượng) không hiện hữu trong chân không (vacuum), mà lại hiện hữu trong một mạng lưới lệ thuộc lẫn nhau và những mối quan hệ chằng chịt, giống như những con người chúng ta sống trong một thế giới quan hệ và phân loại.

Mối quan hệ ***is-a*** (là-một) là mối quan hệ ***chuyên hoá*** *(specialization)*. Khi ta bảo con chó, **Dog**, là một vật có vú (đẻ con, nuôi con bằng sữa, có lông), **Mammal,** có nghĩa là một loại có vú chuyên biệt. Nó mang tất cả các đặc tính của bất cứ vật có vú nào, nhưng lại có những đặc tính chuyên biệt quen thuộc của giống chó nuôi trong nhà. Một con mèo, **Cat**, cũng thuộc vật có vú. Như vậy, ta chờ đợi con mèo cùng chia sẻ một vài đặc tính với con chó được khái quát trong động vật có vú, nhưng lại có những đặc tính riêng biệt giúp phân biệt là giống mèo.

Những mối quan hệ specialization và generalization (tổng quát hoá) cả hai đều có qua có lại và mang tính đẳng cấp (hierarchical). Quan hệ qua lại (reciprocal) là vì specialization là phía "đối trọng" của generalization. Do đó**, Dog** và **Cat** chuyên hoá **Mammal**, còn **Mammal** thì lại tổng quát hóa **Dog** và **Cat.**

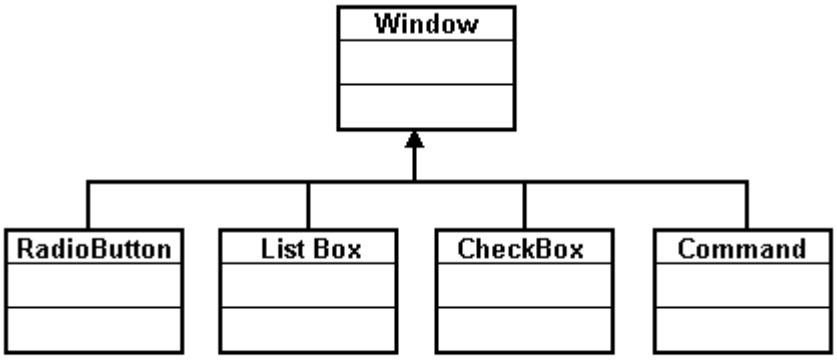
Những mối quan hệ này mang tính đẳng cấp vì chúng tạo ra một "cây quan hệ" (relationship tree), với loại chuyên biệt nằm ở phía lá, còn loại tổng quát hơn thì nằm ở phía nhánh. Nếu bạn đi lần từ ngọn xuống gốc (cây quan hệ này nằm lộn ngược so với cây trong thiên nhiên) thì bạn đi sâu vào *generalization*. Nếu bạn đi lần về **Mammal** bạn khái quát hoá việc **Dogs, Cats** và **Horses** tất cả đều sinh con, còn khi bạn lần xuống cây đẳng cấp thì bạn lại chuyên hóa. Do đó, **Cat** chuyên hóa **Mammal** bằng cách có móng vuốt (một đặc tính) và kêu meo meo (một hành xử).



Hình 6-1: Một mối quan hệ là-một

Cũng tương tự như thế, khi bạn bảo những ô control **ListBox** và **Button** ***là*** những cửa sổ (window) nho nhỏ, bạn cho biết là có những đặc tính và cách hành xử của **Windows** mà bạn chờ đợi tìm thấy trong cả hai loại ô liệt kê và nút ấn. Nói cách khác, **Windows** tổng quát hoá những đặc tính được chia sẻ của cả **ListBox** lẫn **Button**, trong khi ấy mỗi ô control trên lại chuyên hóa những đặc tính và hành xử riêng biệt của mình.

Hình 6-1, cho mô hình hoá những mối quan hệ vừa kể trên, theo UML ([36]Unified Modeling Language).

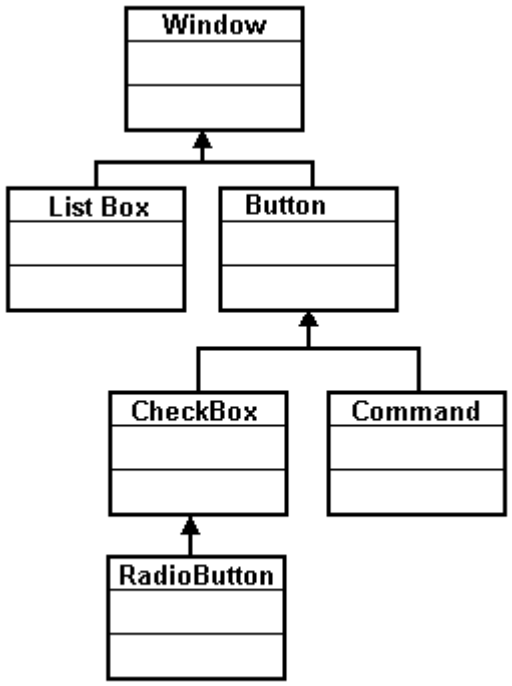


Hình 6-2: Dẫn xuất từ Window.

Thông thường người ta ghi nhận những gì mà hai lớp chia sẻ chức năng, rồi sau đó tách ra những gì chung cho cả hai lớp bỏ vào một lớp cơ bản (base class). Điều này cho phép bạn sử dụng lại đoạn mã viết dùng cho cả hai lớp, và việc bảo trì đoạn mã này cũng dễ dàng hơn.

[36] Bạn có thể tìm đọc bộ sách nói về UML và Rational Rose của Dương Quang Thiện

Thí dụ, giả sử bạn bắt đầu tạo một loạt đối tượng (**Radio Button, Check Box, Command**, và **List Box**) như theo hình 6-2.



Hình 6-3: Một cây đẳng cấp tinh vi hơn

Sau khi làm việc với **RadioButton, CheckBox,** và **Command** một thời gian, bạn nhận ra rằng chúng chia sẻ một vài đặc tính và hành xử mang tính chuyên hóa cao hơn **Window,** nhưng lại tổng quát hơn bất cứ 3 ô control này. Bạn có thể tách những đặc tính và hành xử chung làm thành một lớp cơ bản cho mang tên **Button**, rồi bạn vẽ lại cây đẳng cấp như theo hình 6-3. Đây là một thí dụ cho thấy làm thế nào khái niệm tổng quát hoá được áp dụng trong công tác triển khai hệ thống theo lập trình thiên đối tượng.

Biểu đồ UML trên cho thấy mối quan hệ giữa các lớp tách ra và cho thấy cả hai lớp **ListBox** và **Button** đều được dẫn xuất từ **Window**, và đến phiên **Button** được chuyên hóa thành **CheckBox** và **Command**. Cuối cùng **RadioButton** lại được dẫn xuất từ **CheckBox**. Như vậy bạn có thể nói **RadioButton** *là một* **CheckBox**, và đến phiên **CheckBox** *là một* **Button**, và **Button** *là một* **Window**.

Đây không phải là tổ chức duy nhất, hoặc kể cả tốt nhất, đối với những đối tượng này, nhưng đây là bước khởi đầu trong việc tìm hiểu các lớp (kiểu dữ liệu) này liên hệ thế nào với các lớp khác.

6.2 Tính kế thừa (Inheritance)

Trên C#, mối quan hệ chuyên hóa thường được thiết đặt sử dụng đến tính kế thừa (inheritance). Đây không chỉ là cách thức duy nhất để thiết đặt chuyên hoá, nhưng đây là cách thông dụng và tự nhiên nhất để thiết đặt mối quan hệ này.

Khi bảo lớp **ListBox** kế thừa từ (hoặc được dẫn xuất từ) lớp **Window**, có nghĩa là **ListBox** chuyên hóa **Window**. **Window** được xem như là *lớp cơ bản* (base class), còn **ListBox** được gọi là *lớp dẫn xuất* (derived class). Có nghĩa là **ListBox** dẫn xuất những

đặc tính và lối hành xử của mình từ lớp **Window,** rồi chuyên hóa theo nhu cầu riêng của mình. Thỉnh thoảng, người ta còn gọi lớp cơ bản là **superclass**, còn lớp dẫn xuất là **subclass**.

6.2.1 Thiết đặt tính kế thừa thế nào?

Trên C#, bạn tạo ra một lớp dẫn xuất bằng cách thêm một dấu hai chấm (:) sau tên lớp dẫn xuất, theo sau là tên lớp cơ bản. Dấu hai chấm được đọc như là "được dẫn xuất từ". Thí dụ, ta muốn khai báo một lớp mới, **ListBox**, được dẫn xuất từ lớp cơ bản **Window**, ta viết:

```
public class ListBox: Window
```

Lớp dẫn xuất kế thừa tất cả các thành viên của lớp cơ bản, biến thành viên lẫn hàm hành sự. Ngoài ra, theo nguyên tắc, trên lớp dẫn xuất:

- Ta có thể thêm những thành viên mới (bất cứ loại nào: vùng mục tin, hàm hành sự, thuộc tính v.v..) chỗ nào mà trên lớp cơ bản chưa được định nghĩa.

- Ta có toàn quyền, nếu muốn, thiết đặt phiên bản riêng của mình liên quan đến một hàm hành sự nào đó của lớp cơ bản (hoặc thuộc tính v.v.. ngoại trừ vùng mục tin) hiện đã có trên lớp cơ bản, và chỉ cần đánh dấu hàm mới bởi từ chốt **new**. Với từ chốt **new**, ta muốn nói lớp dẫn xuất có ý cho cất dấu hàm lớp cơ bản và thay thế bởi hàm riêng của lớp dẫn xuất, như theo thí dụ 6-1 sau đây. Ngoài từ chốt **new**, ta còn có một cách thứ hai là **override** (phủ quyết) thi công hàm cũ trên lớp cơ bản. Ta sẽ xem điểm này ở mục 6.3, "Tính đa hình".

Thí dụ 6-1: Sử dụng một lớp được dẫn xuất

```
using System;

public class Window
{   // hàm constructor nhận hai trị số nguyên
    // cho biết vị trí trên màn hình
    public Window(int top, int left)
    {   this.top = top;
        this.left = left;
    }
    // mô phỏng vẽ cửa sổ
    public void DrawWindow()
    {   Console.WriteLine("Vẽ window tại {0}, {1}", top, left);
    }
    // biến thành viên sau đây là private, như vậy là vô hình đối
    // với các hàm hành sự của lớp dẫn xuất
```

```
    private int top;
    private int left;
}

// Lớp ListBox dẫn xuất từ lớp Window
public class ListBox: Window
{   // hàm constructor thêm một thông số theContents
    public ListBox(int top, int left, string theContents):
        base(top, left) // gọi base constructor
    {   listBoxContents = theContents;
    }

    // một phiên bản mới của hàm DrawWindow() trên lớp dẫn xuất
    public new void DrawWindow()
    {   base.DrawWindow();  // triệu gọi hàm hành sự lớp cơ bản
        Console.WriteLine("Viết chuỗi lên ô liệt kê: {0}",
                          listBoxContents);
    }
    private string listBoxContents;   // biến thành viên mới
}

public class Tester
{   static void Main()
    {   // tạo một lớp cơ bản
        Window w = new Window(5, 10);
        w.DrawWindow();

        // tạo một lớp dẫn xuất
        ListBox lb = new ListBox(20, 30, "Xin Chào Bà Con!");
        lb.DrawWindow();
    }   // end Main
}   // end Tester
```

Hình 6-4 cho thấy kết xuất của thí dụ trên:

Hình 6-4: Kết xuất Thí dụ 6-1

Thí dụ 6-1 trên bắt đầu khai báo một lớp cơ bản **Window**. Lớp này thiết đặt một hàm constructor và một hàm hành sự đơn giản vẽ ô control **DrawWindow()**. Có 2 biến thành viên private **top** và **left**.

6.2.2 Triệu gọi các hàm constructor của lớp cơ bản

Trong thí dụ 6-1 trên, lớp mới **ListBox** được dẫn xuất từ **Window** có riêng cho mình một hàm constructor nhận vào 3 thông số. Hàm constructor của **ListBox** triệu gọi hàm constructor của lớp cơ bản "cha mẹ" bằng cách đặt dấu hai chấm sau danh sách các thông số rồi triệu gọi lớp cơ bản bằng cách dùng từ chốt **base**:

```
public ListBox(int top, int left,
   string theContents): base(top, left) // gọi base constructor
```

Vì các lớp không thể thừa kế hàm constructor (hàm constructor xem như là "mở cửa hàng" cho lớp, nên mỗi lớp, kể cả lớp dẫn xuất, có riêng cách "mở cửa hàng" của mình), cho nên lớp dẫn xuất phải thiết đặt riêng cho mình hàm constructor, và chỉ có thể dùng hàm constructor của lớp cơ bản bằng cách triệu gọi một cách tường minh (explicit call).

Ngoài ra, bạn để ý **ListBox** trên thí dụ 6-1 cho thiết đặt một phiên bản mới về **DrawWindow()**:

```
public new void DrawWindow()
```

Từ chốt **new** cho biết ở đây lập trình viên cố ý tạo một phiên bản mới của hàm hành sự **DrawWindow()** cho nằm trong lớp dẫn xuất.

Nếu lớp cơ bản có một hàm constructor mặc nhiên (default constructor) khả dĩ truy xuất được, thì hàm constructor ở lớp dẫn xuất khỏi phải triệu gọi base constructor một cách tường minh; thay vào đó hàm constructor mặc nhiên sẽ được triệu gọi ngầm (implicit). Tuy nhiên, nếu lớp cơ bản không có hàm constructor mặc nhiên, thì mọi hàm constructor ở lớp dẫn xuất *phải* triệu gọi một cách tường minh một trong những hàm constructor lớp cơ bản dùng đến từ chốt **base**.

Bạn để ý: Như đã nói trong chương 5, "Lớp và Đối tượng", nếu bạn không khai báo bất cứ một hàm nào cả, thì trình biên dịch sẽ tạo ra giùm bạn một hàm constructor mặc nhiên. Cho dù bạn tự viết lấy, hay là do trình biên dịch tạo giúp bạn theo mặc nhiên, thì một hàm constructor mặc nhiên phải là *một hàm không thông số.* Tuy nhiên, một khi bạn đã tạo một hàm constructor (có hoặc không có thông số) thì trình biên dịch sẽ *không* tạo một hàm constructor mặc nhiên giùm bạn.

6.2.3 Triệu gọi các hàm hành sự của lớp cơ bản

Trong thí dụ 6-1, hàm **DrawWindow()** của lớp dẫn xuất **ListBox** cất dấu và thay thế hàm hành sự của lớp cơ bản. Khi bạn triệu gọi **DrawWindow()** đối với một đối tượng kiểu dữ liệu **ListBox**, thì đúng là **ListBox.DrawWindow()** sẽ được triệu gọi, chứ không

phải **Window.DrawWindow().** Tuy nhiên, bạn ghi nhận là **ListBox.DrawWindow()** có thể triệu gọi **DrawWindow()** của lớp cơ bản với dòng lệnh sau đây:

```
base.DrawWindow();  // triệu gọi hàm hành sự của lớp cơ bản
```

Từ chốt **base** nhận diện lớp cơ bản đối với đối tượng hiện hành.

6.2.4 Kiểm soát việc truy xuất

Việc lớp với những thành viên của mình có thể nhìn thấy được tới đâu (ta gọi là visibility, tầm nhìn xa) có thể bị hạn chế thông qua việc sử dụng những từ chốt gọi là *access modifier* (từ chốt thay đổi tầm truy xuất) mà ta có thể kể: **public, private, protected, internal** và **internal protected**. Chương 5, "Lớp và Đối tượng" đã đề cập đến ý nghĩa của các từ chốt này.

Như bạn đã biết, **public** (công cộng) cho phép truy xuất một thành viên (biến hoặc hàm) bởi những hàm thành viên của một lớp khác, trong khi **private** (riêng tư) cho biết thành viên của lớp chỉ có thể nhìn thấy được bởi các hàm thành viên thuộc cùng lớp, "người ngoài không nhìn được". Từ chốt **protected** nới rộng tầm nhìn thấy xuống đến các hàm của lớp dẫn xuất, trong khi **internal** lại nới rộng tầm nhìn thấy đến bất cứ lớp nào thuộc cùng *assembly*[37]. Cuối cùng từ chốt **internal protected** cho phép truy xuất những thành viên cùng thuộc *assembly* (internal) hoặc thuộc các lớp dẫn xuất (protected). Bạn có thể hình dung từ chốt này như là **internal** hoặc **protected**.

Lớp cũng như thành viên của lớp có thể được gán với bất cứ cấp độ nhìn thấy được (accessibility level) thông qua các từ chốt vừa kể trên. Nếu một thành viên thuộc lớp mang một mức độ nhìn thấy được khác so với lớp, thì mức độ khe khắt hơn sẽ được áp dụng. Do đó, nếu bạn định nghĩa một lớp, **myClass,** như sau:

```
public class myClass
{
   // ...
   protected int myValue;
}
```

thì mức độ nhìn thấy được đối với **myValue** là **protected** cho dù bản thân lớp **myClass** là ở mức độ **public.** Một *lớp public* là một lớp có thể được nhìn thấy bởi bất cứ lớp nào khác muốn tương tác với lớp này. Thỉnh thoảng, người ta tạo ra những lớp để hỗ trợ

[37] Một assembly là đơn vị chia sẻ và dùng lại trên CLR. Điển hình, assembly là một tập hợp những tập tin (bao gồm các nguồn lực – bitmap, gif) cần thiết đối với chương trình khả thi .EXE kèm theo IL và metadata đối với chương trình này.

những lớp khác trong một *assembly,* và những lớp hỗ trợ (helper) này có thể được đánh dấu bởi từ chốt **internal** thay vì **public**.

6.2.5 Hướng dẫn sử dụng lớp cơ bản

Lớp là một kiểu dữ liệu phổ biến nhất. Một lớp có thể là trừu tượng (abstract) hoặc "vô sinh" (sealed). Một lớp trừu tượng đòi hỏi phải có một lớp dẫn xuất để thiết đặt lớp trừu tượng. Còn lớp bị vô sinh thì không cho phép có một lớp dẫn xuất. Người ta khuyên bạn nên sử dụng lớp so với các kiểu dữ liệu khác.

Các lớp cơ bản (base classe) là một cách hữu ích để gộp lại những nhóm đối tượng chia sẻ chung một số chức năng, đồng thời cung cấp một số chức năng mặc nhiên (default), trong khi vẫn cho phép tự tạo riêng cho mình một số chức năng đặc thù thông qua việc nới rộng chức năng.

6.3 Tính đa hình (polymorphisme)

Có hai khía cạnh rất mạnh đối với tính kế thừa. Khía cạnh thứ nhất là khả năng sử dụng lại các đoạn mã (code reusability) cho phép tăng năng suất lập trình. Khi bạn đang tạo ra một lớp **ListBox**, bạn có khả năng dùng lại một vài lô gic trong lớp cơ bản **Window**.

Tuy nhiên, khía cạnh thứ hai của tính kế thừa, xem ra mạnh hơn, là *tính đa hình (polymorphisme)*. Đa hình là nhiều hình dạng (form). Do đó, tính đa hình ám chỉ khả năng sử dụng nhiều dạng thức của một kiểu dữ liệu không cần biết đến chi tiết.

Khi công ty điện thoại phát đi một tiếng chuông điện thoại, họ không cần biết phía đầu kia bạn có loại điện thoại gì: loại cổ lỗ sĩ phải quay, hoặc loại nút ấn có nhạc đi kèm Như vậy, đối với công ty điện thoại, họ chỉ biết đến "kiểu cơ bản" **phone** và chờ đợi bất cứ "thể hiện" nào của kiểu **phone** này biết sẽ rung chuông thế nào. Khi công ty điện thoại yêu cầu điện thoại bạn *rung chuông* thì đơn giản họ chờ đợi điện thoại của bạn biết làm thế nào cho đúng cách. Như vậy, công ty điện thoại đối xử một cách đa hình điện thoại của bạn.

6.3.1 Tạo những kiểu dữ liệu đa hình

Vì một **ListBox** *là một* **Window** và một **Button** *là một* **Window**, nên ta chờ đợi có khả năng sử dụng không phân biệt các kiểu dữ liệu kể trên trong trường hợp cần gọi đến một **Window**. Thí dụ, một biểu mẫu (form) có thể muốn cất trữ một collection gồm

những thể hiện của **Window** mà biểu mẫu quản lý, như vậy khi biểu mẫu được mở, nó có thể bảo mỗi một trong lô thể hiện **Window** tự vẽ thế nào. Đối với công tác vẽ này, biểu mẫu không cần biết đến phần tử nào trong collection là ô liệt kê, phần tử nào là nút ấn; biểu mẫu chỉ cần rảo qua collection rồi bảo mỗi phần tử tự vẽ (draw) lấy. Nói tóm lại, biểu mẫu muốn đối xử tất cả các đối tượng **Window** theo tính đa hình.

6.3.2 Tạo các hàm hành sự đa hình

Muốn tạo một hàm hành sự chịu hỗ trợ tính đa hình, bạn chỉ cần đánh dấu hàm như là **virtual** (ảo) trong lớp cơ bản.Thí dụ, muốn cho biết hàm **DrawWindow()** thuộc lớp **Window**, trên thí dụ 6-1 mang tính đa hình, bạn chỉ cần thêm từ chốt **virtual** trên phần khai báo hàm này, như theo sau:

```
public virtual void DrawWindow()  // trên lớp Window
```

Bây giờ, mỗi lớp dẫn xuất tự do thiết đặt phiên bản riêng của mình liên quan đến **DrawWindow()**. Muốn thế, bạn chỉ cần "phủ quyết" (override) hàm virtual trên lớp cơ bản bằng cách dùng từ chốt **override** trên dòng lệnh khai báo của hàm lớp dẫn xuất, rồi thêm đoạn mã mới đối với hàm hành sự bị phủ quyết.

Sau đây là đoạn trích từ thí dụ 6-2 (mà chúng tôi sẽ đề cập sau), **ListBox** được dẫn xuất từ **Window** và cho thiết đặt phiên bản DrawWindow riêng của mình:

```
public override void DrawWindow()
{
   base.DrawWindow();      // triệu gọi hàm hành sự lớp cơ bản
   Console.WriteLine("Viết một chuỗi lên ô liệt kê: {0}",
      listBoxContents);
}
```

Từ chốt **override** (phủ quyết) cho trình biên dịch biết là lớp này cố tình phủ quyết cách làm việc của **DrawWindow()** trên lớp cơ bản. Cũng tương tự như thế, bạn sẽ phủ quyết hàm hành sự này trên một lớp khác, **Button**, cũng được dẫn xuất từ lớp **Window**. Bạn thấy ở đây không dùng từ chốt **new** như trong thí dụ trước.

Trong thân thí dụ 6-2, trước tiên bạn tạo ra 3 đối tượng: một **Window**, một **ListBox**, và một **Button**. Sau đó bạn triệu gọi **DrawWindow()** trên mỗi đối tượng:

```
Window win = new Window(1,2);
ListBox lb = new ListBox(3,4, "Ô liệt kê một mình!");
Button but = new Button(5,6);
win.DrawWindow();
lb.DrawWindow();
but.DrawWindow();
```

Đoạn mã này chạy như chờ đợi. Hàm **DrawWindow()** được triệu gọi một cách đúng đắn cho mỗi loại đối tượng. Tới đây thì chả có chi là đa hình. Điều kỳ diệu bắt đầu khi bạn tạo một bản dãy (array) những đối tượng **Window**. Vì **ListBox** *là-một* **Window**, bạn hoàn toàn tự do đưa **ListBox** vào một bản dãy **Window**. Với **Button** cũng thế:

```
Window[] winArray = new Window[3];// bản dãy gồm 3 đối tượng Window
winArray[0] = new Window(1,2);    // phần tử thứ nhất đối tượng Window
winArray[1] = new ListBox(3,4,"Ô liệt kê trên bản dãy"); // phần tử
                          // thứ 2 là ListBox
winArray[2] = new Button(5,6);    // phần tử thứ 3 là một Button
```

Việc gì sẽ xảy ra khi bạn triệu gọi **DrawWindow()** trên mỗi đối tượng này?

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

Trình biên dịch chỉ biết mỗi một việc là có 3 đối tượng **Window**, và bạn triệu gọi **DrawWindow()** trên mỗi đối tượng. Nếu bạn không cho đánh dấu **virtual** trên hàm **DrawWindow**, thì hàm **DrawWindow()** của **Window** sẽ được triệu gọi 3 lần. Tuy nhiên, vì bạn đã cho **DrawWindow()** về virtual, và vì các lớp dẫn xuất đã phủ quyết hàm hành sự này khi bạn triệu gọi **DrawWindow()** trên bản dãy, trình biên dịch sẽ xác định kiểu dữ liệu vào lúc chạy của đối tượng hiện hành (một **Window**, một **ListBox**, và một **Button)**, và gọi vào đúng hàm hành sự cho mỗi đối tượng. Đấy là thực chất của tính đa hình. Sau đây là toàn bộ đoạn mã thí dụ 6-2.

***Bạn để ý*:** Bảng liệt in dưới đây sử dụng đến một bản dãy (array) là một tập hợp những đối tượng cùng một kiểu dữ liệu. Bạn có thể truy xuất các phần tử bản dãy thông qua chỉ mục (index). Chỉ mục bản dãy khởi đi là zero (gọi là zero based):

```
// cho trị của phần tử số 6 về 7
MyArray[5] = 7;
```

Việc sử dụng đến bản dãy trong thí dụ này mang tính trực giác. Chương 10 sẽ đề cập đến bản dãy.

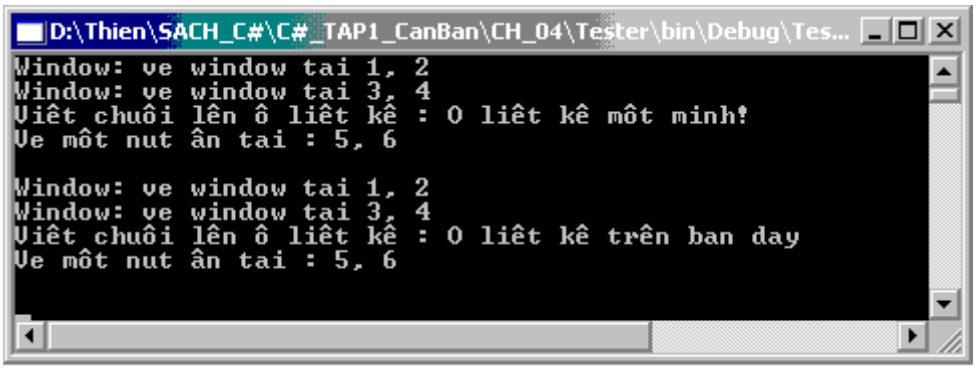
Thí dụ 6-2: Sử dụng các hàm hành sự virtual

```
using System;

public class Window
{   // hàm constructor nhận hai trị số nguyên
    // cho biết vị trí trên màn hình
    public Window(int top, int left)
```

```
    {   this.top = top;
        this.left = left;
    }

    // mô phỏng vẽ cửa sổ
    public virtual void DrawWindow()
    {   Console.WriteLine("Window: vẽ window tại {0}, {1}", top, left);
    }

    // các biến thành viên sau đây là protected, như vậy là
    // có thể nhìn thấy được từ các hàm hành sự của lớp dẫn xuất
    protected int top;
    protected int left;
}

// Lớp ListBox dẫn xuất từ lớp Window
public class ListBox: Window
{   // hàm constructor thêm một thông số
    public ListBox(int top, int left, string contents):
                          base(top, left) // gọi base constructor
    {
        listBoxContents = contents;
    }

    // một phiên bản mới của hàm DrawWindow() trên lớp dẫn xuất
    public override void DrawWindow()
    {   base.DrawWindow();  // triệu gọi hàm hành sự lớp cơ bản
        Console.WriteLine("Viết chuỗi lên ô liệt kê: {0}",
        listBoxContents);
    }
    private string listBoxContents;  // biến thành viên mới
}

// Lớp Button dẫn xuất từ lớp Window
public class Button: Window
{   // hàm constructor thêm một thông số
    public Button(int top, int left): base(top, left) // gọi base
              //constructor
    {
    }

    // một phiên bản mới của hàm DrawWindow() trên lớp dẫn xuất
    public override void DrawWindow()
    {   Console.WriteLine("Vẽ một nút ấn tại: {0}, {1}\n", top, left);
    }
}

public class Tester
{   static void Main()
    {   // tạo một lớp cơ bản
        Window win = new Window(1,2);
        ListBox lb = new ListBox(3,4, "Ô liệt kê một mình!");
        Button but = new Button(5,6);
        win.DrawWindow();
```

```
            lb.DrawWindow();
            but.DrawWindow();

            Window[] winArray = new Window[3];
            winArray[0] = new Window(1,2);
            winArray[1] = new ListBox(3,4,"Ô liệt kê trên bản dãy");
            winArray[2] = new Button(5,6);

            for (int i = 0; i < 3; i++)
            {   winArray[i].DrawWindow();
            }
        }   // end Main
    }   // end Tester
```

Hình 6-5 cho thấy kết xuất của thí dụ 6-2:



Hình 6-5: Kết xuất Thí dụ 6-2

Bạn để ý xuyên suốt thí dụ này, chúng tôi cho đánh dấu các hàm mới bị phủ quyết sử dụng từ chốt **override**:

public override void DrawWindow()

Trình biên dịch biết sẽ xử trí thế nào đối với các hàm bị phủ quyết khi biên dịch các đối tượng này theo đa hình. Trình biên dịch chịu trách nhiệm theo dõi kiểu dữ liệu thực thụ của đối tượng vào lúc chót, ta gọi là "late binding" (gắn kết vào lúc chót), như vậy chính là hàm hành sự **ListBox.DrawWindow()** được triệu gọi khi qui chiếu **Window** chỉ về **ListBox**.

6.3.3 Sử dụng các từ chốt *new* và *override* để đánh số phiên bản

Trên C#, khi lập trình viên quyết định phủ quyết một hàm hành sự virtual thì anh (hoặc cô) ta sử dụng từ chốt **override**. Việc này giúp bạn "tháo khoán" (release) những phiên bản mới đối với đoạn mã của bạn, nếu có thay đổi trên lớp cơ bản cũng sẽ không phá vỡ đoạn mã trên các lớp dẫn xuất. Việc sử dụng từ chốt **override** giúp tránh vấn đề này.

Việc này là thế nào? Giả sử, vào lúc này lớp cơ bản **Window** do công ty A thiết kế. Và cũng giả sử các lớp **ListBox** và **RadioButton** lại do lập trình viên của công ty B viết sử dụng một bản sao lớp cơ bản **Window** của công ty A bán cho. Lập trình viên công ty B không tài nào biết người ta viết cái gì trong lớp cơ bản **Window** cách thiết kế ra sao, kể cả không thể biết những thay đổi gì mà công ty A sẽ tiến hành trong tương lai.[38]

Bây giờ, giả sử một lập trình viên công ty B quyết định thêm hàm hành sự **Sort()** vào lớp **ListBox**:

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

Việc này không gây ra vấn đề cho tới khi công ty A, tác giả lớp **Window**, cho ra phiên bản 2 của lớp **Window**, và điều tồi tệ xảy ra là lập trình viên công ty A lại thêm đúng hàm **Sort()** vào lớp public **Window**

```
public class Window
{
   public virtual void Sort() {…}
}
```

Trong các ngôn ngữ thiên đối tượng khác (như C++ chẳng hạn), hàm virtual mới **Sort()** trên lớp **Window** giờ đây sẽ hành động như là một hàm hành sự cơ sở đối với hàm virtual **Sort()** trên lớp **ListBox**. Trình biên dịch sẽ triệu gọi hàm **Sort()** trên **ListBox** trong khi bạn muốn triệu gọi **Sort()** của **Window**. Trên Java, nếu **Sort()** trên **Window** có kiểu trả về khác, trình nạp lớp sẽ cho rằng **Sort()** trên **ListBox** là một phủ quyết bất hợp lệ, và từ chối nạp lớp vào.

C# ngăn ngừa sự lẫn lộn này. Trên C#, một hàm virtual bao giờ cũng được xem như là nguồn gốc của virtual dispatch (phân bổ virtual), nghĩa là một khi C# tìm thấy một hàm

[38] Đây chính là cuộc chiến giữa Microsoft và Linux trong việc mã nguồn mở hay là không.

virtual, nó sẽ không nhìn lên cây đẳng cấp. Nếu một hàm virtual mới **Sort()** được thêm vào lớp **Window**, thì lối hành xử vào lúc chạy của **ListBox** không thay đổi.

Tuy nhiên, khi **ListBox** được biên dịch lại, trình biên dịch sẽ phát sinh một thông điệp:

```
…\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.
```

Muốn gỡ bỏ thông điệp này, lập trình viên phải cho biết ý đồ của mình. Y có thể đánh dấu hàm **ListBox.Sort()** với từ chốt **new** cho biết đây không phải *là một* phủ quyết của hàm virtual trên **Window**:

```
public class ListBox: Window
{
   public new virtual void Sort() {…}
}
```

Thay đổi trên sẽ gỡ bỏ thông điệp cảnh báo. Nếu, ngược lại, lập trình viên muốn phủ quyết hàm trong **Window**, y chỉ cần dùng từ chốt **override** để tỏ rõ lập trường:

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

Bạn để ý: Để tránh cảnh báo trên, có thể bạn sẽ bị lôi cuốn vào việc thêm từ chốt **new** lên tất cả các hàm virtual. Đây là một ý tưởng tồi. Khi **new** xuất hiện trên đoạn mã, nó phải sưu liệu phiên bản của đoạn mã. Nó chỉ về lớp cơ bản để xem điều gì bạn không phủ quyết. Sử dụng **new** sẽ phá ngầm sưu liệu này. Vả lại, sự hiện diện của dòng cảnh báo giúp thấy ngay ra vấn đề.

6.4 Các lớp trừu tượng (Abstract classes)

Mọi lớp dẫn xuất, còn được gọi là ***subclass***, *phải* cho thiết đặt hàm hành sự **DrawWindow()** riêng của mình, nhưng không có gì đòi hỏi phải làm như thế. Muốn các subclass thiết đặt một hàm hành sự thuộc lớp cơ bản, bạn phải chỉ định hàm cơ bản này là *abstract* (trừu tượng).

Một *hàm hành sự abstract sẽ không có phần thiết đặt thi công (implementation)*. Nó chỉ tạo ra một tên hàm và dấu ấn mà tất cả các lớp dẫn xuất đi sau phải làm phần thiết đặt thi công. Vả lại, làm cho một hoặc nhiều hàm của bất cứ lớp abstract nào thành abstract sẽ gây hiệu ứng phụ là làm cho lớp trở thành abstract.

Lớp abstract thiết lập một căn bản đối với những lớp được dẫn xuất, nhưng là *bất hợp lệ khi bạn cho thể hiện một lớp abstract.* Một khi đã khai báo một lớp nào đó là abstract, thì tuyệt đối cấm không cho hiển lộ trên lớp này, bằng cách dùng từ **new**.

Như vậy, nếu bạn chỉ định **DrawWindow** trên lớp **Window** là *abstract* thì bạn có thể dẫn xuất từ **Window**, nhưng bạn không thể hiển lộ bất cứ đối tượng **Window** nào. Và mỗi lớp dẫn xuất phải lo thiết đặt thi công hàm **DrawWindow()**. Nếu lớp dẫn xuất không thành công trong việc thiết đặt hàm abstract thì lớp này cũng sẽ trở thành abstract, và như thế không thể hiển lộ một đối tượng đối với lớp dẫn xuất này.

Muốn chỉ định một hàm hành sự là abstract, bạn cho đặt từ chốt **abstract** ở đầu phần định nghĩa hàm như sau:

```
abstract public void DrawWindow();
```

Vì hàm hành sự không có phần thiết đặt, nên không có cặp dấu {}, mà chỉ có dấu chấm phẩy (;).

Nếu một hoặc nhiều hàm hành sự là abstract, thì trong phần định nghĩa lớp cũng phải cho đánh dấu **abstract**, như sau:

```
abstract public class Window
```

Thí dụ 6-3, minh họa việc tạo ra lớp abstract **Window** và một hàm abstract **DrawWindow()**.

Thí dụ 6-3: Sử dụng một lớp và hàm hành sự abstract

```
using System;

abstract public class Window
{   // hàm constructor nhận hai trị số nguyên
    // cho biết vị trí trên màn hình
    public Window(int top, int left)
    {
        this.top = top;
        this.left = left;
    }

    // mô phỏng vẽ cửa sổ; để ý không có phần thi công
```

```
    abstract public void DrawWindow();

    // các biến thành viên sau đây là protected đối với
    // các hàm hành sự của lớp dẫn xuất
    protected int top;
    protected int left;
}

// Lớp ListBox dẫn xuất từ lớp Window
public class ListBox: Window
{   // hàm constructor của ListBox thêm một thông số listBoxContents
    public ListBox(int top, int left, string contents):
        base(top, left) // gọi base constructor
    {
        listBoxContents = contents;
    }

    // một phiên bản phủ quyết thiết đặt hàm abstract DrawWindow()
    public override void DrawWindow()
    {
        Console.WriteLine("Viết chuỗi lên ô liệt kê: {0}",
                          listBoxContents);
    }
    private string listBoxContents;  // biến thành viên mới
}

// Lớp Button dẫn xuất từ lớp Window
public class Button: Window
{   // hàm constructor
    public Button(int top, int left): base(top, left)
    {
    }

    // một phiên bản thiết đặt hàm abstract DrawWindow()
    public override void DrawWindow()
    {   Console.WriteLine("Vẽ một nút ấn tại: {0}, {1}\n", top, left);
    }
}

public class Tester
{   static void Main()
    {   Window[] winArray = new Window[3];
        winArray[0] = new ListBox(1,2, "Ô liệt kê thứ nhất ");
        winArray[1] = new ListBox(3,4,"Ô liệt kê thứ hai ");
        winArray[2] = new Button(5,6);

        for (int = 0; i < 3; i++)
        {   winArray[i].DrawWindow();
        }
    }  // end Main
}  // end Tester
```

Trong thí dụ 6-3, lớp **Window** giờ đây là một lớp abstract, do đó không thể thể hiện một đối tượng trên lớp này. Nếu bạn thay thế dòng lệnh:

```
winArray[0] = new ListBox(1,2, "Ô liệt kê thứ nhất ");
```

bởi dòng lệnh sau đây:

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

thì chương trình sẽ phát sinh ra thông báo sai lầm sau đây:

```
Cannot create an instance of the abstract class or interface 'Window'
```

Bạn có thể cho thể hiện các đối tượng của các lớp **ListBox** và **Button** vì các lớp này phủ quyết hàm abstract, làm cho những lớp này trở thành ***hiện thực (concrete,*** nghĩa là không abstract).

6.4.1 Những hạn chế của *abstract*

Mặc dầu chỉ định **DrawWindow()** là hàm abstract, ta không thể ép tất cả các lớp dẫn xuất thiết đặt thi công hàm này, cho nên đây là một giải pháp hạn chế đối với vấn đề. Nếu ta cho dẫn xuất một lớp khác từ **ListBox** (chẳng hạn **DropDownListBox**), không gì ép buộc lớp dẫn xuất này phải thiết đặt hàm **DrawWindow()** riêng của mình.

Bạn để ý: Trên C#, hàm **Window.DrawWindow()** không thể cung cấp phần thiết đặt, do đó ta không thể lợi dụng những thường trình thông dụng của **DrawWindow()** mà các lớp dẫn xuất khác có thể chia sẻ sử dụng.

Cuối cùng, bạn không nên xem lớp abstract chỉ là một "thủ thuật" thiết đặt thi công, nhưng phải coi nó như là một ý niệm trừu tượng cho thiết lập một "hợp đồng" đối với tất cả các lớp dẫn xuất. Nói cách khác, những lớp abstract mô tả những hàm hành sự public của những lớp sẽ làm cho hiện thực ý tưởng trừu tượng của lớp abstract.

Ý niệm một lớp abstract **Window** cho phép ta phác thảo ra những đặc tính và những lối hành xử chung phổ biến giữa tất cả các Windows, cho dù ta không bao giờ cố ý thể hiện sự trừu tượng của bản thân **Window**. Y niệm abstract sẽ được hiện thực trong những thể hiện khác nhau của **Window,** chẳng hạn browser window, frame, button, list box, drop down, v.v.. Một phương án thay thế việc sử dụng **abstract** là định nghĩa một giao diện (interface) mà chúng tôi sẽ đề cập ở chương 9.

6.4.2 Các lớp "vô sinh" (Sealed class)

Đối nghịch với abstract là **sealed**[39] (bị khóa miệng). Mặc dù một lớp abstract được tạo ra với ý đồ là có thể dẫn xuất từ đấy và cung cấp một khuôn mẫu (template) mà những subclass phải tuân theo, một sealed class không cho phép các lớp khác được dẫn xuất từ đấy. Từ chốt **sealed** đặt nằm trước phần khai báo lớp sẽ ngăn chặn việc dẫn xuất. Phần lớn các lớp thường xuyên được đánh dấu **sealed** để tránh tai nạn kế thừa vô ý thức.

Nếu khai báo của **Window** trên thí dụ 6-3 bị thay đổi từ **abstract** qua **sealed** (gỡ bỏ luôn từ chốt **abstract** khỏi hàm **DrawWindow**) chương trình sẽ không biên dịch được. Nếu bạn cố công xây dựng dự án này, trình biên dịch sẽ ra thông báo sai lầm như sau:

```
'ListBox' cannot inherit from sealed class 'Window'
```

Sau đây là những qui tắc hướng dẫn cách sử dụng các lớp "vô sinh":

- Chỉ dùng lớp "vô sinh" nếu thấy không cần thiết tạo những lớp dẫn xuất. Bạn nhớ cho là một lớp không thể được dẫn xuất từ một lớp bị sealed.

- Chỉ sử dụng lớp bị "vô sinh" nếu chỉ có những thuộc tính và hàm hành sự static trên một lớp. Thí dụ đoạn mã sau đây cho thấy một lớp "vô sinh" được định nghĩa một cách đúng đắn:

```
public sealed class Runtime
{
 // Hàm private constructor ngăn không cho phép tạo lớp
   private Runtime();

 // Static method.
 public static string GetCommandLine()
 {
  // Thiết đặt đoạn mã ở đây.
 }
}
```

6.5 Nguồn gốc của tất cả các lớp: Object

Tất cả các lớp C#, bất cứ kiểu dữ liệu nào, cũng đều được dẫn xuất từ **System. Object**. Kể cả những kiểu trị (value type).

---

[39] Theo nguyên tắc "sealed" là "bị khoá miệng" nhưng dịch như thế xem ra kỳ kỳ thế nào đấy. Nên dựa trên công năng, chúng tôi dịch là "vô sinh" cho biết loại lớp này không thể "đẻ" ra lớp khác.

Một lớp cơ bản là lớp cha-mẹ kế cận nhất của một lớp dẫn xuất. Một lớp dẫn xuất cũng có thể là lớp cơ bản cho những lớp dẫn xuất đi sau, tạo ra một "cây kế thừa" (inheritance tree, hoặc hierarchy), giống như gia phả các dòng họ. Lớp gốc (root class) là lớp cao nhất trên cây kế thừa, với lớp gốc nằm trên cao, còn các lớp dẫn xuất nằm ở dưới (ngược lại với cây cối trong thiên nhiên). Trên C#, lớp gốc này là **Object**. Do đó, lớp cơ bản coi như nằm trên lớp dẫn xuất.

Object cung cấp một số hàm hành sự mà các lớp dẫn xuất ở dưới có thể cho phủ quyết. Các hàm này bao gồm: **Equals()** xác định liệu xem hai đối tượng có giống nhau hay không; hàm **GetType()** trả về kiểu dữ liệu của một đối tượng nào đó; và **ToString()** trả về một chuỗi tượng trưng cho đối tượng. Bảng 6-1 dưới đây tóm lược các hàm hành sự của lớp **Object.**

Bảng 6-1: Các hàm hành sự của Object

Equals	**public static bool Equals (object objA, object objB)**: Overloaded. Xác định liệu xem hai đối tượng **Object** , objA và objB có bằng nhau hay không.
GetHashCode	**public virtual int GetHashCode()**: Cho phép các đối tượng tự cung cấp hàm băm (hash function) riêng cho mình đem dùng trong các giải thuật băm (hashing algorithms) và cấu trúc dữ liệu giống như một hash table.
GetType	**public Type GetType()** Cho biết kiểu dữ liệu của đối tượng hiện hành.
ReferenceEquals	**public static bool ReferenceEquals(object *objA*, object *objB*)** Xác định liệu xem hai đối tượng **Object** được đưa ra có cùng một thể hiện hay không, nghĩa là hai qui chiếu có chĩa về cùng một thể hiện hay không.
ToString	**public virtual string ToString()** Trả về một chuỗi tượng trưng cho đối tượng hiện hành **Object**.

Thí dụ 6-4 sau đây minh họa việc sử dụng hàm hành sự **ToString()** được kế thừa từ **Object** cũng như việc các kiểu dữ liệu "bẩm sinh" chẳng hạn **int** có thể được xem như là kế thừa từ **Object**.

Thí dụ 6-4: Kế thừa từ Object

```
using System;

public class SomeClass
{
```

```
    public SomeClass(int val)
    {   value = val;
    }

    public virtual string ToString()
    {   return value.ToString();
    }
    private int value;

}
public class Tester
{   static void Main()
    {   int i = 5;
        Console.WriteLine("Trị của i là: {0}", i.ToString());

        SomeClass s = new SomeClass(7);
        Console.WriteLine("Trị của s là: {0}", s.ToString());
    }   // end Main
}   // end Tester
```

Kết xuất
Trị của i là: 5
Trị của s là 7

Phần sưu liệu đối với **Object.ToString** cho thấy dấu ấn:

```
public virtual string ToString();
```

Đây là một hàm virtual public trả về một chuỗi và không nhận thông số. Tất cả các kiểu dữ liệu bẩm sinh, chẳng hạn **int**, được dẫn xuất từ **Object**, nên có thể triệu gọi các hàm của **Object.**

Thí dụ 6-4 phủ quyết hàm virtual đối với lớp **SomeClass**, là trường hợp thông thường, như vậy hàm **ToString()** của lớp **SomeClass** sẽ trả về một trị mang ý nghĩa. Nếu bạn biến hàm bị phủ quyết thành chú giải, thì hàm **ToString()** của lớp cơ bản sẽ được triệu gọi, sẽ thay đổi phần kết xuất như sau:

```
Trị của s là SomeClass
```

Do đó, trị mặc nhiên được trả về là một chuỗi mang nội dung là tên lớp cơ bản. Các lớp khỏi phải khai báo tường minh được dẫn xuất từ **Object**; tính kế thừa đã hiểu ngầm như thế.

Bây giờ, ta xem một hàm phủ quyết ToString(), sử dụng đến từ chốt override, trong một lớp KhachHang, chẳng hạn:

```
public class KhachHang
{
    private string tenHo;
    protected decimal ketSo;

    public override string ToString()
    {
        string ketQua = "Khách Hàng: " + tenHo;
        ketQua += ", kết số: " + ketSo;
        return ketQua;
    }
    …
}
```

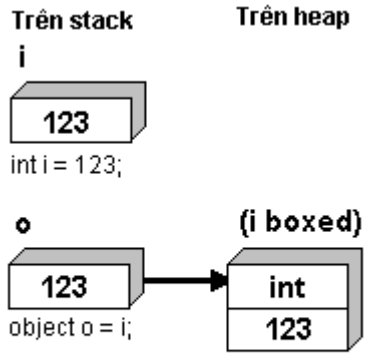
Nếu đối tượng của lớp KhachHang là **kh**, và trị của tenHo là "Đoàn Dự" và ketSo là 12.45, thì khi ta viết một lệnh như sau:

```
Console.WriteLine(kh.ToString());
```

kết xuất sẽ là:

```
Khách hàng: Đoàn Dự, kết số: 12,45
```

6.6 Boxing và Unboxing các kiểu dữ liệu



Hình 6-6: Boxing kiểu dữ liệu qui chiếu

Boxing (đóng hộp) và *unboxing* (mở hộp, rã hộp) là cơ chế cho phép các kiểu trị (value type, chẳng hạn **int**) được đối xử như là những kiểu qui chiếu (đối tượng). Trị được "đóng hộp" trong lòng một đối tượng **Object**, và về sau có thể được "mở hộp" cho trở về lại kiểu trị. Chính cơ chế này cho phép bạn triệu gọi hàm **ToString()** đối với trị số nguyên trên thí dụ 6-4.

6.6.1 Boxing được hiểu ngầm

Boxing được xem như là một chuyển đổi (conversion) hiểu ngầm một kiểu trị thành một kiểu **Object**. Boxing một trị sẽ cho cấp phát một thể hiện **Object**, rồi sao trị lên thể hiện đối tượng mới, như theo hình 6-6.

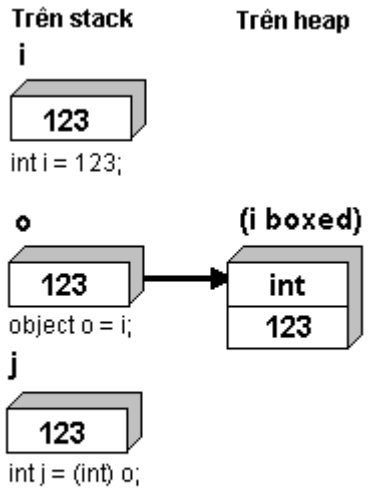
Boxing được hiểu ngầm khi bạn cung cấp một kiểu trị và chờ đợi một qui chiếu, và trị được ngầm đóng hộp. Thí dụ, nếu bạn gán một kiểu dữ liệu bẩm sinh, một **int** chẳng

hạn , cho một biến kiểu **Object** (là hợp lệ, vì **int** được dẫn xuất từ **Object**), thì lúc này trị sẽ được đóng hộp, như được minh họa sau đây:

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

Console.WriteLine() chờ đợi một đối tượng, chứ không phải một số nguyên. Muốn thích ứng với hàm trên, kiểu dữ liệu **int** tự động được đóng hộp bởi CLR, và **ToString()** được triệu gọi đối với đối tượng kết xuất. Cơ chế này cho phép bạn tạo ra những hàm hành sự nhận vào những đối tượng như là thông số bất kể kiểu gì, kiểu trị hoặc kiểu qui chiếu, hàm sẽ hoạt động được.

6.6.2 Unboxing bắt buộc phải tường minh



Hình 6-7: Boxing rồi sau đó Unboxing

Muốn cho một đối tượng bị đóng hộp trở lại nguyên hình, bạn phải unboxing một cách tường minh (explicit). Bạn thực hiện việc này theo hai bước:

1. Phải bảo đảm là thể hiện đối tượng (object instance) là một trị bị đóng hộp của một kiểu trị.

2. Sao trị từ thể hiện qua biến kiểu trị.

Hình 6-7. minh họa việc unboxing

Muốn việc unboxing thành công, đối tượng bị mở hộp phải là một qui chiếu về đối tượng được tạo ra bằng cách boxing một trị kiểu dữ liệu nào đó. Thí dụ 6-5 minh họa boxing và unboxing.

Thí dụ 6-5: Boxing và Unboxing

```
using System;
public class UnboxingTest
{
```

```
    public static void Main()
    {   int i = 123;

        // Boxing
        object o = i;

        // Unboxing (phải explicit)
        int j = (int) o;

        Console.WriteLine("j: {0}", j);
    }   // end Main
}   // end UnboxingTest
```

Thí dụ 6-5 tạo một số nguyên **i** và ngầm đóng hộp trị này khi nó được đem gán cho đối tượng **o**. Trị bị đóng hộp lại được mở hộp rồi được đem gán cho một biến **int**, rồi sau đó nội dung biến này được in ra.

Điển hình là bạn sẽ gói gọn (wrap) một tác vụ unboxing trong một khối **try**, như sẽ được giải thích ở chương 12. Nếu đối tượng bị mở hộp là null hoặc qui chiếu về một đối tượng kiểu dữ liệu khác thì một sai lầm biệt lệ **InvalidCastException** sẽ được tung ra.

6.7 Lớp lồng nhau

Các lớp đều thường có những thành viên, và có khả năng những thành viên của lớp lại thuộc kiểu dữ liệu tự tạo (user-defined type) khác. Do đó, một lớp **Button** có thể có một biến thành viên kiểu **Location** (vị trí), và một lớp **Location** có thể chứa những biến thành viên kiểu **Point** (điểm). Cuối cùng, lớp **Point** có thể chứa những biến thành viên kiểu **int**.

Đôi lúc, lớp bị chứa (thường được gọi là helper class) nằm trong chỉ có thể hiện diện để giúp đở lớp nằm ngoài, và như vậy không lý do gì phải làm cho nhìn thấy được. Bạn có thể định nghĩa lớp helper trong lòng của định nghĩa lớp nằm ngoài. Lớp được gói cho nằm trong được gọi là một lớp *nằm lồng (nested class),* còn lớp chứa lớp nằm lồng được gọi là lớp *nằm ngoài (outer class)*.

Lớp nằm lồng có lợi điểm có thể truy xuất tất cả các thành viên của lớp nằm ngoài. Một hàm hành sự của một lớp nằm lồng có thể truy xuất những biến thành viên private của lớp nằm ngoài. Ngoài ra, lớp nằm lồng có thể được cất giấu khỏi sự dòm ngó của tất cả các lớp khác - nghĩa là có thể biến thành private đối với lớp nằm ngoài.

Cuối cùng, một lớp nằm lồng public sẽ được truy xuất trong phạm vi (scope) của lớp nằm ngoài. Nếu **Outer** là lớp nằm ngoài, và **Nested** là lớp nằm lồng public (còn gọi là

inner class), bạn qui chiếu đến **Nested** bằng cách gọi **Outer.Nested**, với lớp nằm ngoài giữ vai trò (ít hoặc nhiều) như là một namespace hoặc scope.

Thí dụ 6-6 minh họa việc sử dụng một lớp nằm lồng

Thí dụ 6-6: Sử dụng một lớp nằm lồng

```
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("Vẽ ra numerator: {0}", f.numerator);
         Console.WriteLine("Vẽ ra denominator: {0}", f.denominator);
      }
   {
   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);
   }  // end Main
}  // end Tester
```

Bạn thấy lớp **FractionArtist** (in đậm) nằm lồng trong lớp **Fraction**. Lớp **Fraction Artist** chỉ có duy nhất một thành viên là hàm **Draw()**. Nhiệm vụ của **FractionArtist** là vẽ phân số lên màn hình. Ơ đây chúng tôi cho thay thế bởi hai dòng lệnh Console.Write Line. Điểm đáng nhấn mạnh là **Draw()** có thể truy xuất các biến thành viên private của **Fraction**: f.numerator và f.denominator, nếu không nằm lồng thì nó không thể truy xuất được.

Bạn để ý, trên **Main()** muốn khai báo một thể hiện của lớp nằm lồng này, bạn phải cho biết tên kiểu dữ liệu của lớp nằm ngoài:

```
Fraction.FractionArtist fa = new Fraction.FractionArtist();
```

Mặc dù **FractionArtist** là public, nhưng nằm trong phạm vi của lớp **Fraction** mà thôi.

Chương 7

Nạp chồng tác tử[40]

Mục tiêu khi thiết kế ngôn ngữ C# là làm thế nào các lớp tự tạo (user-defined class) bởi người sử dụng cũng sẽ có tất cả những chức năng của các kiểu dữ liệu bẩm sinh (built-in type). Thí dụ, giả sử bạn đã định nghĩa một kiểu dữ liệu tượng trưng phân số (fraction). Bảo đảm rằng lớp này có tất cả các chức năng của các kiểu dữ liệu bẩm sinh, có nghĩa là bạn có thể thực hiện những phép toán số học trên những thể hiện của lớp phân số (nghĩa là cộng, nhân, chia hai phân số v.v..) và có thể chuyển đổi số phân đi đi về về với kiểu dữ liệu bẩm sinh, chẳng hạn **int.** Lẽ dĩ nhiên, bạn có thể thiết đặt những hàm hành sự đối với từng tác vụ kể trên, và triệu gọi chúng bằng cách viết dòng lệnh như sau:

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

Mặc dù dòng lệnh trên chạy được, nhưng thấy cách viết nó sao sao ấy, và không giống cách các kiểu dữ liệu bẩm sinh được sử dụng. Có thể tốt hơn nếu chúng ta viết như sau:

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

Dòng lệnh trên có vẻ tự nhiên hơn và nhất quán với những kiểu dữ liệu bẩm sinh, chẳng hạn **int**, cộng với nhau.

Trong chương này, bạn học cách bổ sung những tác tử chuẩn vào các kiểu dữ liệu tư tạo của bạn. Ngoài ra, bạn cũng sẽ học cách thêm các tác tử chuyển đổi, do đó các kiểu dữ liệu tự tạo của bạn có thể chuyển đổi (ngầm hiểu hoặc tường minh) qua các kiểu dữ liệu khác.

7.1 Sử dụng từ chốt *operator*

Trên C#, các tác tử [41](operator) là những *hàm hành sự static*, theo đấy trị trả về tượng trưng cho kết quả của một tác vụ với những thông số là những tác tố (operand). Khi bạn

[40] Chúng tôi dịch "operator overloading" là "nạp chồng tác tử". Một số dịch giả lại cho dịch "overload" là "quá tải" giống như xe chở quá tải. Ý overload ở đây là một hàm (hoặc tác tử) có thể "kiêm nhiệm" thêm chức năng, giống như mấy ông cán bộ lãnh đạo vậy.

tạo một tác tử cho một lớp, bạn bảo rằng bạn đã "nạp chồng" (overload) tác tử, giống như bạn nạp chồng bất cứ hàm thành viên nào. Do đó, muốn nạp chồng tác tử cộng (+), bạn có thể viết:

```
public static Fraction operator+(Fraction lhs, Fraction rhs).
```

lhs tắt chữ "lefthand side" nghĩa là phía tay trái dấu bằng (=), còn **rhs** tắt chữ "righthand side" phía tay phải dấu bằng. Đây là qui ước do người viết đặt ra để đặt tên cho các thông số.

Cú pháp C# liên quan đến nạp chồng một tác tử là đặt từ chốt **operator** nằm trước tác tử cần nạp chồng. **operator** được gọi là một *method modifier*. Như vậy, muốn nạp chồng tác tử cộng (+), bạn viết **operator+**. Khi bạn viết:

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

tác tử được nạp chồng sẽ được triệu gọi, với **firstFraction** được trao qua như là đối mục đầu tiên, và **secondFraction** như là đối mục thứ hai. Khi trình biên dịch thấy biểu thức:

```
firstFraction + secondFraction
```

thì nó dịch biểu thức trên thành: Fraction.operator+(firstFraction, secondFraction)

Kết quả là một **Fraction** mới được trả về, và được gán cho đối tượng **Fraction** mang tên **theSum** (tổng cộng).

7.2 Hỗ trợ các ngôn ngữ .NET khác

C# cung cấp khả năng nạp chồng tác tử đối với các lớp của bạn, cho dù đây không nằm trong Common Language Specification (CLS). Các ngôn ngữ .NET khác, VB.NET chẳng hạn, có thể không hỗ trợ nạp chồng tác tử, và điểm quan trọng là bảo đảm lớp của bạn hỗ trợ những hàm thay thế mà các ngôn ngữ khác có thể triệu gọi cho ra cùng kết quả.

Do đó, nếu bạn nạp chồng tác tử cộng (+), có thể bạn cũng muốn cung cấp một hàm **add()** cho ra kết quả giống đúc. Nạp chồng tác tử phải là một cú pháp ngắn gọn, chứ không phải là lối đi duy nhất đối với các đối tượng của bạn để hoàn thành một công tác nào đó.

---

[41] Trong toán học người ta thường dịch "operator" là "toán tử", còn "operand" là "toán hạng". Chúng tôi thì chọn dịch là "tác tử" và "tác tố" cho nhất quán với "tác vụ" (operation).

7.3 Tạo những tác tử hữu ích

Nạp chồng tác tử có thể làm cho cách viết đoạn mã tự nhiên hơn, và cho phép bạn hành động giống như với các kiểu dữ liệu bẩm sinh. Nó cũng có thể làm cho đoạn mã của bạn phức tạp khó quản lý nếu bạn phá vỡ sự diễn đạt thông dụng trong việc sử dụng tác tử. Bạn nên tránh cách sử dụng tác tử theo phong cách mới mang tính lập dị.

Thí dụ, mặc dù có thể bạn muốn nạp chồng tác tử tăng một (++) cho thực hiện trên lớp **Employee** (nhân viên) để triệu gọi một hàm hành sự lo tăng cấp bậc lương nhân viên lên một bậc, điều này sẽ gây lúng túng đối với người sử dụng lớp của bạn. Bạn nên sử dụng nạp chồng tác tử một cách dè xẻn, và chỉ dùng nó khi nào bạn thấy ý nghĩa tác tử được nạp chồng đã rõ ràng và nhất quán với hoạt động của các kiểu dữ liệu bẩm sinh.

7.4 Cặp tác tử lô gic

Việc cho nạp chồng tác tử bằng (==, equals operator) là khá phổ biến, để trắc nghiệm liệu xem hai đối tượng có bằng nhau hay không (với điều kiện tác tử bằng được định nghĩa đối với đối tượng của bạn). C# yêu cầu rằng nếu bạn nạp chồng tác tử "==", thì bạn cũng phải nạp chồng luôn tác tử "!=" (không bằng hoặc khác biệt). Cũng tương tự như thế, phải nạp chồng cặp tác tử "<" (nhỏ thua) và ">" (lớn hơn), cũng như cặp '<=" (nhỏ thua hoặc bằng) và ">=" (lớn hơn hoặc bằng).

7.5 Tác tử Equals

Nếu bạn nạp chồng tác tử bằng (==), người ta khuyên bạn cũng nên phủ quyết (override) luôn hàm hành sự virtual **Equals()** thuộc lớp **Object**, và chuyển chức năng của hàm này về lại tác tử bằng. Điều này cho phép lớp của bạn mang tính đa hình và tạo tương thích (compatibility) với các ngôn ngữ .NET khác không hỗ trợ nạp chồng tác tử (chỉ hỗ trợ nạp chồng hàm hành sự - method overloading - mà thôi). Các lớp thư viện FCL (Framework Class Library) sẽ không dùng tác tử được nạp chồng nhưng sẽ chờ đợi các lớp của bạn thiết đặt những hàm hành sự nằm sau. Thí dụ, lớp **ArrayList** chờ đợi bạn thiết đặt hàm **Equals().**

Lớp **object** thiết đặt hàm **Equals()** với dấu ấn:

```
public override bool Equals(object o)
```

Bằng cách phủ quyết hàm này, bạn cho phép lớp **Fraction** ứng xử một cách đa hình

đối với tất cả các đối tượng khác. Trong lòng thân hàm **Equals()**, bạn sẽ phải bảo đảm bạn đang so sánh với một đối tượng **Fraction** khác, và nếu đúng thế, bạn có thể trao phần thiết đặt cùng với định nghĩa tác tử bằng mà bạn đã viết ra.

```
public override bool Equals(object o)
{
    if (!(o is Fraction))
    {
        return false;
    }
    return this == (Fraction) o;
}
```

Tác tử **is** dùng kiểm tra liệu xem kiểu dữ liệu của một đối tượng vào lúc chạy có tương thích với tác tố (trong trường hợp này là **Fraction**) hay không. Do đó, **o** là một đối tượng **Fraction** sẽ được cho là **true** nếu trong thực tế là một kiểu dữ liệu tương thích với **Fraction**.

7.6 Chuyển đổi các tác tử

C# sẽ *ngầm* chuyển đổi (conversion) **int** thành **long**, và cho phép bạn chuyển đổi ngược lại **long** thành **int** một cách tường minh (explicit). Việc chuyển đổi **int** thành **long** là *ngầm hiểu (implicit)*, vì ta biết rằng bất cứ **int** nào cũng vào khớp với cách biểu diễn của một **long** trong ký ức, vì sức chứa của **long** bao giờ cũng lớn hơn sức chứa của **int**. Còn hành động ngược lại, từ **long** qua **int**, phải được thực hiện *một cách tường minh* (sử dụng ép kiểu, cast) vì ta có thể đánh mất thông tin khi thi hành việc chuyển đổi:

```
int myInt = 5;
long myLong;
myLong = myInt;         // implicit
myInt = (int) myLong;   // explicit
```

Bạn cũng muốn có chức năng như thế đối với lớp Fraction của bạn. Có một **int** nào đó, bạn có thể hỗ trợ việc một chuyển đổi ngầm về một phân số, vì bất cứ trị số nguyên nào cũng bằng trị này trên 1 (nghĩa là 15==15/1 chẳng hạn).

Cho một số phân nào đó, có thể bạn muốn cung cấp một chuyển đổi tường minh trở về một số nguyên, biết rằng có thể mất đi một phần trị nào đó. Do đó, có thể bạn muốn chuyển đổi 9/4 thành số nguyên 2, biết trước sẽ mất đi 0,25.

Từ chốt **implicit** sẽ được dùng khi bảo đảm việc chuyển đổi thành công, không thông tin nào bị đánh mất; bằng không thì dùng từ chốt **explicit**.

Thí dụ 7-1 sau đây minh họa việc thiết đặt implicit và explicit conversion và một vài tác tử của lớp **Fraction.** (Mặc dù ta dùng Console.WriteLine để in ra thông điệp báo cho biết ta vào hàm nào, cách hay nhất để theo dõi vết là dùng debugger. Bạn có thể đặt một breakpoint tại mỗi câu lệnh trắc nghiệm, rồi chui vào đoạn mã, quan sát triệu gọi hàm constructor khi chúng xảy ra.)

Thí dụ 7-1: Định nghĩa việc chuyển đổi và các tác tử đối với tác tử lớp Fraction

```
using System;

public class Fraction
{   public Fraction(int numerator, int denominator)
    {   Console.WriteLine("Trên Fraction Constructor(int, int)");
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public Fraction(int wholenumber)
    {   Console.WriteLine("Trên Fraction Constructor(int)");
        this.numerator = wholenumber;
        this.denominator = 1;
    }

    public static implicit operator Fraction(int theInt)
    {   System.Console.WriteLine("Trên Implicit Conversion về Fraction");
        return new Fraction(theInt, 1);
    }

    public static explicit operator Fraction(Fraction theFraction)
    {   System.Console.WriteLine("Trên Explicit Conversion về Int");
        return theFraction.numerator / theFraction.denominator;
    }

    public static bool operator==(Fraction lhs, Fraction rhs)
    {   Console.WriteLine("Trên operator ==");
        if (lhs.denominator == rhs.denominator &&
    lhs.numerator == rhs.numerator)
        {   return true;
        }
        // ở đây, đoạn mã thụ lý không phải số phân
        return false;
    }

    public static bool operator!=(Fraction lhs, Fraction rhs)
    {   Console.WriteLine("Trên operator !=");
        return !(lhs==rhs);
    }

    public override bool Equals(object o)
    {   Console.WriteLine("Trên hàm Equals");
        if (! (o is Fraction)
```

```
        {   return false;
        }
        return this == (Fraction) o;
    }

    public static Fraction operator+(Fraction lhs, Fraction rhs)
    {   Console.WriteLine("Trên operator +");
        if(lhs.denominator == rhs.denominator)
        {   return new Fraction(lhs.numerator + rhs.numerator,
            lhs.denominator);
        }
        // giải pháp đơn giản đối với những số không giống phân số
        // ½ + ¾ == (1*4) + (3*2) / (2*4) == 10/8
        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;
    }
    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("f3 + 5 = f4: {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 constructor: một hàm nhận 2 thông số là tử số (numerator) và mẫu số (denominator), còn hàm kia nhận một số nguyên (whole number).

Tiếp theo là khai báo của hai chuyển đổi tác tử. Tác tử chuyển đổi đầu tiên biến một số nguyên thành một số phân:

```
public static implicit operator Fraction(int theInt)
{   System.Console.WriteLine("Trên Implicit Conversion về Fraction");
    return new Fraction(theInt, 1);
}
```

Chuyển đổi này được ghi với từ chốt implicit, vì bất cứ số nguyên (int) nào cũng có thể chuyển đổi thành một Fraction bằng cách cho tử số về int và mẫu số về 1. Bạn giao trách nhiệm này cho hàm constructor thứ hai.

Tác tử chuyển đổi thứ hai là chuyển đổi tường minh của một Fraction thành số nguyên:

```
public static explicit operator Fraction(Fraction theFraction)
{   System.Console.WriteLine("Trên Explicit Conversion về Int");
    return theFraction.numerator / theFraction.denominator;
}
```

Vì thí dụ này dùng đến chia theo số nguyên (integer division), nên nó xén trị chỉ chừa lại số nguyên. Do đó, nếu phân số là 15/16, kết quả số nguyên sẽ là 1. Một tác tử chuyển đổi tinh vi hơn sẽ thực hiện việc qui tròn (rounding).

Sau đó, đến tác tử bằng (==) và không bằng (!=). Bạn nhớ phải thiết đặt cả cặp (==, !=). Bạn đã định nghĩa sự bằng nhau về trị đối với Fraction bằng cách bảo các tử số và mẫu số phải khớp nhau. Đối với định nghĩa này thì 3/4 và 6/8 không bằng nhau (thật ra là bằng nhau). Một lần nữa, một thiết đặt tinh vi hơn sẽ giảm những phân số này đi và sẽ thấy là bằng nhau.

Bạn cho bao gồm một hàm phủ quyết Equals() như vậy các đối tượng Fraction có thể được cư xử một cách đa hình với bất cứ đối tượng nào khác. Bạn ủy quyền việc định trị bằng cho tác tử bằng.

Lẽ dĩ nhiên là lớp Fraction phải thiết kế những tác tử toán số học, như cộng, trừ, nhân, chia, v.v.. Để cho đơn giản thí dụ chúng tôi chỉ thi công tác tử cộng mà thôi, và ở đây, chúng tôi cũng giản lược đi khá nhiều. Bạn kiểm tra liệu xem những mẫu số có bằng nhau không; nếu bằng thì cộng lại các 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 không bằng thì bạn nhân chéo lại với nhau (bạn xem thí dụ trong phần chú giải):

```
// giải pháp đơn giản đối với những số không giống phân số
// ½ + ¾ == (1*4) + (3*2) / (2*4) == 10/8
int firstProduct = lhs.numerator * rhs.denominator;
int secondProduct = rhs.numerator * lhs.denominator;
return new Fraction(firstProduct + secondProduct,
         lhs.denominator * rhs.denominator);
```

Cuối cùng, để có thể gỡ rối lớp Fraction, đoạn mã được viết làm sao Fraction có khả năng trả về trị của nó dưới dạng chuỗi theo dạng thức numerator/denominator.

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

Bạn tạo một đối tượng chuỗi mới bằng cách triệu gọi hàm ToString() đối với tử số. Vì numerator là một đối tượng, điều này làm cho trình biên dịch ngầm cho đóng hộp con số nguyên (tạo một đối tượng rồi sau đó triệu gọi ToString() làm việc trên đối tượng này, trả về một chuỗi tượng trưng cho tử số. Bạn ghép chuỗi "/" rồi lại ghép với chuỗi kết quả do việc triệu gọi **ToString**() đối với mẫu số.

Với lớp Fraction đã hình thành xong, bạn thử cho trắc nghiệm. Trắc nghiệm đầu tiên là tạo những phân số đơn giản:

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

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

Phần kết xuất của đoạn mã trên đúng theo ý muốn của bạn: triệu gọi hàm constructor và in kết quả sử dụng Console.WriteLine():

```
Trên Fraction Constructor (int, int)
f1: 3 / 4
Trên Fraction Constructor (int, int)
f2: 2 / 4
```

Các dòng lệnh kế tiếp triệu gọi static operator+ với mục đích là cộng hai phân số và trả về tổng cộng vào một phân số mới:

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

Khi xét kết quả bạn sẽ thấy operator+ hoạt động thế nào:

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

Phần trắc nghiệm kế tiếp là cộng một int lên Fraction f3 rồi gán kết quả lên một Fraction f4 mới:

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

Kết quả cho thấy những bước chuyển đổi:

```
Trên implicit conversion về Fraction
Trên Fraction Constructor(int)
Trên operator+
Trên Fraction Constructor(int, int)
f3 + 5 = f4: 25/4
```

Bạn để ý, tác tử chuyển đổi hiểu ngầm được triệu gọi để chuyển đổi 5 thành phân số. Trong lệnh return từ tác tử chuyển đổi ngầm, hàm constructor của Fraction được gọi vào, tạo ra phân số 5/1, rồi trao phân số mới này cùng với Fraction f3 cho tác tử operator+, và kết quả tổng cộng được trao qua cho hàm constructor đối với f4.

Trong phần trắc nghiệm cuối cùng, một phân số mới f5 được tạo ra, và bạn trắc nghiệm liệu xem có bằng f2 hay không.

```
Fraction f5 = new Fraction(2,4);
if (f5 == f2)
{   Console.WriteLine("f5: {0} == f2: {1}",
    f5.ToString(), f2.ToString());
}
```

Phần kết xuất sau đây cho thấy f5 được tạo ra, rồi sau đó triệu gọi tác tử bằng được phủ quyết:

```
Trên Fraction Constructor(int, int)
Trên operator ==
f5: 2/4 == f2: 2/4
```

Chương 8

Cấu trúc Struct[42]

Lớp sẽ được dùng để tạo phần lớn những đối tượng. Tuy nhiên, đôi khi người ta cũng muốn tạo dựng những đối tượng hoạt động tương tự như những kiểu dữ liệu bẩm sinh, ít dùng ký ức, có thể được cấp phát nhanh, và không tốn nhiều ký ức do việc qui chiếu. Trong trường hợp này, ta dùng đến kiểu dữ liệu trị, được thực hiện bằng cách khai báo một ***struct*** trên C#.

Một ***struct*** là một kiểu dữ liệu tự tạo (user-defined type, UDT), "nhẹ cân" có thể thay thế lớp. Struct cũng tương tự như lớp, nghĩa là một cấu trúc dữ liệu có thể chứa các thành viên dữ liệu và thành viên hàm (hàm khởi dựng, thuộc tính, hàm hành sự, vùng mục tin, tác tử, kiểu dữ liệu nằm lồng, và indexer).

Tuy nhiên, cũng có nhiều khác biệt giữa lớp và struct. Khác với lớp (là kiểu dữ liệu qui chiếu), struct là một loại dữ liệu kiểu trị nghĩa là không cần được cấp phát ký ức trên heap. Một biến struct sẽ chứa trực tiếp dữ liệu của struct, trong khi một biến kiểu lớp thì lại chứa một qui chiếu về dữ liệu được biết như là một đối tượng. Ngoài ra, *struct không hỗ trợ kế thừa và hàm hủy (destructor).*

Do đó, struct chỉ hữu ích đối với những đối tượng không đòi hỏi ý nghĩa qui chiếu (reference semantic), nghĩa là những cấu trúc dữ liệu nhỏ mang ý nghĩa trị (value semantics). Các số phức (complex number), các điểm trên một hệ thống toạ độ, hoặc cặp key-value trên dictionary là những thí dụ tốt về struct. Người ta thỏa thuận chỉ nên sử dụng struct đối với những kiểu dữ liệu nhỏ, đơn giản, có những hành xử và đặc tính giống như những kiểu dữ liệu "bẩm sinh" (built-in type), như int, double hoặc bool.

Dùng struct trong các bản dãy sẽ hữu hiệu hơn về mặt sử dụng ký ức. Tuy nhiên, struct lại kém hiệu quả khi đem sử dụng trong các tập hợp (collection). Vì collection chờ đợi nhiều ở qui chiếu, nên nếu dùng struct thì phải dùng đến cơ chế đóng hộp (boxing) và rã hộp (unboxing) tốn hao ký ức (overhead). Nếu collection thuộc loại đồ sộ thì nên sử dụng lớp.

Trong chương này, bạn sẽ học cách định nghĩa struct, sử dụng struct thế nào, và sử dụng hàm constructor để khởi gán trị của struct.

---

[42] Struct tắt chữ structure. Tạm thời chúng tôi không dịch, để yên từ này.

8.1 Struct được khai báo thế nào?

Struct mang cú pháp khai báo tương tự như với một lớp:

[*attributes*] [*acces-modifiers*] **struct** *identifier* [:*interface-list*] {*struct-members*}

Vì cú pháp giống như với lớp, nên chúng tôi không giải thích chi thêm. Thí dụ 8-1, minh họa định nghĩa một struct, **Location**, cho biết một điểm tọa lạc trên một mặt bằng 2 chiều. Bạn để ý struct **Location** được khai báo giống đúc một lớp, ngoại trừ việc sử dụng từ chốt **struct**. Ngoài ra, bạn để ý hàm constructor **Location** có hai số nguyên và gán trị của chúng cho những thể hiện thành viên **x** và **y**. Các thuộc tính tọa độ **x** và **y** được khai báo là những thuộc tính.

Thí dụ 8-1: Tạo một struct

```
using System

public struct Location
{
    // hàm khởi dựng
    public Location(int xCoordinate, 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);
    }
```

```
    private int xVal;
    private int yVal;
}

public class Tester
{
    public void myFunc(Location loc)
    {
        loc.x = 50;
        loc.y = 100;
        Console.WriteLine("In myFunc loc: {0}", loc);
    }  // end myFunc

    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);
    }  // end Main
}
```

Kết xuất
Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 200, 300

8.1.1 Hàm khởi dựng và kế thừa

Khác với lớp, *struct không hỗ trợ tính kế thừa*. Các đối tượng struct được dẫn xuất từ lớp **object**, một cách hiểu ngầm (như đối với tất cả các kiểu dữ liệu C#, kể cả những kiểu dữ liệu bẩm sinh), nhưng struct không thể kế thừa từ bất cứ lớp hoặc struct nào khác, và cũng không thể dùng làm cơ sở (base) cho một lớp. Tuy nhiên, giống như lớp, struct có thể thiết đặt nhiều giao diện, và làm y chang như với lớp. Sau đây là một đoạn mã nhỏ cho thấy việc một struct thiết đặt một giao diện thế nào. Chương 9, "Giao diện" sẽ đề cập đến giao diện.

```
// Khai báo một giao diện IImage
interface IImage
{  void Paint();
}

// Khai báo một struct có thiết đặt một giao diện
struct Picture: IImage
{
```

```
    public void Paint() // hàm giao diện được thiết đặt
    {
       // đoạn mã thiết đặt hàm Paint() ở đây
    }
    private int x, y, z; // các biến thành viên khác của struct
}
```

Không được có destructor hoặc custom default constructor

Struct không thể có hàm hủy nhưng có thể khai báo những hàm khởi dựng (constructor), nhưng các hàm khởi dựng này phải có thông số. Sẽ là sai lầm khi khai báo một hàm khởi dựng mặc nhiên không thông số. Nếu bạn không cung cấp hàm constructor, thì struct sẽ nhận một hàm constructor mặc nhiên từ trình biên dịch, và hàm này sẽ cho những vùng mục tin kiểu trị về những trị mặc nhiên thích ứng với kiểu dữ liệu, còn những vùng mục tin kiểu qui chiếu thì cho về **null**. Còn nếu bạn muốn cung cấp bất cứ hàm constructor nào có thông số, thì bạn phải khởi gán tất cả các vùng mục tin ngay trong struct. Các thành viên của struct không thể có những initializer (bộ khởi gán). Một hàm constructor mặc nhiên sẽ lo việc này như đã nói trên. Ta có thí dụ con về việc khai báo một struct với một hàm constructor có thông số:

```
struct KeyValuePair
{
    string key;
    string value;
    public KeyValuePair(string key, string value) // hàm constructor
                                                  // khởi gán key & value
  {
       if (key == null || value == null)
          throw new ArgumentException();// nếu key, value null
                                        // thì tung ra biệt lệ
     this.key = key;
     this.value = value;
  }
}
```

Hàm constructor tự tạo bảo vệ việc key và value không được null chỉ khi nào được triệu gọi tường minh. Trong trường hợp biến của KeyValuePair thuộc khởi gán mặc nhiên thì các vùng mục tin key và value sẽ là null, và struct phải chuẩn bị thụ lý trường hợp này.

Khi bạn tạo một đối tượng struct dùng đến tác tử **New**, nó sẽ tạo ra và hàm constructor thích ứng sẽ được gọi vào. Khác với lớp, *đối tượng struct có thể được khởi gán không dùng đến tác tử **New**,* các vùng mục tin nằm yên chưa được gán, và đối tượng struct không thể dùng được cho tới khi tất cả các vùng mục tin được khởi gán.

Không được khởi gán

Bạn không thể khởi gán một instance field trên một struct. Do đó, viết như sau sẽ là bất hợp lệ:

```
private int xVal = 50;
private int yVal = 100;
```

mặc dù điều này bạn có thể thực hiện trên lớp. Struct được thiết kế sao cho đơn giản và "nhẹ cân". Trong khi các biến thành viên private đem lại tính che dấu dữ liệu và gói ghém, thì một số lập trình viên cảm thấy hơi "cường điệu" đối với struct, nên cho biến thành viên thành public, làm cho việc thiết đặt struct nhẹ nhàng hơn. Còn số lập trình viên khác thì lại cho rằng thuộc tính cung cấp một giao diện sạch sẽ và đơn giản hơn, và tập quán lập trình tốt là cho cất giấu thông tin kể cả khi đối tượng thuộc loại nhẹ cân. Bạn chọn một trong hai cách tiếp cận kể trên là tùy triết lý design của bạn, ngôn ngữ sẽ hỗ trợ cả hai cách tiếp cận.

8.2 Tạo những đối tượng struct

Bạn có thể tạo một thể hiện đối tượng struct bằng cách dùng từ chốt **new** trong câu lệnh gán, giống như với lớp. Trong thí dụ 8-1, lớp **Tester** cho tạo một thể hiện của struct **Location** như sau:

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

với **loc1** là thể hiện mới của struct **Location**, và tiếp nhận hai trị số, 200 và 300, làm toạ độ.

8.2.1 Struct thuộc kiểu trị

Định nghĩa lớp **Tester** trong thí dụ 8-1, bao gồm một đối tượng **Location**, **loc1**, được tạo với hai trị số 200 và 300. Dòng lệnh này sẽ triệu gọi hàm constructor **Loaction()**:

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

Sau đó, **WriteLine()** được gọi vào:

```
Console.WriteLine("Loc1 location: {0}", loc1);
```

Console.WriteLine() chờ đợi một đối tượng; nhưng, lẽ dĩ nhiên, **Location** là một struct (một kiểu trị). Trình biên dịch tự động cho "đóng hộp" (boxing) struct (giống như

với bất cứ kiểu dữ liệu trị nào) và là một đối tượng bị "đóng hộp" được trao qua cho **WriteLine(). ToString()** được gọi vào xử lý đối tượng bị "đóng hộp", và vì struct thừa kế ngầm từ **object**, nó có khả năng đáp ứng một cách đa hình, cho phủ quyết **ToString**, giống như bất cứ đối tượng khác có thể làm:

```
Loc1 location: 200, 300
```

Tuy nhiên, struct là đối tượng kiểu trị (được tạo trên stack, thay vì trên heap như với lớp) và khi được trao qua cho các hàm, struct sẽ được trao qua theo trị (nghĩa là một bản "photocopy" trị được trao qua), như ta đã thấy trong câu lệnh kế tiếp, theo đấy đối tượng **loc1** được trao qua cho hàm **myFunc()**:

```
t.myFunct(loc1);
```

Trong **myFunct()**, những trị mới được gán cho **x** và **y**, và những trị mới được in ra:

```
In MyFunc loc: 50, 100
```

Khi bạn trở về **Main()** rồi gọi **WriteLine()**, lần nữa trị không thay đổi:

```
Loc1 location: 200, 300
```

Bạn nhớ cho struct trao qua một đối tượng kiểu trị, và một bản sao được thực hiện trong hàm **myFunc()**. Bạn thử làm cuộc trắc nghiệm, bằng cách thay struct bởi class:

```
public class Location
```

rồi cho chạy lại Tester. Sau đây là kết xuất:

```
Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 50, 100
```

Lần này, đối tượng **Location** mang ý nghĩa qui chiếu. Do đó, khi các trị thay đổi trong **myFunc()**, thì chúng bị thay đổi trên đối tượng hiện hành khi trở về **Main()**.

8.2.2 Triệu gọi hàm constructor mặc nhiên

Như đã nói trước đây, nếu bạn không tự tạo một hàm constructor, thì một hàm constructor mặc nhiên hiểu ngầm sẽ được triệu gọi bởi trình biên dịch. Ta có thể thấy điều này khi cho đóng khung hàm constructor trong cặp dấu /* */ (ta gọi là comment out) như sau:

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

và thay thế câu lệnh thứ nhất trên **Main()** bởi câu lệnh tạo thể hiện đối tượng **Location** không thông số:

```
// Location loc1= new Location(200, 300);
Location loc1= new Location();    // hàm constructor mặc nhiên
```

Vì không có thông số, nên hàm constructor mặc nhiên hiểu ngầm được triệu gọi. Kết xuất sẽ như sau:

```
Loc1 location: 0, 0
In MyFunc loc: 50, 100
Loc1 location: 0, 0
```

Hàm constructor minh họa đã cho khởi gán các biến thành viên về zero, vì kiểu dữ liệu là int.

Bạn để ý: Trên C#, không phải bao giờ từ chốt **new** cũng tạo đối tượng trên vùng ký ức heap. Lớp được tạo trên heap, còn struct thì lại được tạo trên stack. Ngoài ra, khi **new** bị bỏ qua, thì hàm constructor sẽ không bao giờ được gọi vào. Vì C# đòi hỏi gán trị rõ ràng, bạn phải khởi gán một cách tường minh tất cả các biến thành viên trước khi dùng struct.

8.2.3 Tạo đối tượng struct không dùng *new*

Vì **loc1** là một đối tượng struct (chứ không phải class), nên nó được tạo trên stack. Do đó, trên thí dụ 8-1, khi tác tử **new** được gọi vào:

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

thì đối tượng **Location** sẽ được tạo trên ký ức stack.

Tác tử **new** triệu gọi hàm constructor của **Location**. Tuy nhiên, khác với class, ta có khả năng tạo một đối tượng struct mà không dùng đến **new**. Điều này rất là nhất quán khi nhìn đến cách các biến kiểu dữ liệu bẩm sinh (như **int** chẳng hạn) được định nghĩa, và được minh họa bởi thí dụ 8-2 dưới đây.

Thí dụ 8-2: Tạo một struct không dùng new

```
using System

public struct Location
{   // hàm khởi dựng
    public Location(int xCoordinate, 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);
    }

    private int xVal;
    private int yVal;
}

public class Tester
{
    static void Main()
    {
        Location loc1;   // không triệu gọi hàm constructor
        loc1.xVal = 75;
        loc1.yVal = 225;
        Console.WriteLine(loc1);
    }   // end Main
}
```

Trong thí dụ 8-2 trên, bạn khởi gán trực tiếp các biến cục bộ, trước khi triệu gọi một hàm hành sự của **loc1** và trước khi trao đối tượng qua cho **WriteLine:**

```
    loc1.xVal = 75;
    loc1.yVal = 225;
```

Nếu bạn cho "comment out" một trong những câu lệnh gán trên, rồi cho biên dịch lại

```
static void Main()
{
   Location loc1;   // không triệu gọi hàm constructor
   loc1.xVal = 75;
   // loc1.yVal = 225;
   Console.WriteLine(loc1);
}
```

Bạn sẽ nhận thông báo sai lầm khi biên dịch: Use of unassigned local variable 'loc1'. Một khi bạn gán tất cả các trị, bạn có thể truy xuất những trị này thông qua các thuộc tính **x** và **y**:

```
static void Main()
{
   Location loc1;   // không triệu gọi hàm constructor
   loc1.xVal = 75;  // gán biến thành viên
   loc1.yVal = 225; // gán biến thành viên
   loc1.x = 300;    // dùng thuộc tính
   loc1.y = 400;    // dùng thuộc tính
   Console.WriteLine(loc1);
}  // end Main
```

Bạn cẩn thận khi sử dụng thuộc tính. Mặc dù các thuộc tính này cho phép bạn hỗ trợ encapsulation bằng cách làm chúng thành những trị private, bản thân các thuộc tính hiện là những hàm hành sự thành viên, và bạn không thể triệu gọi một hàm thành viên cho tới khi bạn khởi gán tất cả các biến thành viên.

Bạn để ý: Tạo những struct không dùng **new** không đem lại lợi ích nhiều, và có thể tạo khó khăn cho chương trình: đọc khó hiểu, dễ sai, và khó bảo trì. Bạn có thể thử dùng, nhưng ráng mà chịu rủi ro.

8.3 Struct và tính kế thừa

Tất cả các đối tượng struct đều ngầm kế thừa từ lớp **Object**. Trên khai báo một struct bạn có thể khai báo một danh sách những giao diện được thiết đặt, nhưng không thể nào một struct lại khai báo một lớp cơ sở.

Kiểu dữ liệu struct, vì thuộc kiểu trị, nên không bao giờ được trừu tượng, nên được hiểu ngầm là bị "vô sinh", nghĩa là không được dẫn xuất. Do đó, các từ chốt modifier **abstract** và **sealed** không được dùng trên dòng lệnh khai báo struct.

Vì struct không hỗ trợ kế thừa, việc khai báo tầm nhìn thấy được trên biến thành viên struct không thể có các từ chốt **protected** hoặc **protected internal,** vì protected liên quan đến lớp dẫn xuất.

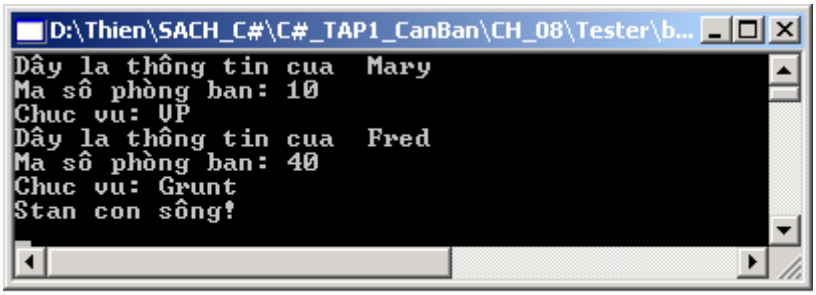
Còn các hàm trả về của một struct không thể là **abstract** hoặc **virtual**, và từ chốt modifier **override** chỉ được phép dùng phủ quyết những hàm hành sự được kế thừa từ kiểu dữ liệu **Object**.

8.4 Một thí dụ để kết thúc chương

```
namespace Prog_CSharp
{
   using System;

   // Khai báo một enum "cây nhà lá vườn"
   enum EmpType: byte  // loại nhân viên
   {  Manager = 10,
      Grunt = 1,
      Contractor = 100,
      VP = 9
   }

   // Khai báo một struct sử dụng enum kể trên
   struct EMPLOYEE
   {  // Các vùng mục tin
      public EmpType title;  // chức vụ, kiểu enum
      public string name;    // tên nhân viên
      public short deptID;       // mã số phòng ban

      // Hàm constructor của struct.
      public EMPLOYEE(EmpType et, string n, short d)
      {  title = et;  // khởi gán
         name = n;    // …
         deptID = d;  // …
      }
   }

   class Tester
   {
      public void DisplayEmpStats(EMPLOYEE e) // hiển thị tình trạng
                                              // nhân viên
      {
         Console.WriteLine("Đây là thông tin của {0}", e.name);
         Console.WriteLine("Mã số phòng ban: {0}", e.deptID);
         Console.WriteLine("Chức vụ: {0}", Enum.Format(typeof(EmpType),
                         e.title, "G"));
      }

      public void UnboxThisEmployee(object o) // rã hộp nhân viên
```

```
        {
            EMPLOYEE temp = (EMPLOYEE)o;
            Console.WriteLine(temp.name + " còn sống!");
        }

    public static int Main(string[] args)
    {   // Tạo và định dạng Fred.
            EMPLOYEE fred;
            fred.deptID = 40;
            fred.name = "Fred";
            fred.title = EmpType.Grunt;

            // Tạo và định dạng Mary dùng đến hàm constructor.
            EMPLOYEE mary = new EMPLOYEE(EmpType.VP, "Mary", 10);

            // Để cho nhân viên kê khai lý lịch
            Tester t = new Tester();
            t.DisplayEmpStats(mary);
            t.DisplayEmpStats(fred);

            // Tạo và đóng hộp một nhân viên
            EMPLOYEE stan = new EMPLOYEE(EmpType.Grunt, "Stan", 10);
            object stanInBox = stan;

            // Chuyển object cho hàm rã hộp
            t.UnboxThisEmployee(stanInBox);

        return 0;
    }   // end Main
    }   // end Tester
}   // end Prog_CSharp
```

Hình 8-1 cho thấy kết xuất của thí dụ trên.



Hình 8-1: Kết xuất thí dụ Tester

Trong thí dụ trên, bạn tạo ra một struct mang tên EMPLOYEE (nhân viên) trên stack và thao tác trên mỗi vùng mục tin sử dụng tác tử chấm (dot operator). Bạn tạo một hàm constructor "cây nhà lá vườn". Bạn nhớ cho là bạn không thể định nghĩa một hàm constructor mặc nhiên. Từ hàm constructor tự tạo, bạn có thể tạo một nhân viên mới như sau:

```
// Phải dùng "new" để triệu gọi hàm constructor."cây nhà lá vườn"
EMPLOYEE mary = new EMPLOYEE(EmpType.VP, "Mary", 10);
```

Struct cũng có thể dùng làm thông số đối với bất cứ hàm thành viên nào. Thí dụ hàm DisplayEmpStats() cho hiển thị tình trạng nhân viên:

```
public void DisplayEmpStats(EMPLOYEE e) // hiển thị tình trạng
                                        // nhân viên
{   Console.WriteLine("Đây là thông tin của: {0}", e.name);
    Console.WriteLine("Mã số phòng ban: {0}", e.deptID);
    Console.WriteLine("Chức vụ: {0}", Enum.Format(typeof(EmpType),
                    e.title, "G"));
}
```

và cách sử dụng hàm này trong Main()

```
public static int Main(string[] args)
{
       ...
       // Để cho nhân viên kê khai lý lịch
       Tester t = new Tester();
       t.DisplayEmpStats(mary);
       t.DisplayEmpStats(fred);
       ...
   return 0;
}  // end Main
```

Chương 9

Giao diện

Một ***giao diện*** (interface) được xem như là một "khế ước" hoặc "hợp đồng" (contract) bảo đảm với người sử dụng rằng một lớp hoặc một struct sẽ hoạt động tốt, theo hoạch định. Khi một lớp nào đó thiết đặt thi công một giao diện, nó bảo với bất cứ khách hàng nào rằng "Tôi bảo đảm sẽ hỗ trợ những hàm hành sự, những thuộc tính, những tình huống (event) và những indexer (bộ chỉ muc) của giao diện mang tên này".

Giao diện được sử dụng như là một phương án thay thế một lớp trừu tượng, tạo những "khế ước" giữa người sử dụng lớp và người tạo ra lớp. Những "khế ước" được thực hiện dùng đến từ chốt **interface**, khai báo một kiểu dữ liệu qui chiếu gói gọn lại trong "khế ước".

Về mặt cú pháp, giao diện cũng giống như một lớp nhưng *chỉ gồm toàn những hàm hành sự trừu tượng*. Một lớp trừu tượng thường được dùng làm lớp cơ sở cho một dòng họ lớp dẫn xuất, trong khi giao diện được dùng vào mục đích trộn lẫn với những "cây kế thừa" (inheritance tree) khác

Khi một lớp thiết đặt (implement) một giao diện, nó phải thi công tất cả các hàm hành sự của giao diện này; nghĩa là lớp bảo rằng "Tôi thỏa thuận tuân thủ mọi điều khoản hợp đồng đã được định nghĩa bởi giao diện này". Vì là một khế ước nên hai bên phải tuân thủ: người thiết kế giao diện không được thay đổi giao diện, còn người thi công phải làm đúng những gì đã định nghĩa trong giao diện. (Nó giống như bên xây dựng)

Khi kế thừa từ một lớp trừu tượng bạn thực sự thiết đặt một mối quan hệ ***is-a*** (là-một) mà ta đã đề cập qua ở chương 5, "Lớp và Đối tượng". Trong khi ấy, thiết đặt một giao diện, bạn lại định nghĩa một mối quan hệ loại khác chưa từng thấy từ trước đến nay: mối *quan hệ thiết đặt* (implement relationship). Hai mối quan hệ "là-một" và "thiết đặt" khác biệt một cách tế nhị. Một chiếc xe *là một* loại xe, nhưng nó có thể *thiết đặt* khả năng **CanBeBoughtWithABigLoan** (có thể mua theo kiểu trả góp, giống như mua nhà vậy).

Trong chương này, bạn sẽ học cách tạo một giao diện, thiết đặt nó thế nào, và cách sử dụng giao diện trong chương trình. Ngoài ra, bạn sẽ học cách thiết đặt cùng lúc nhiều giao diện, làm sao phối hợp các giao diện với nhau cũng như cách nới rộng giao diện, kể cả việc trắc nghiệm xem một lớp có thiết đặt giao diện hay không.

9.1 Thiết đặt một giao diện thế nào?

Cú pháp định nghĩa một giao diện mới như sau:

[*attributes*] [*access-modifier*] **interface** *interface-name* [**:***base-list*] {***interface body***}

Những mục nằm trong cặp dấu ngoặc vuông [ ] là tùy chọn, không bắt buộc. Tạm thời bạn khoan quan tâm đến *attributes*. Chúng tôi sẽ đề cập sau trong chương 4, "Tìm hiểu biểu mẫu", tập 3 của bộ sách này.

Mục *access-modifier*, bao gồm các từ chốt **public, private, protected** và **protected internal** đã được đề cập ở chương 5.

Từ chốt **interface** được theo sau là tên giao diện, *interface-name*. Theo sử dụng thông thường (nhưng không bắt buộc) tên giao diện sẽ bắt đầu bởi chữ I hoa. Thí dụ, IStorable, ICloneable, ILongBong, ILinhTinh, v.v.

Danh sách các giao diện mà giao diện này nới rộng (như sẽ được mô tả ở mục "Thiết đặt nhiều hơn một giao diện" trong chương này) sẽ được liệt kê trên ***base-list*** sau dấu hai chấm, mỗi giao diện được phân cách bởi dấu phẩy. Thí dụ sau đây cho thấy giao diện **IListCounter** được nới rộng bởi base-list gồm hai giao diện **IList** và **ICounter**.

```
interface IListCounter: IList, ICounter {}
```

Cuối cùng ***interface body*** là phần thân của giao diện, đóng khung trong hai dấu {}, khai báo việc thiết đặt của giao diện, mà chúng tôi sẽ mô tả dưới đây. Phần thân này gồm một số thành viên. Các thành viên này (bao gồm methods, properties, events, hoặc indexers) có thể được kế thừa từ base interface, hoặc là những thành viên được khai báo ngay bởi giao diện. Một interface không thể chứa hằng, vùng mục tin, tác tử, hàm khởi dựng thể hiện, hàm hủy, hoặc kiểu dữ liệu, cũng như không thể chứa thành viên static bất cứ kiểu nào. Ngoài ra, thành viên giao diện *không được bao gồm* các từ chốt access modifier sau đây: **`abstract, public, protected, internal, private, virtual, override`**, hoặc **`static`**. Lớp chịu thiết đặt giao diện sẽ đặt để phạm vi nhìn thấy được (visibility) của các thành viên trong giao diện.

Thí dụ sau đây cho thấy một giao diện **IStringList** với đủ thứ thành viên:

```
public delegate void StringListEvent(IStringList sender); // event
public interface IStringList
{
    void Add(string s);              // hàm hành sự
    int Count { get; }               // thuộc tính
```

```
    event StringListEvent Changed;       // event
    string this[int index] { get; set; } // indexer
}
```

Giả sử, bạn mong muốn tạo một giao diện mô tả những hàm hành sự và thuộc tính mà một lớp cần được trữ hoặc tìm lại từ một căn cứ dữ liệu hoặc từ nguồn lực trữ khác, như từ một tập tin chẳng hạn. Bạn quyết định gọi giao diện này là **IStorable**. Và trong giao diện này có thể bạn khai báo hai hàm hành sự trong phần thân: **Read()** và **Write():**

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

Read() và **Write()** được gọi là thành viên giao diện (interface member). *Mục đích của một giao diện là định nghĩa những khả năng mà bạn muốn một lớp phải có sẵn.*

Thí dụ, có thể bạn muốn tạo một lớp public mang tên **Document**. Xem ra kiểu dữ liệu **Document** này có thể đem trữ lên một căn cứ dữ liệu, do đó bạn quyết định thiết đặt giao diện **IStorable** trên **Document**. Giao diện theo sau hai dấu chấm. Do đó, bạn viết:

```
public class Document: IStorable
{
    public void Read() {…}
    public void Write(object obj) {…}
}
```

Tới đây, xem như trách nhiệm của bạn, tác giả lớp **Document**, là phải cung cấp việc thi công đầy ý nghĩa của các hàm hành sự của giao diện **IStorable**. Lớp **Document** phải thi công tất cả các hàm khớp với mỗi hàm trên giao diện **IStorable**. Do đó, nếu bạn không cung cấp việc thi công đối với các hàm hành sự **Read()** và **Write()** của giao diện **IStorable,** thì trình biên dịch sẽ la làng cho mà coi. Thí dụ 9-1 cho thấy lớp **Document** thiết đặt giao diện **IStorable** thế nào.

Thí dụ 9-1: Sử dụng một giao diện đơn giản

```
using System;

// khai báo một giao diện
interface IStorable
{
    // không có access modifier, các thành viên là public,
    // không có thiết đặt thi công
    void Read();
    void Write(object obj);
    int Status {get; set;} // thuộc tính Status cho biết trạng thái
```

```
}

// Tạo một lớp Document lo thiết đặt giao diện IStorable
public class Document:IStorable
{
   public Document(string s)     // hàm constructor
   {
      Console.WriteLine("Tạo document với: {0}", s);
   }

   // thi công hàm hành sự Read()
   public void Read()
   {
      Console.WriteLine("Thi công Read() dành cho IStorable");
   }

   // thi công hàm hành sự Write()
   public void Write(object o)
   {
      Console.WriteLine("Thi công Write() dành cho IStorable");
   }

   // thi công thuộc tính Status
   public int Status
   {  get
      {  return status;
      }
      set
      {  status = value;
      }
   }

   // biến dành trữ trị thuộc tính status
   private int status = 0;
}  // end Document

public class Tester
{
   static void Main()
   {
      // truy xuất các hàm hành sự trên đối tượng Document
      Document doc = new Document("Test Document");
      doc.Status = -1;
      doc.Read();
      Console.WriteLine("Document Status: {0}", doc.Status);

      // ép về kiểu giao diện và sử dụng giao diện
      IStorable isDoc = (IStorable) doc;   // ép thể hiện Document về
                                           // kiểu giao diện
      isDoc.Status = 0;
      isDoc.Read();
      Console.WriteLine("IStorable Status: {0}", isDoc.Status);
```

```
    }   // end Main
}   // end Tester
```

Kết xuất
Tạo document với: Test Document
Thi công Read() dành cho IStorable
Document Status: -1
Thi công Read() dành cho IStorable
IStorable Status: 0

Thí dụ trên định nghĩa một giao diện đơn giản, **IStorable**, với hai hàm hành sự **Read()** và **Write()** và một thuộc tính **Status** kiểu int. Bạn để ý, khai báo thuộc tính **Status** không cung cấp cho ta việc thi công đối với **get()** và **set()**, mà chỉ cho biết là có một **get()** và một **set():**

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

Bạn để ý các hàm hành sự của giao diện **IStorable** không được bao gồm những access modifier (nghĩa là **public, protected, internal, private)**. Thực thế, nếu bạn ghi một access modifier, trình biên dịch sẽ phát ra báo sai lầm. Ta phải hiểu ngầm các hàm giao diện là **public** vì đây là một khế ước mà các lớp khác sẽ dùng đến. Bạn không thể tạo một thể hiện giao diện (vì ở đây giao diện giống như một lớp trừu tượng), *mà chỉ có thể tạo một thể hiện trên một lớp có thiết đặt giao diện.*

Lớp thi công giao diện phải tuân thủ khế ước một cách chính xác và trọn vẹn. **Document** phải cung cấp cả hai hàm hành sự **Read(), Write()** và thuộc tính **Status**. Tuy nhiên, việc tuân thủ thế nào là thuộc quyền của lớp **Document**. Mặc dầu **IStorable** yêu cầu **Document** phải có một thuộc tính **Status**, nhưng nó không cần biết hoặc quan tâm liệu xem **Status** hiện là một biến thành viên, hoặc sẽ được lấy từ một căn cứ dữ liệu. Chi tiết là do phía lớp thi công lo lấy.

Bạn để ý, để cho đơn giản việc thi công các hàm hành sự cũng như thuộc tính, chúng tôi chỉ dùng câu lệnh **Console.WriteLine()** cho biết là đã chui vào thi hành các hàm và thuộc tính này. Trong thực tế, việc thi công phức tạp hơn nhiều. Thông thường, bạn tạo khung sườn như theo thí dụ của chúng tôi để xem tổng thể có chạy như theo ý muốn hay không. Sau đó, bạn thi công thực thụ từng hàm một, thay thế dòng lệnh **Console. WriteLine()** tương ứng rồi lần lượt trắc nghiệm các hàm của giao diện.

9.1.1 Thiết đặt cùng lúc nhiều giao diện

Trong một lúc, lớp có thể thiết đặt nhiều giao diện hơn chỉ là một. Thí dụ, nếu lớp **Document** của ta có thể được trữ và còn có thể được trữ ở dạng dồn nén (compressed), thì có thể bạn chọn thi công cả hai giao diện **IStorable** và **ICompressible** (lo nén dữ liệu). Muốn thế, bạn cho thay đổi việc khai báo (trên phần base-list của lớp) cho biết cả hai giao diện kể trên được thiết đặt, phân cách nhau bởi dấu phẩy:

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

Như vậy, lớp **Document** cũng phải thiết đặt những hàm hành sự được khai báo trong giao diện **ICompressible**, đó là **Compress()** và **Decompress()**:

```
public void Compress()
{
   Console.WriteLine("Thi công hàm hành sự Compress()");
}

public void Decompress()
{
   Console.WriteLine("Thi công hàm hành sự Decompress()");
}
```

Nếu ta cho chạy lại thí dụ được thay đổi, ta thấy đối tượng **Document** có thể dùng đến các hàm **Compress(), Decompress():**

```
Tạo document với: Test Document
Thi công Read() dành cho IStorable
Thi công hàm hành sự Compress()
```

9.1.2 Nới rộng các giao diện

Ta có thể nới rộng một giao diện hiện hữu bằng cách bổ sung hàm hành sự và thuộc tính mới, hoặc thay đổi cách hoạt động của thành viên hiện hữu. Thí dụ, có thể bạn muốn nới rộng chức năng của **ICompressible** với một giao diện mới, **ILoggedCompressible**, lo việc nới rộng giao diện nguyên thủy, với hàm hành sự, **LogSavedBytes** lo theo dõi những bytes tiết kiệm được:

```
interface ILoggedCompressible: ICompressible
{
   void LogSavedBytes();  // theo dõi bytes tiết kiệm được
}
```

Như vậy, tha hồ các lớp tự do lựa chọn thi công hoặc **ICompressible** hoặc **ILoggedCompressible**. Nếu một lớp thiết đặt **ILoggedCompressible**, thì nó phải thiết đặt tất cả các hàm hành sự của cả hai giao diện **ILoggedCompressible** và

ICompressible. Các đối tượng kiểu dữ liệu này có thể được "ép kiểu" (casting) hoặc về **ILoggedCompressible** hoặc về **ICompressible.**

9.1.3 Phối hợp nhiều giao diện với nhau

Cũng tương tự như thế, bạn có thể tạo những giao diện mới bằng cách phối hợp nhiều giao diện hiện có, và nếu thấy cần thiết, có thể bổ sung thêm những hàm hành sự và thuộc tính mới riêng của mình. Nghĩa là, một giao diện có thể kế thừa từ zero hoặc nhiều giao diện (được gọi là explicit base interface). Trong trường hợp này, theo sau tên giao diện là dấu hai chấm kèm theo danh sách các base interface phân cách bởi dấu phẩy.

Thí dụ, bạn quyết định tạo một giao diện **IStorableCompressible**, vừa trữ vừa nén; nghĩa là explicit base interface gồm **IStorable** và **ILoggedCompressible**. Giao diện **IStorableCompressible** này phối hợp các hàm hành sự của hai giao diện **IStorable** và **ILoggedCompressible**, nhưng cũng bổ sung một hàm hành sự mới, **LogOriginalSize()**, để trữ kích thước nguyên thủy của mục tin trước khi bị nén:

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

Thí dụ 9-2 sau đây minh hoạ việc sử dụng các giao diện nới rộng và phối hợp

Thí dụ 9-2: Nới rộng và phối hợp các giao diện

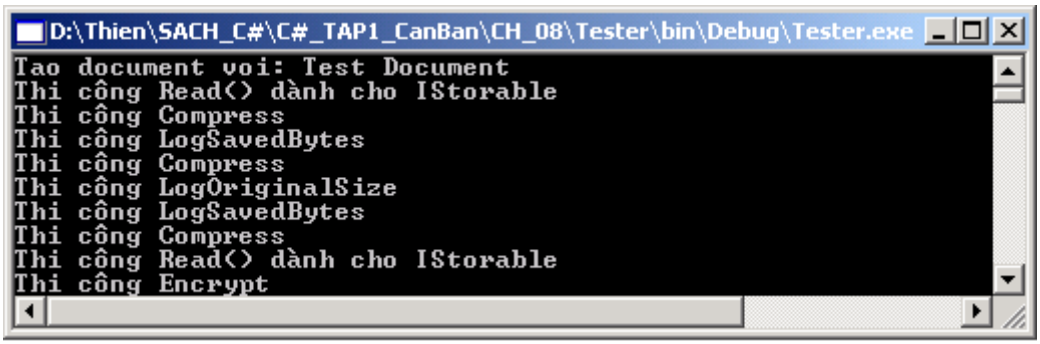
```
using System;

// Một giao diện
interface IStorable
{
   void Read();
   void Write(object obj);
   int Status {get; set;}
}

// Lại thêm một giao diện mới
interface ICompressible
{
   void Compress();
   void Decompress();
}

// Nới rộng giao diện
```

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

// Phối hợp các giao diện với nhau tạo thành một giao diện khác
interface IStorableCompressible: IStorable, ILoggedCompressible
{
   void LogOriginalSize();
}

// Và đây thêm một giao diện khác lo mã hoá và giải mã
interface IEncryptable
{
   void Encrypt();  // mã hóa dữ liệu
   void Decrypt();  // giải mã dữ liệu
}

// Tạo lớp Document cho thiết đặt các giao diện
//  IStorableCompressible và IEncryptable
public class Document: IStorableCompressible, IEncryptable
{
   public Document(string s)     // hàm constructor
   {
      Console.WriteLine("Tạo document với: {0}", s);
   }

   // Thi công hàm hành sự Read()
   public void Read()
   {
      Console.WriteLine("Thi công Read() dành cho IStorable");
   }

   // Thi công hàm hành sự Write()
   public void Write(object o)
   {
      Console.WriteLine("Thi công Write() dành cho IStorable");
   }

   // Thi công thuộc tính Status
   public int Status
   {   get{ return status; }
      set{ status = value; }
   }

   // Thi công các hàm thuộc giao diện ICompressible
   public void Compress()
   {   Console.WriteLine("Thi công Compress");
   }
   public void Decompress()
   {   Console.WriteLine("Thi công Decompress");
   }

   // Thi công hàm bổ sung của giao diện ILoggedCompressible
```

```
    public void LogSavedBytes()
    {   Console.WriteLine("Thi công LogSavedBytes");
    }

    // Thi công hàm bổ sung của giao diện IStorableCompressible
    public void LogOriginalSize()
    {   Console.WriteLine("Thi công LogOriginalSize");
    }

    // Thi công các hàm của giao diện IEncryptable
    public void Encrypt()
    {   Console.WriteLine("Thi công Encrypt");
    }
    public void Decrypt()
    {   Console.WriteLine("Thi công Decrypt");
    }

    // Biến trữ trị cho thuộc tính status của IStorable
    private int status = 0;
}   // end Document

public class Tester
{
    static void Main()
    {   // Tạo một đối tượng Document
        Document doc = new Document("Test Document");

        // ép kiểu đối tượng Document qui chiếu về các giao diện khác nhau
        IStorable isDoc = doc as IStorable;
        if (isDoc != null)
        {   isDoc.Read();
        }
        else
            Console.WriteLine("IStorable không được hỗ trợ");

        ICompressible icDoc = doc as ICompressible;
        if (icDoc != null)
        {   icDoc.Compress();
        }
        else
            Console.WriteLine("ICompressible không được hỗ trợ");

        ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
        if (ilcDoc != null)
        {   ilcDoc.LogSavedBytes();
            ilcDoc.Compress();
            // ilcDoc.Read(); hàm này thuộc IStorable không dính dáng
            // với giao diện ILoggedCompressible, nếu cho chạy sẽ sai
        }
        else
            Console.WriteLine("ILoggedCompressible không được hỗ trợ");

        IStorableCompressible iscDoc = doc as IStorableCompressible;
        if (iscDoc != null)
```

```
        {   iscDoc.LogOriginalSize();      // IStorableCompressible
            iscDoc.LogSavedBytes();        // ILoggedCompressible
            iscDoc.Compress();             // ICompressible
            iscDoc.Read();                 // IStorable
        }
        else
            Console.WriteLine("IStorableCompressible không được hỗ trợ");

        IEncryptable ieDoc = doc as IEncryptable;
        if (ieDoc != null)
        {   ieDoc.Encrypt();
        }
        else
            Console.WriteLine("IEncryptable không được hỗ trợ");
    }   // end Main
}   // end Tester
```

Hình 9-1 cho thấy kết xuất:



Chắc bạn đọc và hiểu rõ thí dụ làm gì. Chỉ lưu ý bạn, chúng tôi có cho "comment out" (nghĩa là dùng hai dấu // để biến một câu lệnh thành chú thích) dòng lệnh sau đây:

```
// ilcDoc.Read();
```

hàm **Read()** này thuộc giao diện **IStorable** không dính dáng với giao diện **ILogged Compressible**, nếu cho biên dịch sẽ phát ra thông điệp sai lầm.

Nếu bạn cho ép kiểu đối tượng **Document** về **IStorableCompressible** (phối hợp giao diện nới rộng với giao diện **IStorable**) thì lúc này bạn có thể triệu gọi các hàm khác nhau của **IStorableCompressible**, **ILoggedCompressible, ICompressible** và **IStorable**.

```
IStorableCompressible iscDoc = doc as IStorableCompressible;
    if (iscDoc != null)
    {   iscDoc.LogOriginalSize();      // IStorableCompressible
        iscDoc.LogSavedBytes():        // ILoggedCompressible
        iscDoc.Compress();             // ICompressible
```

```
            iscDoc.Read();                  // IStorable
        }
```

9.1.4 Thuộc tính giao diện

Bạn có thể khai báo thuộc tính trên giao diện, theo cú pháp sau đây:

[*attributes*] [**new**] *type identifier* **{***interface-accessors***}**

theo đấy *attributes* và *type* được định nghĩa giống như với các lớp; *identifier* là tên thuộc tính, còn **{***interface-accessors***}** là thân các hàm accessor của thuộc tính giao diện, mỗi hàm chỉ gồm dấu chấm phây (;). Mục đích của hàm accessor chỉ là cho biết thuộc tính thuộc loại read-write (thân có get; và set;), read-only (thân chỉ có get;), hoặc write-only (thân chỉ có set;). Còn từ chốt **new** dùng khai báo rõ ra là cho ẩn mình một thành viên thuộc tính được kế thừa từ một lớp cơ sở, thường được khai báo trong lớp được dẫn xuất cũng dùng cùng tên với từ chốt **new** nằm ở đầu tên thành viên (xem 9.4.2).

Thí dụ sau đây cho thấy một interface indexer accessor:

```
public interface IMyInterface
{
 // khai báo thuộc tính Name của giao diện:
 string Name
 {
  get;   // đọc
  set;   // viết
 }
}
```

Trong thí dụ 9-3 sau đây, giao diện **`IEmployee`** gồm có một thuộc tính read-write, **`Name`**, và một thuộc tính read-only, **`Counter`**. Lớp **`Employee`** thiết đặt giao diện **`IEmployee`** và sử dụng hai thuộc tính kể trên. Chương trình sẽ đọc tên nhân viên mới và tổng số nhân viên hiện hành rồi cho hiển thị tên nhân viên và số lượng nhân viên được tính ra.

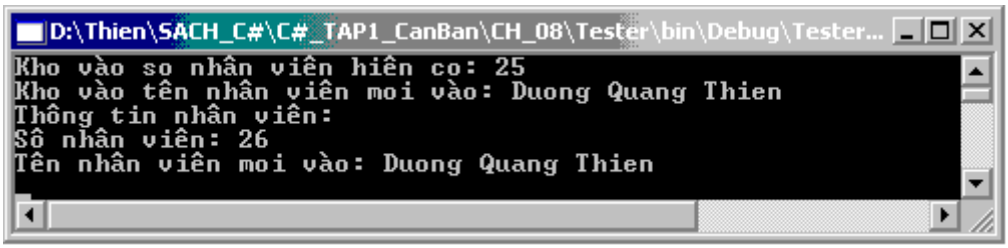
Thí dụ 9-3: Lớp Employee và giao diện IEmployee

```
using System;
// Các thuộc tính giao diện
interface IEmployee
{
    string Name // thuộc tính Name trữ tên nhân viên; đọc và viết
    { get; set;
    }
```

```
    int Counter // thuộc tính Counter đếm số nhân viên, read-only
    { get;
     }
}

public class Employee: IEmployee
{
    public static int numberOfEmployees; // tổng số nhân viên
    private int counter;
    private string name;

    // Read-write instance property:
    public string Name
    {
      get
      {
          return name;
       }
       set
       {
          name = value;
       }
    }

    // Read-only instance property:
    public int Counter
    {
       get
       {
          return counter;
       }
    }

    // Hàm constructor:
    public Employee()
    {
       counter = ++counter + numberOfEmployees;
    }
}

public class MainClass
{
    public static void Main()
    {
       Console.Write("Khỏ vào số nhân viên hiện có: ");
       string s = Console.ReadLine();
       Employee.numberOfEmployees = int.Parse(s);
       Employee e1 = new Employee();
       Console.Write("Khỏ vào tên nhân viên mới vào: ");
       e1.Name = Console.ReadLine();
       Console.WriteLine("Thông tin nhân viên:");
       Console.WriteLine("Số nhân viên: {0}", e1.Counter);
       Console.WriteLine("Tên nhân viên mới vào: {0}", e1.Name);
```

```
    }
}
```

Hình sau đây cho thấy kết xuất thí dụ trên:



9.2 Truy xuất các hàm hành sự giao diện

Trong thí dụ 9-2, bạn có thể truy xuất các thành viên giao diện **IStorable** giống như là thành viên của lớp **Document**:

```
Document doc = new Document("Test Document");
doc.Status = -1;    // biến thành viên của giao diện IStorable
doc.Read();  // hàm thành viên của giao diện IStorable
```

hoặc bạn có thể tạo một thể hiện của giao diện bằng cách ép (casting) đối tượng document về kiểu giao diện **IStorable** chẳng hạn:

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

Trong trường hợp này, trên Main() bạn biết "chắc cú" là **Document** là một **IStorable**, do đó bạn lợi dụng sự hiểu biết này.

Bạn để ý: Như đã nói, bạn *không thể cho thể hiện trực tiếp một giao diện*, (vì chẳng qua giao diện chỉ là một loại lớp trừu tượng "không có da có thịt"), nghĩa là ta không thể viết như sau:

```
IStorable isDoc = new IStorable();   // trật lấc rồi !!!
```

Tuy nhiên, bạn có thể tạo một thể hiện của lớp thi công giao diện này, ở đây là **Document**, như theo câu lệnh sau đây:

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

rồi sau đó, bạn mới có thể tạo một *thể hiện giao diện* (instance of interface) bằng cách cho ép kiểu (casting) đối tượng thi công (ở đây là doc) về interface ***type*** (ở đây là **IStorable**) như sau:

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

bạn có thể phối hợp 2 bước kể trên thành:

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

Nói chung, tốt hơn là truy xuất các hàm hành sự giao diện thông qua một *giao diện qui chiếu* (interface reference). Thí dụ lệnh sau đây:

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

cho thấy **isDoc** là một giao diện qui chiếu (interface reference). Nghĩa là, tốt hơn nên dùng **isDoc.Read()** thay vì **doc.Read()**. Truy xuất thông qua một giao diện cho phép bạn xử lý giao diện một cách đa hình (polymorphically) hơn. Nói cách khác, bạn có thể có một hoặc nhiều lớp "xịn" thi công giao diện, rồi sau đó bằng cách truy xuất các lớp này chỉ thông qua giao diện, bạn có thể phớt lờ kiểu dữ liệu thực thụ vào lúc chạy của các lớp này, và xem như thay thế lẫn nhau. Bạn xem lại chương 6 về tính đa hình của lớp.

9.2.1 Cho ép kiểu về một giao diện

Trong nhiều trường hợp, bạn không biết trước một đối tượng sẽ hỗ trợ một giao diện nào. Thí dụ, giả dụ bạn có một tập hợp (collection) những đối tượng **Documents**, một số có thể được trữ, một số lại không. Giả sử, bạn lại thêm một giao diện thứ hai, **ICom pressible**, đối với những đối tượng có thể tự mình nén dữ liệu để chuyển nhanh qua đường email:

```
interface ICompressible
{
   void Compress();     // nén
   void Decompress();   // giải nén
}
```

Ta thử lấy một kiểu dữ liệu **Document**, bạn không thể biết liệu xem nó hỗ trợ **IStorable** hay là **ICompressible** hay cả hai. Bạn chỉ có thể ép về các giao diện:

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

ICompressible icDoc = (ICompressible) doc;
icDoc.Compress();
```

Nếu xem ra **Document** chỉ có thiết đặt giao diện **IStorable** mà thôi:

```
public class Document: IStorable
```

thì việc ép kiểu về giao diện **ICompressible** vẫn được biên dịch vì giao diện này hợp pháp. Tuy nhiên, vì việc "ép duyên" không đúng cách, cho nên khi chương trình chạy, một biệt lệ (exception) sẽ bị tung ra:

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

Chúng tôi sẽ đề cập sau về biệt lệ, ở chương 12, "Thụ lý các biệt lệ**"**

9.2.2 Tác tử is

Chúng tôi đã đề cập tác tử này ở chương 4, mục 4.6.7.1, đề nghị bạn xem lại. Bạn muốn có khả năng hỏi xem đối tượng có hỗ trợ giao diện hay không, để có thể triệu gọi hàm hành sự thích ứng. Trên C#, có hai cách để thực hiện điều này. Cách thứ nhất là dùng tác tử **is**, cú pháp sau đây:

expression **is** *type*

Tác tử **is** sẽ định trị cho ra **true** nếu *biểu thức* (*expression*, phải là một kiểu dữ liệu qui chiếu) có thể được ép an toàn về *type* mà không cho tung ra một biệt lệ. Thí dụ 9-4 minh hoạ việc sử dụng tác tử **is** để trắc nghiệm xem **Documents** có thiết đặt các giao diện **IStorable** và **ICompressible** hay không.

Thí dụ 9-4: Sử dụng tác tử is

```
using System;

interface IStorable
{   void Read();
    void Write(object obj);
    int Status {get; set;}
}

// Thêm một giao diện mới
```

```
interface ICompressible
{   void Compress();
    void Decompress();
}

// Tạo một lớp Document cho thiết đặt interface IStorable
public class Document: IStorable
{
    public Document(string s)     // hàm constructor
    {   Console.WriteLine("Tạo document với: {0}", s);
    }

    // Thi công hàm hành sự Read(), Write, và thuộc tính Status
    public void Read()
    {   Console.WriteLine("Thi công Read() dành cho IStorable");
    }

    public void Write(object o)
    {   Console.WriteLine("Thi công Write() dành cho IStorable");
    }

    public int Status
    {   get
        {   return status;
        }
        set
        {   status = value;
        }
    }

    // Biến dùng trữ trị cho thuộc tính status của IStorable
    private int status = 0;
}   // end Document

public class Tester
{
    static void Main()
    {   // Truy xuất các hàm hành sự trên đối tượng Document
        Document doc = new Document("Test Document");

        // chỉ ép kiểu nếu nó an toàn
        if (doc is IStorable)
        {   IStorable isDoc = (IStorable) doc;
            isDoc.Read();
        }

        if (doc is ICompressible)
        {   ICompressible icDoc = (ICompressible) doc;
            icDoc.Compress();
        }
    }   // end Main
}   // end Tester
```

Thí dụ 9-4 giống như thí dụ 9-2, ngoại trừ nó thêm khai báo đối với giao diện **ICompressible**. Giờ đây **Main()** xác định liệu xem việc ép kiểu có hợp lệ hay không (còn gọi là an toàn, safe, hay không) bằng cách khảo điều khoản **if** sau đây:

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

Như vậy có vẻ "sạch sẽ" hơn và mang tính sưu liệu (documenting) cao. Câu lệnh **if** cho bạn biết việc "ép duyên" chỉ có thể xảy ra nếu đối tượng mang đúng loại giao diện. Tuy nhiên, việc sử dụng tác tử **is** xem ra không hiệu quả. Nếu bạn muốn biết, thì phải phân tích đoạn mã của bản kết sinh MISL, sử dụng trình tiện ích ILDASM.EXE. Sau đây là một trích đoạn (bạn để ý số hàng lệnh được ghi theo hexa).

```
IL_0016: ldloc.0
IL_0017: isinst   Tester.ICompressible
IL_001c: brfalse.s IL_002b
IL_001e: ldloc.0
IL_001f: castclass Tester.ICompressible
IL_0024: stloc.2
IL_0025: ldloc.2
IL_0026: callvirt  instance void Tester.ICompressible::Compress()
```

Ở đây, điều quan trọng là trắc nghiệm đối với **ICompressible** diễn ra ở hàng 17. Từ isinst là mã MISL đối với tác tử is. Nó trắc nghiệm liệu xem đối tượng (doc) có thật là đúng kiểu dữ liệu hay không. Vượt qua trắc nghiệm này, chúng ta xuống hàng `1f` theo đấy `castclass` được triệu gọi. Rất tiếc là `castclass` lại trắc nghiệm kiểu dữ liệu của đối tượng. Như vậy, việc trắc nghiệm được thực hiện hai lần. Một giải pháp hữu hiệu hơn là sử dụng tác tử **as**.

9.2.3 Tác tử as

Tác tử **as** phối hợp tác tử **is** với việc ép kiểu bằng cách trước tiên trắc nghiệm xem việc ép kiểu có hợp lệ hay không (nghĩa là trắc nghiệm xem **is** có cho ra true hay không) rồi sau đó sẽ cho ép kiểu nếu là true. Còn nếu ép kiểu không hợp lệ (nghĩa là trắc nghiệm **is** cho ra false), thì tác tử **as** sẽ trả về **null** (từ chốt **null**, tượng trưng cho null reference, nghĩa là qui chiếu vào "hư vô").

Khi sử dụng **as**, ta khỏi cần xử lý các trường hợp biệt lệ về cast. Vì lý do trên, ta tối ưu hoá việc ép giao diện bằng cách dùng tác tử **as**, mang cú pháp sau đây:

expression **as** *type*

Đoạn mã sau đây cho sửa lại phần trắc nghiệm trên thí dụ 9-4 dùng đến tác tử **as** đồng thời trắc nghiệm null reference:

```
static void Main()
{
    // Truy xuất các hàm hành sự trên đối tượng Document
    Document doc = new Document("Test Document");

    // Cast document về các giao diện khác nhau IStorable &
    // ICompressible
    IStorable isDoc = doc as IStorable;
    if (isDoc != null)
        isDoc.Read();
    else
        Console.WriteLine("IStorable không được hỗ trợ");

    ICompressible icDoc = doc as ICompressible;
    if (icDoc != null)
        icDoc.Compress();
    else
        Console.WriteLine("ICompressible không được hỗ trợ");
```

Hình sau đây cho thấy kết xuất của thí dụ vừa sửa đổi:

Nếu bạn nhìn vào đoạn mã được kết sinh MSIL, bạn sẽ thấy phiên bản này hữu hiệu hơn.

```
 IL_0022: ldloc.0
IL_0023: isinst   Tester.ICompressible
IL_0028: stloc.2
IL_0029: ldloc.2
IL_002a: brfalse.s IL_0034

IL_002c: ldloc.2
IL_002d: callvirt  instance void Tester.ICompressible::Compress()
IL_0032: br.s    IL_003e
```

9.2.4 Tác tử *is* so với tác tử *as*

Nếu ý đồ của bạn là trắc nghiệm xem đối tượng có mang kiểu dữ liệu bạn cần hay không và nếu có bạn sẽ cho ép ngay lập tức, thì lúc này sử dụng tác tử **as** hiệu quả hơn. Tuy nhiên, có lúc bạn chỉ muốn trắc nghiệm kiểu dữ liệu của một tác tử, nhưng lại không

muốn cast ngay lập tức. Mà cũng có thể bạn muốn trắc nghiệm nhưng lại không cast. Trong trường hợp này, tác tử **is** là thích hợp hơn.

9.2.5 Giao diện so với lớp trừu tượng

Giống như một lớp không trừu tượng, một lớp trừu tượng cũng có thể thiết đặt giao diện, và do đó phải thi công tất cả các thành viên của giao diện được liệt kê trên base-list của lớp. Tuy nhiên, một lớp trừu tượng được phép ánh xạ những hàm hành sự giao diện thành những hàm hành sự trừu tượng. Thí dụ

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

abstract class SomeAbstClass: IStorable
{
   public abstract void Read();
   public abstract void Write(object);
}
```

Như bạn thấy đấy, việc thiết đặt giao diện **`IStorable`** trên **SomeAbstClass** cho ánh xạ các hàm **`Read()`** và **Write()** thành những hàm hành sự trừu tượng, và các hàm này sẽ bị phủ quyết trên những lớp không trừu tượng được dẫn xuất từ **SomeAbstClass**. Xem mục 9.3 về vấn đề này.

Nhưng khi các thành viên giao diện được thiết đặt một cách tường minh (explicit implementation, xem mục 9.4) thì lại không thể là trừu tượng; tuy nhiên, chúng có thể được phép triệu gọi những hàm hành sự trừu tượng. Thí dụ:

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

abstract class SomeAbstClass: IStorable
{
   void IStorable.Read() { FF(); }  // khai báo tường minh
   void IStorable.Write() { GG(); } // khai báo tường minh
   protected abstract void FF();
   protected abstract void GG();
}
```

Ở đây, các lớp không trừu tượng được dẫn xuất từ **SomeAbstClass** phải cho phủ quyết những hàm FF và GG, như vậy thiết đặt thực sự giao diện **IStorable**.

Ngoài ra, *giao diện được xem như gần giống một lớp trừu tượng*. Thật thế, bạn có thể thay đổi khai báo của giao diện **IStorable** thành một lớp trừu tượng **Storable** như sau:

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

Lớp **Document** giờ đây có thể kế thừa những thành viên từ **Storable**, không khác chi mấy so với khi sử dụng giao diện **IStorable**.

Tuy nhiên, giả sử bạn đi mua một lớp **List** nào đó từ thị trường phần mềm, với mong muốn phối hợp khả năng của **List** với khả năng của lớp **Storable**. Trên C++, bạn có thể tạo một lớp **StorableList**, thừa hưởng khả năng của **List** lẫn **Storable.** Nhưng trong C#, bạn sẽ bị kẹt ngay, vì C# không cho phép multiple inheritance (kế thừa nhiều nơi) đối với các lớp, do đó bạn không thể kế thừa từ cả lớp trừu tượng **Storable** lẫn lớp **List**.

Tuy nhiên, C# lại cho phép bạn thiết đặt bất cứ bao nhiêu giao diện cũng được, và cho dẫn xuất duy nhất từ một lớp cơ sở. Do đó, khi cho **Storable** thành một giao diện, bạn có thể kế thừa từ lớp **List**, đồng thời từ **IStorable**, giống như **StrorableList** đã làm như sau:

```
public class StorableList: List, IStorable
{
    // Các hàm hành sự của List…
    public void Read() {…}
    public void Write(object obj) {…}

    // …
}
```

Bạn để ý: Khi một lớp kế thừa từ một lớp cơ sở, ở đây là **List,** và một giao diện, ở đây là **IStorable** thì trên base-list bạn phải kê khai lớp cơ sở trước rồi sau đó mới đến các giao diện.

9.2.6 Giao diện so sánh với lớp cơ sở

Một kiểu dữ liệu giao diện (interface type) là sự mô tả phần nào một trị, tiềm tàng được hỗ trợ bởi nhiều kiểu dữ liệu đối tượng. Bất cứ lúc nào, bạn nên sử dụng những lớp cơ sở (base class) càng nhiều càng tốt thay vì giao diện. Nhìn theo viễn ảnh tạo phiên bản (versioning) khác nhau, lớp thường uyển chuyển hơn là với giao diện. Với một lớp, bạn có thể tung ra Version 1.0 rồi sau đó Version 2.0, và bổ sung thêm một hàm hành sự mới vào lớp. Miễn là lớp không phải trừu tượng, thì bất cứ những lớp dẫn xuất hiện hữu tiếp tục không thay đổi.

Vì giao diện không hỗ trợ thiết đặt kế thừa (implementation inheritance), đường lối áp dụng đối với lớp không áp dụng được đối với giao diện. Thêm một hàm hành sự vào một giao diện chẳng khác nào thêm một hàm hành sự trừu tượng vào một lớp cơ sở; và bất cứ lớp nào thiết đặt giao diện sẽ tiêu tùng vì lớp không thiết đặt hàm hành sự mới.

Giao diện chỉ thích hợp trong những trường hợp sau đây:

- Nhiều lớp không dính dáng chi nhau muốn hỗ trợ một nghi thức (protocol)
- Những lớp nào đã thiết lập những lớp cơ sở (thí dụ, một vài lớp là những ô user interface (UI) controls, và một vài lớp khác là dịch vụ XML Web).
- Aggregation không thích ứng hoặc thực tiển.

Còn trong những trường hợp khác, kế thừa lớp (class inheritance) là một mô hình tốt hơn cả.

9.3 Phủ quyết thiết đặt giao diện

Một lớp thi công sẽ tự do đánh dấu bất cứ hàm nào hoặc toàn bộ các hàm hành sự thiết đặt giao diện như là virtual (ảo). Các lớp được dẫn xuất có thể **phủ quyết** (override) hoặc cung cấp thi công **new**. Thí dụ, lớp **Document** muốn thiết đặt giao diện **IStorable** và cho đánh dấu các hàm **Read()** và **Write()** như là virtual. **Document** có thể **Read()** và **Write()** nội dung của nó về một kiểu dữ liệu **File**. Nhà triển khai phần mềm sau đó có thể dẫn xuất những kiểu dữ liệu mới khác từ **Document**, chẳng hạn kiểu **Note** hoặc **EmailMessage,** và quyết định rằng **Note** sẽ đọc và viết lên một database thay vì lên một tập tin (file).

Thí dụ 9-5 lột bỏ sự phức tạp của thí dụ 9-4, và minh hoạ việc phủ quyết một thi công giao diện. Hàm hành sự **Read()** được đánh dấu là virtual, và được thi công bởi

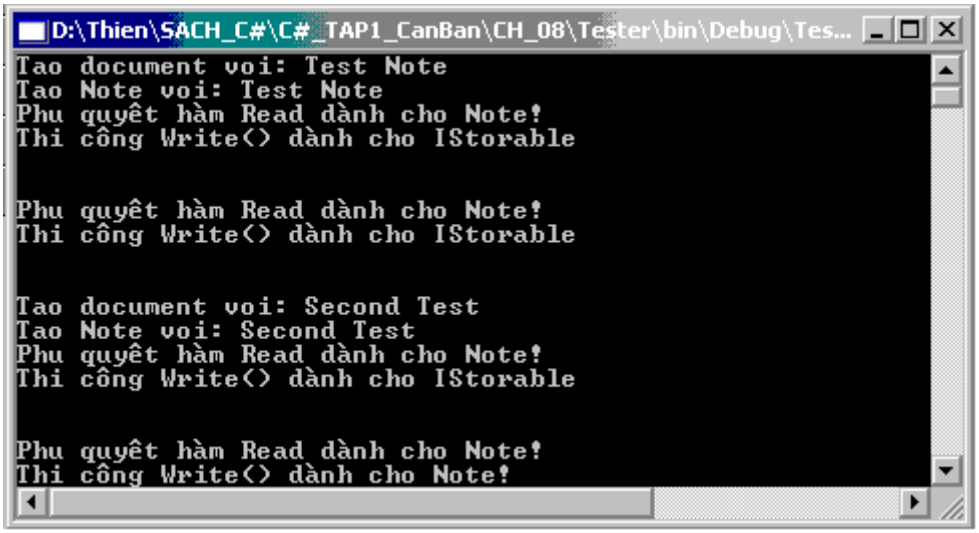
Document. **Read()** sau đó được phủ quyết trong một kiểu dữ liệu **Note** được dẫn xuất từ **Document**.

Thí dụ 9-5: Phủ quyết một thiết đặt giao diện

```
using System;

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

// Tạo một lớp Document cho thiết đặt interface IStorable
public class Document: IStorable
{   public Document(string s)         // hàm constructor
    {   Console.WriteLine("Tạo document với: {0}", s);
    }

    // Thi công hàm hành sự Read(), cho về virtual
    public virtual void Read()
    {   Console.WriteLine("Thi công Read() dành cho IStorable");
    }

    // Thi công hàm hành sự Write(), không virtual
    public void Write()
    {   Console.WriteLine("Thi công Write() dành cho IStorable");
    }
}   // end Document

// Tạo một lớp mới Note được dẫn xuất từ Document
public class Note: Document
{   public Note(string s): base(s)       // hàm constructor
    {   Console.WriteLine("Tạo Note với: {0}", s);
    }

    // Phủ quyết hàm hành sự Read()
    public override void Read()
    {   Console.WriteLine("Phủ quyết hàm Read dành cho Note!");
    }

    // Note tự thiết đặt hàm Write() của mình
    public void Write()
    {   Console.WriteLine("Thi công Write() dành cho Note!");
    }
}   // end Note

public class Tester
{
    static void Main()
    {   // Truy xuất các hàm hành sự trên đối tượng Document
        Document theNote = new Note("Test Note");
        IStorable isNote = theNote as IStorable;
```

```
        if (isNote != null)
        {   isNote.Read();
            isNote.Write();
        }
        Console.WriteLine("\n");

        // Triệu gọi trực tiếp các hàm
        theNote.Read();
        theNote.Write();

        Console.WriteLine("\n");

        // Tạo một đối tượng Note
        Note note2 = new Note("Second Test");
        IStorable isNote2 = note2 as IStorable;
        if (isNote != null)
        {   isNote2.Read();
            isNote2.Write();
        }
        Console.WriteLine("\n");

        // triệu gọi trực tiếp các hàm
        note2.Read();
        note2.Write();
    }   // end Main
}   // end Tester
```

Hình sau đây cho thấy kết xuất của thí dụ 9-5 trên:



Trong thí dụ này, lớp **Document** thiết đặt một giao diện **IStorable** được đơn giản hoá để cho thí dụ rõ ràng hơn:

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

Ta chọn cho **Read()** virtual, còn **Write()** thì không virtual:

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

Thật ra, trong thực tế bạn nên cho cả hai thành virtual. Nhưng ở đây, ta muốn cho thấy là ta toàn quyền lựa chọn hàm nào cho thành virtual.

Lớp mới **Note** được dẫn xuất từ **Document**. Không nhất thiết đối với **Note** là cho phủ quyết **Read()**, nhưng ta toàn quyền làm thế, và đã làm như thế:

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

Trong **Tester,** các hàm **Read()** và **Write()** được triệu gọi đến 4 cách thức khác nhau:

1. Thông qua việc qui chiếu lớp cơ sở chĩa về một đối tượng được dẫn xuất.

2. Thông qua một giao diện được tạo từ một qui chiếu lớp cơ sở chĩa về một đối tượng được dẫn xuất.

3. Thông qua một đối tượng được dẫn xuất.

4. Thông qua một giao diện được tạo từ đối tượng được dẫn xuất.

Để thực hiện hai triệu gọi đầu tiên, một qui chiếu **Document** (base class) được tạo ra, theNote, và vị chỉ của một đối tượng **Note** (được dẫn xuất) mới được tạo ra trên ký ức heap và được gán cho qui chiếu **Document,** theNote:

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

Một qui chiếu giao diện, **isNote**, được tạo ra, và tác tử **as** được dùng để ép kiểu **Document** về qui chiếu giao diện **IStorable:**

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

Sau đó, bạn triệu gọi **Read()** và **Write()** thông qua giao diện **IStorable**. Kết xuất cho thấy hàm hành sự **Read()** phản ứng một cách đa hình (polymorphically), còn **Write()** thì không. Đúng như ý muốn của ta:

```
Phủ quyết hàm Read dành cho Note!
Thi công Write() dành cho IStorable
```

Sau đó, các hàm được triệu gọi trực tiếp đối với bản thân đối tượng:

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

và một lần nữa, bạn thấy là cách thiết đặt đa hình hoạt động như theo ý muốn:

```
Phủ quyết hàm Read dành cho Note!
Thi công Write() dành cho IStorable
```

Trong cả hai trường hợp, hàm **Read()** của **Note** được gọi vào, nhưng lại là hàm **Write()** của **Document** được triệu gọi .

Để tự chứng minh đây là kết quả của hàm bị phủ quyết, bạn tạo kế tiếp một đối tượng **Note** thứ hai, note2, lần này gán vị chỉ về cho một qui chiếu giao diện **Note**, isNote2. Lần này để chứng minh cho hai trường hợp triệu gọi chót (nghĩa là triệu gọi thông qua một đối tượng được dẫn xuất, và một triệu gọi thông qua một giao diện được tạo từ đối tượng được dẫn xuất):

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

Một lần nữa, khi bạn cast về một qui chiếu, hàm **Read()** bị phủ quyết sẽ được triệu gọi. Tuy nhiên, khi các hàm được triệu gọi trực tiếp lên đối tượng **Note**:

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

kết xuất phản ảnh bạn cho triệu gọi đối tượng **Note** chứ không phải đối tượng **Document** bị phủ quyết:

```
Phủ quyết hàm Read dành cho Note!
Thi công Write() dành cho Note
```

9.4 Thiết đặt giao diện tường minh (explicit interface implementation)

Trong việc thiết đặt giao diện mà chúng ta đã đề cập mãi tới giờ, lớp lo thiết đặt giao diện (ở đây là **Document**) tạo một hàm hành sự thành viên cùng mang dấu ấn (signature)

cũng như kiểu dữ liệu trả về giống như hàm đã được chi tiết hoá trên giao diện. Ta khỏi phải khai một cách tường minh (explicit) đây là một thiết đặt giao diện, trình biên dịch đã hiểu ngầm như thế.

Tuy nhiên, việc gì sẽ xảy ra, nếu lớp thiết đặt cùng lúc hai giao diện, mỗi giao diện có một hàm hành sự cùng mang một dấu ấn giống nhau? Thí dụ 9-6 dưới đây sẽ tạo 2 giao diện: **IStorable** và **ITalk**. Giao diện **ITalk** này lại cho thiết đặt một hàm **Read()** lo đọc to lên nội dung một quyển sách. Như vậy, là có xung khắc với **Read()** trên **IStorable**.

Vì cả hai giao diện, **IStorable** và **ITalk**, đều có hàm **Read()**, nên lớp lo thiết đặt giao diện, **Document,** phải sử dụng đến việc *thiết đặt tường minh* (explicit implementation), đối với ít nhất một hàm, hàm **Read()**. Như vậy, lớp lo thiết đặt (**Document**) sẽ nhận diện một cách tường minh hàm hành sự giao diện nào:

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

Việc này giải quyết xung khắc về tên hàm, nhưng lại gây ra một loạt phản ứng phụ khá ấn tượng.

Trước tiên, khỏi phải dùng đến thiết đặt tường minh đối với hàm hành sự **Talk()** của **ITalk**. Vì không có xung khắc với hàm này, nên ta có thể khai báo hàm này một cách bình thường:

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

Điều quan trọng là *hàm nào được thiết đặt một cách tường minh, thì không thể có một access modifier* (nghĩa là các từ chốt public, private…):

```
void ITalk.Read();  // chứ không được viết:
                    // public void ITalk.Read();
```

Hàm Read() trên được hiểu ngầm là public.

Ngoài ra, một hàm hành sự được khai báo là thiết đặt tường minh, thì không được khai báo với những modifier: **abstract, virtual, override**, hoặc **new**.

Một điều quan trọng khác là bạn không thể truy xuất hàm hành sự được thiết đặt tường minh thông qua ngay bản thân đối tượng. Khi bạn viết:

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

thì trình biên dịch ngụ ý bạn muốn nói đến giao diện được thiết đặt tường minh đối với **IStorable**. Cách duy nhất để truy xuất một giao diện được thiết đặt tường minh là thông qua việc ép về một giao diện:

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

Thí dụ 9-6 minh hoạ việc sử dụng thiết đặt tường minh:

Thí dụ 9-6: Sử dụng thiết đặt tường minh

```
using System;

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

interface ITalk
{   void Read();
    void Talk();
}

// Tạo một lớp Document cho thiết đặt interface IStorable và ITalk
public class Document: IStorable, ITalk
{   public Document(string s)     // hàm constructor
    {   Console.WriteLine("Tạo document với: {0}", s);
    }

    // Thi công hàm hành sự Read(), cho về virtual
    public virtual void Read()
    {   Console.WriteLine("Thi công Read() dành cho IStorable");
    }

    // Thi công hàm hành sự Write(), không virtual
    public void Write()
    {   Console.WriteLine("Thi công Write() dành cho IStorable");
    }

    // Thi công hàm hành sự Read của giao diện ITalk()
    void ITalk.Read()
    {   Console.WriteLine("Thi công Read() dành cho ITalk");
    }

    // Thi công hàm hành sự Talk của giao diện ITalk()
    public void Talk()
    {   Console.WriteLine("Thi công Talk dành cho ITalk");
    }
}   // end Document
```

```
public class Tester
{   static void Main()
    {   // truy xuất các hàm hành sự trên đối tượng Document
        Document theDoc = new Document("Test Document");
        IStorable isDoc = theDoc as IStorable;
        if (isDoc != null)
        {   isDoc.Read();
        }
        Console.WriteLine("\n");

        ITalk itDoc = theDoc as ITalk;
        if (iiDoc != null)
        {   itDoc.Read();
        }
        Console.WriteLine("\n");

        // triệu gọi trực tiếp các hàm
        theDoc.Read();
        theDoc.Talk();
    }   // end Main
}   // end Tester
```

Kết xuất

Tạo document với: Test Document
Thi công Read() dành cho IStorable

Thi công Read() dành cho ITalk

Thi công Read() dành cho IStorable
Thi công Talk dành cho ITalk

9.4.1 Cho trưng ra một cách có lựa chọn những hàm hành sự giao diện

Khi thiết kế lớp đối tượng, bạn có thể lợi dụng sư kiện là khi một giao diện được thiết đặt một cách tường minh, giao diện sẽ không hiện lên trước khách hàng của lớp lo thiết đặt giao diện, ngoại trừ thông qua ép kiểu.

Giả sử, vì ý nghĩa của đối tượng **Document,** bạn buộc lòng phải thiết đặt giao diện **IStorable** nhưng bạn lại không muốn các hàm hành sự **Read()** và **Write()** lại thuộc thành phần của giao diện public của đối tượng **Document** của bạn. Bạn có thể sử dụng thiết đặt tường minh để bảo đảm các hàm này sẽ không có sẵn ngoại trừ thông qua ép kiểu. Việc

này cho phép bạn bảo toàn ý nghĩa của lớp **Document** trong khi vẫn tiếp tục thiết đặt giao diện **IStorable**. Nếu khách hàng của bạn muốn một đối tượng thiết đặt giao diện **IStorable,** nó có thể thực hiện một ép kiểu tường minh (explicit cast), nhưng khi sử dụng tài liệu *như là một đối tượng Document*, thì ý nghĩa sẽ không cho bao gồm **Read()** và **Write()**.

Thật thế, bạn có thể chọn những hàm hành sự nào cho hiện lên thông qua thiết đặt tường minh, như vậy bạn có thể trưng ra một vài hàm hành sự thiết đặt như là thành viên của **Document**, một số khác thì không. Trên thí dụ 9-5, đối tượng **Document** cho trưng hàm **Talk()** như là hàm hành sự của **Document**, nhưng hàm **Talk.Read()** chỉ có thể triệu gọi thông qua một việc ép kiểu. Cho dù **IStorable** không có hàm **Read()**, bạn có thể chọn cho **Read()** được thiết đặt tường minh, như vậy bạn khỏi trưng **Read()** như là một hàm hành sự của **Document**.

Bạn để ý, vì thiết đặt tường minh giao diện ngăn không cho ta dùng từ chốt **virtual**, một lớp dẫn xuất buộc phải thiết đặt lại hàm hành sự. Do đó, nếu **Note** được dẫn xuất từ **Document**, ta buộc phải thiết đặt lại **Talk.Read()** vì thiết đặt **Document** đối với **Talk.Read()** không thể dùng virtual.

9.4.2 Cho ẩn một thành viên giao diện

Ta có thể cho ẩn mình (hidden) một giao diện thành viên. Thí dụ, giả sử bạn có một giao diện **IBase** với một thuộc tính **P**:

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

và bạn cho dẫn xuất từ giao diện này một giao diện mới khác, **IDerived**. Giao diện dẫn xuất này cho ẩn mình thuộc tính **P** dùng đến hàm mới **P()**:

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

Gạt qua một bên việc liệu xem đây có phải là một điều tốt lành hay không, giờ đây bạn đã cho ẩn mình thuộc tính **P** trên giao diện cơ sở. Một thiết đặt của giao diện dẫn xuất này đòi hỏi ít nhất một giao diện tường minh thành viên. Bạn có thể sử dụng thiết đặt tường minh đối với *hoặc* thuộc tính cơ sở *hoặc* hàm hành sự dẫn xuất, hoặc bạn có

thể sử dụng thiết đặt tường minh đối với cả hai. Do đó, bất cứ 3 phiên bản sau đây đều hợp lệ:

```
class myClass: IDerived
{   // thiết đặt tường minh đối với thuộc tính cơ sở
    int IBase.P {get {…} }

    // thiết đặt implicit hàm hành sự dẫn xuất
    public int P() {…}
}

class myClass: IDerived
{   // thiết đặt implicit đối với thuộc tính cơ sở
    public int P {get {…}}

    // thiết đặt tường minh hàm hành sự dẫn xuất
    int IDerived.P() {…}
}

class myClass: IDerived
{   // thiết đặt tường minh đối với thuộc tính cơ sở
    int IBase.P {get {…}}

    // thiết đặt tường minh hàm hành sự dẫn xuất
    int IDerived.P() {…}
}
```

9.4.3 Truy xuất các lớp vô sinh và kiểu trị

Thông thường, người ta truy xuất các hàm hành sự giao diện thông qua một ép kiểu giao diện (interface cast), ngoại trừ đối với các kiểu dữ liệu kiểu trị (như structs chẳng hạn), hoặc đối với các lớp vô sinh (sealed). Trong trường hợp này, tốt hơn là triệu gọi hàm hành sự giao diện thông qua đối tượng, thay vì thông qua một ép kiểu giao diện.

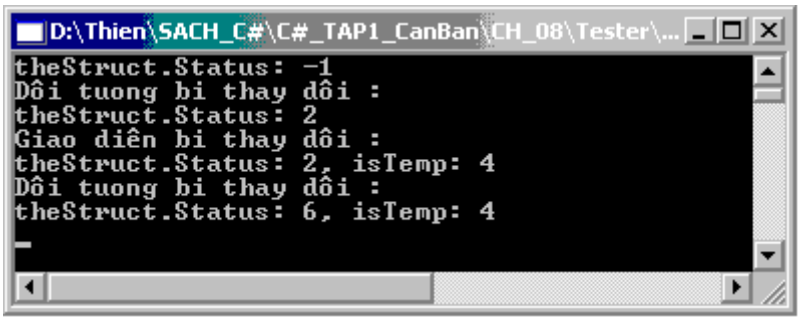
Khi bạn thiết đặt một giao diện trên một struct, bạn đang thiết đặt nó trên một kiểu trị. Khi bạn ép kiểu về một qui chiếu giao diện, thì đã xảy ra một đóng hộp ngầm đối với đối tượng. Rất tiếc là khi bạn dùng giao diện này để thay đổi đối tượng, thì chính đối tượng đã bị đóng hộp chứ không phải đối tượng nguyên thủy bị thay đổi. Ngoài ra, nếu bạn thay đổi kiểu trị, thì kiểu bị đóng hộp sẽ giữ nguyên hiện trạng. Thí dụ 9-7 sau đây sẽ tạo một struct có thiết đặt giao diện **IStorable**, và minh hoạ tác động của implicit boxing khi bạn ép kiểu struct về một qui chiếu giao diện:

Thí dụ 9-7: Qui chiếu về các kiểu trị

```
using System;

// Khai báo một giao diện đơn giản
interface IStorable
{   void Read();
    int Status {get; set;}
}

// Tạo một struct cho thiết đặt interface IStorable
public struct myStruct:IStorable
{   // thi công hàm hành sự Read()
    public void Read()
    {   Console.WriteLine("Thi công Read() dành cho IStorable");
    }

    // thi công thuộc tính Status
    public int Status
    {   get
        {   return status;
        }
        set
        {   status = value;
        }
    }

    // trữ trị cho thuộc tính status
    private int status;
}   // end myStruct

public class Tester
{
    static void Main()
    {   // tạo một đối tượng myStruct
        myStruct theStruct = new myStruct();
        theStruct.Status = -1;  // khởi gán
        Console.WriteLine("theStruct.Status: {0}", theStruct.Status);

        // thay đổi trị
        theStruct.Status = 2;
        Console.WriteLine("Đối tượng bị thay đổi:");
        Console.WriteLine("theStruct.Status: {0}", theStruct.Status);

        // ép kiểu về một giao diện IStorable;
        // implicit boxing về một kiểu qui chiếu
        IStorable isTemp = (IStorable) theStruct;

        // cho đặt để trị thông qua qui chiếu giao diện
        isTemp.Status = 4;
        Console.WriteLine("Giao diện bị thay đổi:");
        Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
            theStruct.Status, isTemp.Status);

        // thay đổi trị một lần nữa
        theStruct.Status = 6;
```

```
        Console.WriteLine("Đối tượng bị thay đổi:");
        Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
            theStruct.Status, isTemp.Status);
    }   // end Main
}   // end Tester
```

Hình sau đây cho thấy kết xuất của thí dụ 9-7:



Trong thí dụ 9-7, giao diện **IStorable** có một hàm hành sự **Read()** và một thuộc tính **Status**. Giao diện này được thiết đặt bởi struct mang tên **myStruct**:

```
public struc myStruct: IStorable
```

Đoạn mã khá lý thú nằm trong phần **Tester**. Bạn bắt đầu tạo một thể hiện cấu trúc, `theStruct`, rồi khởi gán trị -1 cho thuộc tính **Status**. Trị thuộc tính này sẽ được in ra:

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

Kết xuất cho thấy **Status** được đặt để dúng theo ý muốn:

```
theStruct.Status: -1
```

Tiếp theo, bạn truy xuất thuộc tính để thay đổi trị, một lần nữa thông qua bản thân trị đối tượng:

```
theStruct.Status = 2;
Console.WriteLine("Đối tượng bị thay đổi:");
Console.WriteLine("theStruct.Status: {0}", theStruct.Status);
```

Kết xuất cho thấy thay đổi này:

```
Đối tượng bị thay đổi:
theStruct.Status: 2
```

Chả có gì đáng ngạc nhiên. Tới đây, bạn tạo một qui chiếu về giao diện **IStorable**. Điều này gây ra một implicit boxing của **theStruct**. Sau đó, bạn dùng giao diện này để thay đổi trị của **Status** cho về 4:

```
IStorable isTemp = (IStorable) theStruct;
isTemp.Status = 4;
Console.WriteLine("Giao diện bị thay đổi:");
Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
   theStruct.Status, isTemp.Status);
```

Và lần này, kết xuất có hơi ngỡ ngàng:

```
Giao diện bị thay đổi:
theStruct.Status: 2, isTemp: 4
```

vì đối tượng mà qui chiếu giao diện, **isTemp**, chỉ về đã bị thay đổi trị thành 4, nhưng trị đối tượng struct thì lại không thay đổi. Điều thú vị hơn là khi bạn truy xuất thuộc tính thông qua bản thân đối tượng **theStruct**:

```
theStruct.Status = 6;
Console.WriteLine("Đối tượng bị thay đổi:");
Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
   theStruct.Status, isTemp.Status);
```

kết xuất cho thấy đối tượng trị bị thay đổi, còn trị qui chiếu bị boxed đối với qui chiếu giao diện thì không nhúc nhích:

```
Đối tượng bị thay đổi:
theStruct.Status: 6, isTemp: 4
```

Nếu ta nhìn nhanh vào đoạn mã kết sinh IL sau đây (thí dụ 9-8) ta sẽ thấy việc gì xảy ra ở hậu trường:

Thí dụ 9-8: Đoạn mã IL được kết sinh từ thí dụ 9-7

```
.method private hidebysig static void Main() cil managed
{
 .entrypoint
 // Code size    193 (0xc1)
 .maxstack 4
 .locals init ([0] valuetype Tester.myStruct theStruct,
      [1] class Tester.IStorable isTemp)
 IL_0000: ldloca.s  theStruct
 IL_0002: initobj  Tester.myStruct
```

```
IL_0008: ldloca.s  theStruct
IL_000a: ldc.i4.m1
IL_000b: call    instance void
                              Tester.myStruct::set_Status(int32)
IL_0010: ldstr   "theStruct.Status: {0}"
IL_0015: ldloca.s  theStruct
IL_0017: call    instance int32 Tester.myStruct::get_Status()
IL_001c: box     [mscorlib]System.Int32
IL_0021: call    void [mscorlib]System.Console::WriteLine
                            (string, object)
IL_0026: ldloca.s  theStruct
IL_0028: ldc.i4.2
IL_0029: call    instance void Tester.myStruct::
                                                  set_Status(int32)
IL_002e: ldstr   bytearray (44 00 F4 00 69 00 20 00 74 00 75 00
                            6F 00 6E 00  // D...i. .t.u.o.n.
            67 00 20 00 62 00 69 00 20 00 74 00 68
                            00 61 00  // g. .b.i. .t.h.a.
            79 00 20 00 64 00 F4 00 69 00 20 00 3A
                            00 )    // y. .d...i. .:.
IL_0033: call    void [mscorlib]System.Console::
                                                    WriteLine(string)
IL_0038: ldstr   "theStruct.Status: {0}"
IL_003d: ldloca.s  theStruct
IL_003f: call    instance int32 Tester.myStruct::get_Status()
IL_0044: box     [mscorlib]System.Int32
IL_0049: call    void [mscorlib]System.Console::
                            WriteLine(string, object)
IL_004e: ldloc.0
IL_004f: box     Tester.myStruct
IL_0054: stloc.1
IL_0055: ldloc.1
IL_0056: ldc.i4.4
IL_0057: callvirt  instance void Tester.IStorable::
                                                  set_Status(int32)
IL_005c: ldstr   bytearray (47 00 69 00 61 00 6F 00 20 00 64 00
                               69 00 EA 00  // G.i.a.o. .d.i...
                 6E 00 20 00 62 00 69 00 20 00 74 00 68
                               00 61 00  // n. .b.i. .t.h.a.
                 79 00 20 00 64 00 F4 00 69 00 20 00 3A
                               00 )    // y. .d...i. .:.
IL_0061: call    void [mscorlib]System.Console::
                                       WriteLine(string)
IL_0066: ldstr   "theStruct.Status: {0}, isTemp: {1}"
IL_006b: ldloca.s  theStruct
IL_006d: call    instance int32 Tester.myStruct::get_Status()
IL_0072: box     [mscorlib]System.Int32
IL_0077: ldloc.1
IL_0078: callvirt  instance int32 Tester.IStorable::get_Status()

IL_007d: box     [mscorlib]System.Int32
IL_0082: call    void [mscorlib]System.Console::
                                    WriteLine(string, object, object)
IL_0087: ldloca.s  theStruct
```

```
  IL_0089: ldc.i4.6
  IL_008a: call     instance void Tester.myStruct::
                                                     set_Status(int32)
  IL_008f: ldstr    bytearray (44 00 F4 00 69 00 20 00 74 00 75 00
                               6F 00 6E 00  // D...i. .t.u.o.n.
                     67 00 20 00 62 00 69 00 20 00 74 00
                               68 00 61 00  // g. .b.i. .t.h.a.
                  79 00 20 00 64 00 F4 00 69 00 20 00 3A
                               00 )     // y. .d...i. .:.
  IL_0094: call     void [mscorlib]System.Console::
                                                     WriteLine(string)
  IL_0099: ldstr    "theStruct.Status: {0}, isTemp: {1}"
  IL_009e: ldloca.s  theStruct
  IL_00a0: call     instance int32 Tester.myStruct::get_Status()
  IL_00a5: box     [mscorlib]System.Int32
  IL_00aa: ldloc.1
  IL_00ab: callvirt  instance int32 Tester.IStorable::get_Status()
  IL_00b0: box     [mscorlib]System.Int32
  IL_00b5: call     void [mscorlib]System.Console::
                               WriteLine(string, object, object)
  IL_00ba: call     string [mscorlib]System.Console::ReadLine()
  IL_00bf: pop
  IL_00c0: ret
} // end of method Tester::Main
```

Trên hàng IL_000b, `set status` được triệu gọi trên đối tượng trị. Ta thấy lần triệu gọi thứ hai trên hàng IL_0017. Bạn để ý là các triệu gọi `WriteLine()` gây ra boxing đối với trị số nguyên của status do đó hàm hành sự `GetString()` có thể được triệu gọi.

Hàng chủ chốt IL_004f theo đấy bản thân struc bị boxed tạo ra một kiểu dữ liệu qui chiếu đối với interface reference. Bạn để ý trên hàng IL_0057 là lần này `IStorage::set_status` được triệu gọi thay vì `myStruct::setStatus`.

Nói tóm lại, nếu bạn thiết đặt một giao diện với một kiểu trị, một struct chẳng hạn, bạn nên chắc truy xuất các thành viên giao diện thông qua đối tượng thay vì thông qua một qui chiếu giao diện.

9.5 Dùng giao diện như là thông số

Giao diện được xem như là những biến được kiểm tra chặt chẽ về mặt kiểu dữ liệu, bạn có thể xây dựng những hàm hành sự dùng giao diện như là thông số cũng như là trị trả về. Sau đây là một thí dụ 9-9 về vẽ hình đồ học. Tới đây bạn đã "đủ lông đủ cánh" để hiểu thí dụ này. Chúng tôi đã cho chú thích từng câu lệnh, bạn đọc hiểu ngay.

Thí dụ 9-9: Vẽ hình đồ học

```
namespace Prog_CSharp
{   using System;

    // Giao diện IPointy dùng tìm số điểm trên một hình vẽ
    public interface IPointy
    {   byte GetNumberOfPoints();  // đi lấy số điểm

        // giao diện có thể có thuộc tính
        // byte Points{get; set;}
    }


    // Giao diện IDraw3D vẽ 3 chiều public interface IDraw3D
    public interface IDraw3D
    {    void Draw(); // vẽ
    }

    // Lớp cơ sở trừu tượng Shape vẽ hình dạng
    public abstract class Shape
    {   protected string petName;     // biến tên thân thương

        public Shape(){petName = "NoName";}  // hàm constructor 1,
                                             // không thông số

        public Shape(string s)     // hàm constructor 2,
                                   // một thông số chuỗi chữ
        {   this.petName = s;
        }

        // tất cả các đối tượng con phải tự mình định nghĩa cần muốn vẽ gì
        public abstract void Draw();

        public string PetName      // thuộc tính PetName (tên thân thương)
        {   get {return this.petName;}
            set {this.petName = value;}
        }
    }

    // Lớp dẫn xuất từ lớp cơ sở Shape và giao diện IDraw3D
    public class Circle: Shape, IDraw3D
    {   public Circle(){}   // hàm constructor, không thông số

        public Circle(string name): base(name)  // triệu gọi hàm
                                                // constructor lớp cơ sở
        {}

        public override void Draw()   // hàm vẽ bị phủ quyết
        {   Console.WriteLine("Vẽ " + PetName + " Hình Tròn");
        }

        void IDraw3D.Draw()     // hàm tường minh giao diện vẽ 3 chiều
```

```
    {   Console.WriteLine("Vẽ Hình Tròn 3D!");
    }
}   // end Circle

// Lớp dẫn xuất từ lớp cơ sở Shape, giao diện IPointy và IDraw3D
public class Hexagon: Shape, IPointy, IDraw3D
{   public Hexagon(){}  // hàm constructor
    public Hexagon(string name): base(name) // triệu gọi hàm
                                            // constructor lớp cơ sở
    {}

    public override void Draw()   // hàm vẽ bị phủ quyết
    {   Console.WriteLine("Vẽ " + PetName + " Hình Lục Giác");
    }

    public byte GetNumberOfPoints()  // hàm lấy số điểm
    {   return 6;
    }

    void IDraw3D.Draw()     // hàm tường minh giao diện vẽ 3 chiều
    {   Console.WriteLine("Vẽ Hình Lục Giác 3D!");
    }
}   // end Hexagone

// Lớp dẫn xuất từ lớp cơ sở Shape, và giao diện IPointy
public class Triangle: Shape, IPointy
{   public Triangle(string name): base(name) {}
    public Triangle() {}

    public override void Draw()   // hàm vẽ bị phủ quyết
    {   Console.WriteLine("Vẽ " + PetName + " Hình Tam Giác");
    }

    public byte GetNumberOfPoints()  // hàm lấy số điểm
    {   return 3;
    }
}   // end Triangle

// Lớp dẫn xuất từ lớp cơ sở Circle (lớp này lại là
// một lớp dẫn xuất )
public class Oval: Circle
{   public Oval(){base.PetName = "Joe";} // hàm constructor

    new public void Draw() // hàm riêng mới của lớp Oval
    {   Console.WriteLine("Vẽ một Hình Bầu Dục dùng một giải thuật
                        quái quỷ");
    }
}   // end Oval

// Lớp dẫn xuất từ lớp cơ sở Shape và giao diện IDraw3D
public class Line: Shape, IDraw3D
{   void IDraw3D.Draw()     // hàm tường minh giao diện vẽ 3 chiều
    {   Console.WriteLine("Vẽ một đường thẳng 3D ...");
    }
```

```
    public override void Draw()   // hàm vẽ bị phủ quyết
    {   Console.WriteLine("Vẽ một đường thẳng ...");
    }
}   // end Line

public class Tester
{
    // Tôi sẽ vẽ bất cứ hình nào cũng được
    public static void DrawThisShapeIn3D(IDraw3D i3d) // thông số là
                                                      // giao diện
    {   i3d.Draw();
    }

    public static int Main(string[] args)
    {   Line l = new Line();
        l.Draw(); // vẽ đường thẳng bình thường

        IDraw3D iDraw3d= (IDraw3D)l;
        iDraw3d.Draw();  // vẽ đường thẳng 3 chiều

        // Cách thứ nhất để có một giao diện
        Circle c = new Circle("Mitch");
        IPointy iPt;
        try
        {
            iPt = (IPointy)c;
            Console.WriteLine("Đi lấy giao diện dùng ép kiểu...");
        }
        catch(InvalidCastException e)
        {Console.WriteLine("OOPS! Không có điểm...");}


        // Cách thứ hai để lấy một giao diện
        Hexagon hex = new Hexagon("Fran");
        IPointy iPt2;
        iPt2 = hex as IPointy;
        if(iPt2 != null)
            Console.WriteLine("Đi lấy giao diện dùng từ chốt as ");
        else
            Console.WriteLine("OOPS! Không có điểm...");

        // Cách thứ ba để chụp lấy giao diện
        Triangle t = new Triangle();
        if(t is IPointy)
            Console.WriteLine("Đi lấy giao diện dùng từ chốt is");
        else
            Console.WriteLine("OOPS! Không có điểm...");

        Console.WriteLine();    // nhảy một hàng

        // Mẹo C# base class pointer.
        // Tạo một collection Shape là một bản dãy
        Shape[] sh = {new Hexagon(), new Circle(),
    new Triangle("Joe"), new Circle("JoJo")};
```

```
            // Rảo qua collection
            for(int i = 0; i < sh.Length; i++)
            {   sh[i].Draw();

                // Anh nào có điểm?
                if(sh[i] is IPointy)
                    Console.WriteLine("Số điểm: {0}\n",
                                ((IPointy)sh[i]).GetNumberOfPoints());
                else
                    Console.WriteLine(sh[i].PetName + "không có điểm!\n");

                // Ta vẽ anh theo 3D được không ?
                if(sh[i] is IDraw3D)
                    DrawThisShapeIn3D((IDraw3D)sh[i]);
                    Console.WriteLine();
            }
            return 0;
        } // end Main
    }   // end Tester
}   // namespace Prog_CSharp
```

9.6 Thiết đặt kế thừa giao diện

Thông thường một lớp sẽ kế thừa tất cả các thiết đặt giao diện (interface implemen tations) mà các lớp cơ sở cung cấp.

Nếu không có việc thiết đặt lại một cách tường minh một giao diện, thì một lớp dẫn xuất không thể bất cứ cách nào gỡ bỏ việc ánh xạ giao diện mà nó thừa hưởng từ các lớp cơ sở. Thí dụ: ta có một giao diện **IControl**, một lớp **Control** thiết đặt giao diện **IControl**, và một lớp dẫn xuất **TextBox** từ lớp cơ sở **Control:**

```
// Giao diện IControl
interface IControl
{   void Paint();
}

// Lớp thiết đặt giao diện IControl
class Control:Icontrol
{   public void Paint() {…}// hàm Paint () được thi công
}

// Lớp TextBox được dẫn xuất từ lớp Control
class TextBox: Control
{     new public void Paint() {….} // hàm này cho ẩn mình
                                   // hàm kế thừa giao diện
}
```

Bạn thấy hàm hành sự **Paint()** trong lớp **TextBox** cho ẩn mình hàm **Paint()** trên lớp **Control,** nhưng không thể gỡ bỏ ánh xạ của **Control.Paint()** trên **IControl.Paint()** và việc triệu gọi thông qua thể hiện lớp (class instance) và thông qua thể hiện giao diện (interface instance) sẽ có những tác dụng sau đây:

```
Control c = new Control(); // c là thể hiện lớp Control
                        // có thiết đặt giao diện IControl
TextBox t = new TextBox(); // t là thể hiện lớp dẫn xuất từ lớp
Control
IControl ic = c;      // ic là thể hiện giao diện chĩa về lớp Control
IControl it = t;      // it là thể hiện giao diện chĩa về
     // lớp TextBox, rồi lại chĩa về Control
c.Paint();    // triệu gọi hàm Control.Paint();
t.Paint();    // triệu gọi hàm TextBox.Paint();
ic.Paint();           // triệu gọi hàm Control.Paint();
it.Paint();           // triệu gọi hàm Control.Paint();
```

Tuy nhiên, khi một hàm hành sự giao diện được ánh xạ lên một hàm virtual trên một lớp thiết đặt giao diện này, thì có thể những lớp được dẫn xuất cho phủ quyết hàm virtual này và thay đổi việc thiết đặt giao diện. Thí dụ, ta viết lại những khai báo kể trên:

```
// Giao diện IControl
interface IControl
{  void Paint();
}

// Lớp thiết đặt giao diện IControl
class Control:Icontrol
{  public virtual void Paint() {…}  // hàm Paint () được thi công
                                    // và thành virtual
}

// Lớp TextBox được dẫn xuất từ lớp Control
class TextBox: Control
{  public override void Paint() {….}    // hàm này phủ quyết
                                        // hàm virtual
}
```

ta có thể nhận thấy những tác dụng sau đây:

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();    // triệu gọi Control.Paint();
t.Paint();    // triệu gọi TextBox.Paint();
ic.Paint();           // triệu gọi Control.Paint();
it.Paint();           // triệu gọi TextBox.Paint(); khác so với trước
```

Vì thiết đặt tường minh thành viên giao diện không thể khai báo là virtual, nên khó lòng phủ quyết một thiết đặt tường minh thành viên giao diện. Tuy nhiên, hoàn toàn hợp lệ đối với một thiết đặt tường minh thành viên giao diện triệu gọi một hàm hành sự khác, và hàm khác này có thể được khai báo là virtual cho phép những lớp dẫn xuất phủ quyết hàm này. Thí dụ:

```
// Gaio diện IControl
interface IControl
{   void Paint();
}

// Lớp thiết đặt giao diện IControl
class Control:Icontrol
{   void IControl.Paint() { PaintControl();} // thiết đặt tường minh
                                             // thành viên giao diện
    protected virtual void PaintControl() {...}
}

// Lớp TextBox được dẫn xuất từ lớp Control
class TextBox: Control
{   protected override void PaintControl() {….} // hàm này phủ quyết
                                                // hàm virtual
}
```

Ơ đây, những lớp dẫn xuất từ **Control,** có thể chuyên môn hoá (specialize) việc thiết đặt của **`IControl.Paint`** bằng cách phủ quyết hàm hành sự **`PaintControl`**.

9.7 Thiết đặt lại giao diện

Một lớp kế thừa một thiết đặt giao diện được phép tái thiết đặt (re-implement) giao diện bằng cách cho bao gồm giao diện trong base class list.

Việc tái thiết đặt giao diện tuân thủ đúng các qui tắc ánh xạ giao diện tương tự như thiết đặt ban đầu của một giao diện. Do đó, ánh xạ giao diện được kế thừa không ảnh hưởng chi lên việc ánh xạ giao diện được thiết lập do việc tái thiết đặt của giao diện. Thí dụ, trong những khai báo sau đây:

```
// Giao diện IControl
interface IControl
{   void Paint();
}
```

```
// Lớp thiết đặt giao diện IControl
class Control:Icontrol
{   void IControl.Paint() {…}  // explicit member implementation
}

class MyControl: Control, IControl
{   public void Paint() {}
}
```

sự kiện lớp **Control** ánh xạ **`IControl.Paint`** lên **`Control.IControl.Paint`** không ảnh hưởng chi lên việc tái thiết đặt giao diện **IControl** trên **MyControl** cho ánh xạ **IControl.Paint** lên **MyControl.Paint.**.

Những khai báo các thành viên được kế thừa và những khai báo thành viên giao diện tường minh được kế thừa sẽ tham gia vào tiến trình ánh xạ giao diện đối với những giao diện được tái thiết đặt. Thí dụ

```
interface IMethods
{   void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{   void IMethods.F() {}       // explicit implementation
    void IMethods.G() {}       // explicit implementation
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{   public void F() {}         // re-implementation bình thường
    void IMethods.H() {}       // re-implementation tường minh
}
```

Ở đây, việc tái thiết đặt giao diện **IMethods** trên lớp **Derived** ánh xạ những hàm hành sự giao diện trên **`Derived.F`**, **`Base.IMethods.G`**, **`Derived.IMethods.H`**, **and `Base.I`.**

Khi một lớp thiết đặt một giao diện, nó coi như hiểu ngầm thiết đặt tất cả các giao diện cơ sở của giao diện. Cũng như thế, một tái thiết đặt một giao diện cũng ngầm hiểu là tái thiết đặt tất cả các giao diện cơ sở của giao diện. Thí dụ

```
interface IBase
{   void F();
}
```

```
interface IDerived: IBase
{   void G();
}

class C: IDerived
{   void IBase.F() {…}
    void IDerived.G() {…}
}

class D: C, IDerived
{   public void F() {…}
    public void G() {…}
}
```

Ở đây, việc tái thay đổi giao diện **IDerived** cũng tái thiết đặt **IBase**, ánh xạ **IBase.F** lên D.F

9.8 Thí dụ về thiết đặt giao diện

Sau khi bạn đã nhận diện một giao diện hiện hữu có thể hữu ích đối với cấu kiện (component) của bạn, hoặc bạn đã định nghĩa những giao diện riêng cho bạn, bạn có thể thiết đặt giao diện này lên trên những lớp mà cấu kiện bạn cung cấp.

Giả sử bạn có một hệ thống BankManager bao gồm một số cấu kiện, mỗi cấu kiện tượng trưng cho các tài khỏan (account), các giao dịch tài chính (transactions), và những qui tắc kinh doanh (business rules). Thí dụ, một cấu kiện có thể cung cấp một lớp BusinessAccount, trong khi một cấu kiện khác cung cấp một lớp PersonalAccount.

Định nghĩa một giao diện chuẩn

Có thể bạn tạo một giao diện chuẩn mang tên **IAccount**, bao gồm các hàm hành sự **PostInterest** (ghi nhận lãi suất) và **DeductFees** (chi phí cấn trừ):

```
public interface IAccount
{
 void PostInterest();
void DeductFees(IFeeSchedule feeSchedule);
}
```

Bạn để ý đối mục của hàm hành sự **DeductFees** lại là một giao diện **IFeeSchedule**. Ta có thể chờ đợi giao diện này gồm những hàm mô tả tổng quát những đặc tính khách hàng, chẳng hạn cách tính phí giao dịch tài chính đề cập những tài khoản khác nhau và có

thể được thiết đặt bởi nhiều lớp -- BusinessAccount, PersonalAccount, PreferredAccount, v.v..

Ngoài ra, bạn thấy không có đoạn mã trong các hàm trên. **IAccount** đơn giản định nghĩa giao diện, còn chi tiết thi công tùy thuộc các đối tượng thiết đặt giao diện này.

Thiết đặt một giao diện chuẩn

Thí dụ, lớp **BusinessAccount** có thể thiết đặt giao diện **IAccount** như sau:

```
class BusinessAccount: IAccount
{   void IAccount.PostInterest()
    {   // Đoạn mã ghi nhận lãi suất dùng đến tỉ suất có lợi nhất.
    }

    void IAccount.DeductFees(IFeeSchedule feeSchedule)
    {   // Đoạn mã dùng thay đổi một tỉ suất ưu tiên
        // đối với những dịch vụ khác nhau
    }
}
```

Thiết đặt giao diện khác đi trên lớp khác

Ngược lại, lớp **PersonalAccount** có thể thiết đặt giao diện **IAccount**, như theo thí dụ sau đây:

```
class PersonalAccount: IAccount
{   void IAccount.PostInterest()
    {   // Tính lãi suất đối với những tài khỏ và ghi nhận.
    }

    void IAccount.DeductFees(IFeeSchedule feeSchedule)
    {   // Tính xem bao nhiêu dịch vụ mà khách hàng đã huởng
        // tương tác với nhau và cho tính tiền dựa vào
        // các dịch vụ này.
    }
}
```

Nâng cấp những ứng dụng để sử dụng một giao diện mới

Một khi bạn đã có những lớp thiết đặt **IAccount**, bạn có thể nâng cấp các ứng dụng hiện hữu từng ứng dụng một để sử dụng giao diện mới hiệu quả hơn. Bạn có thể truy xuất các hàm **PostInterest** và **DeductFees** bằng cách những đối tượng **BusinessAccount** hoặc **PersonalAccount** cho một biến kiểu **IAccount**, như theo những thí dụ sau đây

```
BusinessAccount business = new BusinessAccount();
IAccount account = business;
account.PostInterest();
```

Bạn cũng có thể khai báo những đối mục thủ tục như **IAccount** chẳng hạn và trao cho thủ tục bất cứ đối tượng nào thiết đặt giao diện IAccount, như theo thí dụ sau đây:

```
public void AccountMaintenance(IAccount account,
                               IFeeSchedule feeSchedule)
{
    account.PostInterest();
    account.DeductFees(feeSchedule);
}
```

Hàm kể trên có thể được triệu gọi với bất cứ **account** nào như là đối mục đầu tiên, và bất cứ **feeSchedule** nào như là đối mục thứ hai. Hàm triệu gọi thủ tục có thể sử dụng bất cứ account nào thích hợp đối với trường hợp.

Chương 10

Bản dãy, Indexers và Collections

.NET Framework cung cấp cho bạn một loạt lớp tập hợp (collection) rất phong phú bao gồm **Array, ArrayList, NameValueCollection, StringCollection, Queue, Stack** và **BitArray.**

Tập hợp đơn giản nhất là **Array,** lớp bản dãy, collection "bẩm sinh" duy nhất mà C# hỗ trợ. Trong chương này, bạn sẽ làm quen với những bản dãy một chiều (single array), bản dãy nhiều chiều (multidimensional array) và bản dãy "lởm chởm" (jagged array). Bạn cũng sẽ được dẫn nhập vào bộ chỉ mục (indexer) cho phép truy xuất những thuộc tính của lớp xem lớp như được chỉ mục giống một bản dãy.

.NET Framework cung cấp cho bạn một lô giao diện, như **IEnumerable** và **ICollection**, mà khi bạn thiết đặt lên lớp của bạn sẽ cho phép bạn một thể thức chuẩn để tương tác với các tập hợp. Trong chương này, bạn sẽ làm quen với một số giao diện thiết yếu biết chúng hoạt động thế nào với các collection. Chương này sẽ kết thúc bằng cách dẫn bạn đi dạo vòng quanh những .NET collection phổ biến nhất bao gồm **ArrayList, Dictionary, Hashtable, Queue** và **Stack.**

10.1 Bản dãy (array)

Bản dãy là một cấu trúc dữ liệu cấu tạo bởi một số biến được gọi là những phần tử bản dãy (array element). Tất cả các phần tử bản dãy này đều phải cùng thuộc một kiểu dữ liệu, bất cứ kiểu dữ liệu bẩm sinh nào được định nghĩa bởi C#, kể cả bản dãy các đối tượng, các giao diện hoặc các struct. Bạn có thể truy xuất phần tử bản dãy dựa theo một chỉ số (index). Bản dãy được chỉ mục theo cơ sở 0 (zero-based), nghĩa là chỉ số bắt đầu từ zero (chứ không phải bắt đầu từ 1 như theo lẽ thường tình).

Nhìn chung, trên C#, bản dãy hoạt động cũng tương tự như trên các ngôn ngữ lập trình khác, như C++ chẳng hạn. Tuy nhiên, có khác một chút vì bản dãy trên C# là những đối tượng, kiểu **System.Array**, nghĩa là bản dãy là một kiểu dữ liệu kiểu qui chiếu được dẫn xuất từ lớp cơ sở **System.Array**. Do đó, bản dãy có những thuộc tính và hàm hành sự rất hữu ích dùng để thao tác trên các phần tử bản dãy .

Bảng 10-1 sau đây liệt kê các hàm hành sự và thuộc tính của **System.Array**

Bảng 10-1: Các hàm hành sự và thuộc tính của System.Array

IsFixedSize	Trả về một trị bool cho biết **Array** có một kích thước cố định hay không .
IsReadOnly	Trả về một trị bool cho biết **Array** thuộc read-only hay không.
IsSynchronized	Trả về một trị bool cho biết việc truy xuất **Array** có đồng bộ hay không (thread-safe).
Length	Trả về tổng số phần tử bản dãy trên tất cả các chiều của **Array**.
Rank	Trả về trị số chiều (rank) của **Array**.
SyncRoot	Trả về một đối tượng mà ta có thể dùng để đồng bộ hoá truy xuất **Array**.
Hàm hành sự Public	
BinarySearch	Overloaded. Đi tìm một trị nào đó trên một bản dãy một chiều **Array** đã được sắp xếp, dùng đến giải thuật truy tìm nhị phân (binary search algorithm).
Clear	Cho về zero, về **false**, hoặc về null reference (dựa vào kiểu dữ liệu) đối với một phần các phần tử bản dãy **Array**.
Clone	Tạo một bản sao shallow của **Array**.
Copy	Overloaded. Cho sao một phần của một bản dãy **Array** lên một bản dãy **Array** khác và thực hiện ép kiểu (type casting) và đóng hộp (boxing) nếu thấy cần thiết.
CopyTo	Cho sao tất cả các phần tử bản dãy của bản dãy một chiều **Array** hiện hành lên một **Array** một chiều được chỉ định khởi đi từ chỉ số của **Array** nơi đến được chỉ định.
CreateInstance	Overloaded. Khởi gán một thể hiện mới đối với lớp **Array**.
Equals (kế thừa từ Object)	Overloaded. Xác lập xem hai thể hiện Object có giống nhau không .
GetEnumerator	Trả về cho **Array** một giao diện **IEnumerator.**
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng cho việc sử dụng trong các giải thuật băm và các cấu trúc dữ liệu giống như một hash table.
GetLength	Trả về số lượng phần tử bản dãy trên một chiều chỉ định của **Array**.
GetLowerBound	Trả về chỉ số thấp đối với một chiều được chỉ định của **Array**.
GetType (kế thừa từ Object)	Trả về kiểu dữ liệu **Type** của thể hiện hiện hành.
GetUpperBound	Trả về chỉ số cao đối với một chiều được chỉ định của bản dãy **Array**.

GetValue	Overloaded. Trả về trị của phần tử bản dãy **Array** hiện hành được chỉ định.
IndexOf	Overloaded. Trả về chỉ số của một trị xuất hiện lần đầu tiên trên một bản dãy **Array** một chiều hoặc trên một phần bản dãy **Array**.
Initialize	Khởi gán mọi phần tử bản dãy của **Array** kiểu trị (value-type) bằng cách triệu gọi hàm khởi dựng mặc nhiên (default constructor) của kiểu trị.
LastIndexOf	Overloaded. Trả về chỉ số của một trị xuất hiện lần chót trên một bản dãy **Array** một chiều hoặc trên một phần bản dãy **Array**.
Reverse	Overloaded. Đảo ngược thứ tự những phần tử bản dãy trên một bản dãy **Array** một chiều hoặc trên một phần bản dãy **Array**.
SetValue	Overloaded. Cho phần tử bản dãy được chỉ định trên bản dãy **Array** hiện hành về trị được chỉ định.
Sort	Overloaded. Sắp theo thứ tự các phần tử bản dãy một chiều **Array**.
ToString (kế thừa từ Object)	Trả về một **String** đại diện cho **Object** hiện hành.

Người ta phân biệt nhiều loại bản dãy: bản dãy một chiều, bản dãy nhiều chiều và đặc biệt bản dãy "lởm chởm" kiểu răng cưa. Chúng tôi sẽ lần lượt xem các loại bản dãy này.

Trước khi có thể sử dụng một bản dãy, bạn phải thực hiện một số bước:

(1) Bạn phải khai báo một bản dãy theo một cú pháp nào đó. Trong bước này, bạn cho biết kiểu dữ liệu và tên bản dãy. Thí dụ: **int [] myIntArray**. Cặp dấu [] cho trình biên dịch biết là bản dãy. int là kiểu dữ liệu số nguyên. **myIntArray** là tên biến bản dãy, được trữ trên ký ức stack.

(2) Bạn phải hiển lộ một đối tượng bản dãy, nghĩa là tạo ra một bản dãy trong ký ức heap. Vị chỉ của khối ký ức bản dãy sẽ được ghi trong biến bản dãy được tạo ra trong bước (1), **myIntArray** chẳng hạn. Do đó, bản dãy thuộc kiểu dữ liệu qui chiếu.

(3) Cuối cùng, bạn phải khởi gán các phần tử bản dãy, sử dụng đến initializer (một danh sách các dữ liệu phân cách bởi dấu phẩy được đóng khung trong cặp dấu ngoặc nghéo {}) để điền dữ liệu vào các phần tử bản dãy. Tuy nhiên, bước này có thể không thực hiện. Trong trường hợp này, các phần tử bản dãy sẽ được cho về trị mặc nhiên, tùy theo kiểu dữ liệu của bản dãy. Zero đối với kiểu dữ liệu số nguyên chẳng hạn hoặc null đối với phần tử bản dãy là kiểu object.

Một khi thể hiện bản dãy (array instance) đã được tạo ra, số chiều (rank) và kích thước của mỗi chiều được xem như đã được thiết lập và được cho cố định suốt cuộc đời của thể hiện bản dãy, nghĩa là bạn không thể thay đổi số chiều cũng như thay đổi kích thước bản dãy, một khi "ván đã đóng thuyền".

10.1.1 Bản dãy một chiều

bạn khai báo một bản dãy một chiều C# theo cú pháp sau đây:

*type***[]** *array-name***;**

Cặp dấu [] phải được đặt nằm sau kiểu dữ liệu *type*, báo cho trình biên dịch biết là một bản dãy. Chỉ một cặp dấu [] cho bản dãy một chiều. Sau cặp [] là tên "cúng cơm" bản dãy *array-name*. Thí dụ:

```
int[] myIntArray;       // bản dãy một chiều kiểu số nguyên.
string[] myStrArray;    // bản dãy một chiều kiểu chuỗi chữ.
Object[] myObjArray;    // bản dãy một chiều kiểu object
```

Tiếp theo, bạn thể hiện một đối tượng bản dãy sử dụng từ chốt **new**. Thí dụ:

```
myIntArray = new int[5];      // myIntArray gồm 5 số nguyên.
myStrArray = new string[4];   // myStrArray gồm 4 phần tử chuỗi chữ.
myObjArray = new Object[3];   // myObjArray gồm 3 phần tử object
```

Trong các thí dụ trên, bạn khai báo kiểu dữ liệu và kích thước bản dãy dựa trên hai câu lệnh riêng rẽ. Tuy nhiên, bạn cũng có thể viết một cách ngắn gọn hơn thành một câu lệnh như sau:

```
int[] myIntArray = new int[5];   // vừa khai báo vừa tạo một bản dãy
```

Từ chốt **new** cho phép bạn tạo một bản dãy kích thước cố định lúc ban đầu, đồng thời tự động khởi gán những trị mặc nhiên cho các phần tử bản dãy. Thí dụ, các phần tử bản dãy của ***myIntArray*** sẽ được khởi gán về zero. Ngoài ra, bạn nên nhớ kích thước bản dãy được thiết lập bởi **new**, chứ không bởi khi khai báo. Do đó, nếu bạn chọn khai báo một bản dãy với kích thước cố định lúc ban đầu, thì bạn phải sử dụng từ chốt **new**. Tuy nhiên, bạn cũng có thể để cho trình biên dịch ấn định kích thước bản dãy, bằng cách sử dụng cách ghi ngắn gọn như sau:

```
// Kích thước bản dãy myAges sẽ tự động cho về 4.
// Bạn để ý không có từ chốt new và cặp dấu [] trống.
int [] myAges = {20, 22, 23, 0};
```

Sau khi đã khai báo và tạo một bản dãy, bạn có thể khởi gán các phần tử bản dãy bằng cách sử dụng cặp dấu {} (curly bracket), thay vì gán trị cho từng phần tử bản dãy một. Trong cặp dấu {} là danh sách các dữ liệu được phân cách bởi dấu phẩy. Phần này được gọi là initializer. Do đó, hai cách khởi gán như sau đều giống nhau. Thí dụ:

```
int[] myIntArray = new int[5] {1, 3, 5, 7, 9};// Số phần tử bản
                                                //dãy được kê ra
int[] myIntArray = new int[] {1, 3, 5, 7, 9}; // Số phần tử bản
                                                // dãy không kê ra
```

hoặc

```
// khởi gán mỗi phần tử bản dãy vào lúc khai báo
string[] weekDays = new string[]
   {"Sun", "Sat", "Mon", "Tue", "Wed", "Thu", "Fri"};

// hoặc gán trị cho từng phần tử bản dãy một:
string[] weekDays = new string[7];
weekDays[0] = "Sun";
weekDays[1] = "Sat";
weekDays[2] = "Mon";
weekDays[3] = "Tue";
weekDays[4] = "Wed";
weekDays[5] = "Thu";
weekDays[6] = "Fri";
```

Khi khởi gán một bản dãy vào lúc khai báo, bạn cũng có thể sử dụng cách viết ngắn gọn như sau:

```
int[] myIntArray = {1, 3, 5, 7, 9};
string[] weekDays = {"Sun", "Sat", "Mon", "Tue",
          "Wed", "Thu", "Fri"};
Object[] myObjArray = {26, 27, 28};
```

Ngoài ra, bạn cũng có thể khai báo một biến bản dãy mà không cần khởi gán, nhưng bạn phải dùng đến từ chốt **new** khi gán một bản dãy cho biến này. Thí dụ:

```
int[] myIntArray;
myIntArray = new int[] {1, 3, 5, 7, 9}; // OK
myIntArray = {1, 3, 5, 7, 9}; // Sai
```

10.1.1.1 Tìm hiểu trị mặc nhiên

Điều quan trọng bạn phải phân biệt giữa bản thân bản dãy (là tập hợp những phần tử bản dãy) và các phần tử bản dãy. **myIntArray** là một bản dãy, còn phần tử bản dãy là 5 con số nguyên mà bản dãy nắm giữ. Như đã nói, bản dãy C# là kiểu dữ liệu qui chiếu nên thường được cấp phát ký ức trên vùng heap. Do đó, **myIntArray** được cấp phát trên

heap. Các phần tử bản dãy được cấp phát ký ức dựa trên kiểu dữ liệu. int của **myIntArray** thuộc kiểu trị, do đó các phần tử bản dãy của **myIntArray** là thuộc kiểu trị. Còn một bản dãy các phần tử kiểu qui chiếu sẽ chỉ chứa toàn những qui chiếu về các phần tử bản dãy, và những phần tử bản dãy này cũng sẽ được cấp phát ký ức trên heap.

Khi bạn tạo một bản dãy kiểu trị, thì mỗi phần tử bản dãy khởi đi chứa trị mặc nhiên thuộc kiểu dữ liệu được trữ trên bản dãy. Thí dụ, việc thể hiện đối tượng:

```
myIntArray = new int[5];
```

sẽ tạo một bản dãy gồm 5 số nguyên, được cho về zero, là trị mặc nhiên của kiểu dữ liệu số nguyên.

Còn một bản dãy kiểu qui chiếu thì lại không được khởi gán theo trị mặc nhiên, mà lại được khởi gán theo null reference. Do đó, nếu bạn muốn truy xuất một phần tử thuộc bản dãy kiểu qui chiếu trước khi chưa đặc biệt khởi gán phần tử này, bạn sẽ gây ra một biệt lệ sai.

Giả sử, bạn đã tạo ra một lớp **Button**. Bạn khai báo một bản dãy các đối tượng Button như theo câu lệnh sau đây:

```
Button[] myButtonArray;
```

và bạn thể hiện đối tượng bản dãy myButtonArray như sau:

```
myButtonArray = new Button[3];
```

Khác với thí dụ **myIntArray**, 2 câu lệnh trên sẽ không tạo một bản dãy với những qui chiếu về 3 đối tượng **Button**. Thay vào đó, sẽ tạo bản dãy **myButtonArray** với 3 null reference. Muốn sử dụng **myButtonArray**, trước tiên bạn phải tạo và gán những đối tượng **Button** cho mỗi qui chiếu trên **myButtonArray**. Bạn sẽ sử dụng một vòng lặp tạo đối tượng **Button** thêm từng đối tượng một vào **myButtonArray**.

10.1.1.2 Truy xuất các phần tử bản dãy thế nào?

bạn truy xuất một phần tử bản dãy bằng cách sử dụng tác tử chỉ mục (index operator, []). Vì bản dãy được đánh số theo zero-based, nên phần tử bản dãy thứ nhất bao giờ cũng mang chỉ số zero, trong trường hợp này**, myIntArray[0].** Những phần tử bản dãy **myIntArray** kế tiếp sẽ tuần tự được đánh số **myIntArray[1]…myIntArray[4].**

Như đã nói trên, bản dãy là những đối tượng, do đó mang những thuộc tính. **Length** là một trong những thuộc tính rất hữu ích. **Length** cho biết kích thước bản dãy, nghĩa là

tổng cộng có bao nhiêu phần tử trong bản dãy. Như vậy, các phần tử bản dãy có thể được đánh số từ 0 đến Length-1.

Thí dụ 10-1 sau đây, minh hoạ các khái niệm về bản dãy mà chúng ta đã đề cập qua. Trong thí dụ này, một lớp Tester sẽ tạo ra một bản dãy **Employees** (nhân viên), đưa dữ liệu vào các phần tử bản dãy **Employees**, rồi sau đó cho in ra trị các phần tử bản dãy này.

Thí dụ 10-1: Làm việc với một bản dãy

```
namespace Prog_CSharp
{
   using System;

   // một lớp đơn giản để trữ lên bản dãy
   public class Employee
   {
      public Employee(int empID) {this.empID = empID;}
      public override string ToString() {return empID.ToString();}
      private int empID;
      private int size;
   }

   public class Tester
   {
      static void Main()
      {  int[] intArray;
         Employee[] empArray;
         intArray = new int[5]; // 5 phần tử
         empArray = new Employee[3];   // 3 phần tử

         // Cho điền dữ liệu
         for (int i = 0; i <empArray.Length; i++)
         {  empArray[i] = new Employee(i+5);
         }

         for (int i=0; i<intArray.Length; i++)
         {  Console.WriteLine(intArray[i].ToString());
         }

         for (int i = 0; i <empArray.Length; i++)
         {  Console.WriteLine(empArray[i].ToString());
         }
      }
   }
}
```

Kết xuất:

```
0
0
0
0
0
5
6
7
```

Thí dụ trên bắt đầu định nghĩa một lớp **Employee** lo thiết đặt một hàm khởi dựng chỉ dùng đến một thông số int, **empID.** Hàm hành sự **ToString()**, được kế thừa từ **Object** được phủ quyết để in ra trị mã số nhân viên empID.

Hàm Main của lớp Tester khai báo rồi thể hiện một cặp bản dãy **intArray** và **empArray**. **intArray** được tự động khởi gán bởi zero, còn nội dung các phần tử bản dãy **empArray** phải được điền bằng tay.

Cuối cùng, nội dung các bản dãy được in ra, bảo đảm chúng được điền như theo ý muốn. Trước tiên, 5 số nguyên in ra trị của chúng, theo sau là nội dung của 3 đối tượng **Employee**.

10.1.1.3 Câu lệnh foreach

Câu lệnh vòng lặp **foreach** khá mới mẻ đối với dòng họ ngôn ngữ C/C++, nhưng lại quá quen thuộc đối với lập trình viên Visual Basic. **foreach** cho phép bạn rảo qua (iterate) tất cả các phần tử bản dãy hoặc các tập hợp (collection) khác, và tuần tự xem xét từng phần tử một. Cú pháp câu lệnh **foreach** như sau:

foreach (*type identifier* **in** *expression*) *statement*

với *type* là kiểu dữ liệu của *identifier*, một biến tượng trưng cho phần tử collection; còn *expression* là một đối tượng collection hoặc là một bản dãy; cuối cùng *statement* là những câu lệnh phải thi hành.

Thí dụ 10-1 có thể được thay thế sử dụng câu lệnh **foreach** thay vì **for**:

Thí dụ 10-2: Sử dụng foreach

```
namespace Prog_CSharp
{
   using System;

   // một lớp đơn giản để trữ lên bản dãy
```

```
    public class Employee
    {   public Employee(int empID) {this.empID = empID;}
        public override string ToString() {return empID.ToString();}
        private int empID;
        private int size;
    }

    public class Tester
    {
        static void Main()
        {   int[] intArray;
            Employee[] empArray;
            intArray = new int[5];
            empArray = new Employee[3];
            // Cho điền dữ liệu
            for (int i = 0; i <empArray.Length; i++)
            {   empArray[i] = new Employee(i+10);
            }

            foreach (int i in intArray)
            {   Console.WriteLine(i.ToString());
            }

            foreach (Employee e in empArray)
            {   Console.WriteLine(e.ToString());
            }
        }
    }
}
```

Kết xuất thí dụ 10-2 cũng tương tự như với thí dụ 10-1. Tuy nhiên, thay vì tạo một câu lệnh for đo kích thước bản dãy (thông qua **Length**) và sử dụng một biến đếm **i** như là một chỉ mục bản dãy:

```
    for (int i = 0; i < empArray.Length; i++)
    {
        Console.WriteLine(empArray[i].ToString());
    }
```

ta bây giờ rảo qua bản dãy sử dụng vòng lặp **foreach** tự động trích phần tử bản dãy kế tiếp và tạm thời gán nó cho một đối tượng Employee, ở đây là **e**.

```
    foreach (Employee e in empArray)
    {
        Console.WriteLine(e.ToString());
    }
```

Đối tượng được trích từ bản dãy đều là kiểu dữ liệu thích ứng, do đó bạn có thể triệu gọi bất cứ hàm hành sự nào đối với đối tượng này

10.1.1.4 Trao bản dãy như là thông số

Bạn có thể trao một bản dãy đã được khởi gán cho một hàm hành sự như là một thông số. Thí dụ:

```
PrintArray(myIntArray);
```

Bạn cũng có thể khởi gán rồi trao qua một bản dãy mới trong một bước. Thí dụ:

```
PrintArray(new int[] {1,3,5,7,9});   // tạo, khởi gán bản dãy
                                     // rồi trao cho hàm
```

Thí dụ 10-3 sau đây khởi gán một bản dãy kiểu string rồi trao qua cho hàm hành sự PrintArray như là một thông số, sau đó các phần tử bản dãy được hiển thị:

Thí dụ 10-3: Sử dụng bản dãy như là thông số hàm

```
namespace Prog_CSharp
{
   using System;

   public class Tester
   {
      static void PrintArray(string[] myStrArray)
      {  for (int i = 0; i< myStrArray.Length; i++)
            Console.Write(myStrArray[i] + "{0}",
               i < myStrArray.Length - 1 ? " ": "");
         Console.WriteLine();
      }

      public static void Main()
      {  // khai báo và khởi gán một bản dãy
         string[] conGiap = new string[] {"Tí","Sửu","Dần","Mão", +
            "Thìn","Tỵ","Ngọ","Mùi","Thân","Dậu","Tuất","Hợi"};

         // Trao bản dãy như là thông số
         PrintArray(conGiap);
      }
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Tí Sửu Dần Mão Thìn Tỵ Ngọ Mùi Thân Dậu Tuất Hợi

10.1.1.5 Thông số hàm hành sự và các từ chốt params, ref và out

Khi bạn chuyển một thông số cho một hàm hành sự, ta gọi đây là một thông số hàm hành sự (method parameter). Thông số này có thể đi kèm theo hoặc không các từ chốt **params**, **ref** hoặc **out**, nằm trước thông số.

Nếu thông số không đi kèm theo với các từ chốt **ref** hoặc **out**, thì thông số có thể có một trị được gắn liền với nó. Đây là một bản sao trị đối mục trên hàm triệu gọi (caller). Trị này có thể bị thay đổi trong hàm hành sự. Tuy nhiên, trị bị thay đổi này sẽ không bị giữ lại khi quyền điều khiển được trả về hàm triệu gọi. Ta gọi cách chuyển thông số này là By Value. Bằng cách sử dụng các từ chốt **ref** hoặc **out**, bạn có thể thay đổi tình trạng này.

Ta sẽ lần lượt xem qua các thông số hàm hành sự **params**, **ref** và **out**.

Từ chốt **params** cho phép bạn khai báo một thông số hàm hành sự chấp nhận một đối mục theo đấy số lượng đối mục thường là không cố định. Trong khai báo hàm hành sự chỉ một từ chốt **params** mà thôi. Sau từ chốt **params** sẽ không thêm thông số nào khác.

Từ chốt **ref** (viết tắt chữ reference) hoặc **out** cho phép hàm hành sự qui chiếu về cùng biến được trao qua cho hàm hành sự. Bất cứ mọi thay đổi được thực hiện trên thông số bởi hàm sẽ được phản ảnh lên biến khi quyền điều khiển được trả về cho hàm triệu gọi. Ta gọi cách trao thông số này là By Reference.

Với từ chốt **ref,** đối mục trước tiên phải được khởi gán, còn với từ chốt **out** thì khỏi phải khởi gán trước khi trao cho hàm hành sự.

Bạn sử dụng **out** khi bạn muốn một hàm hành sự trả về nhiều trị. Một hàm hành sự sử dụng thông số **out** có thể vẫn trả về một trị. Một hàm hành sự có thể có nhiều thông số **out**.

Một thuộc tính (property) không phải là một biến, nên không thể trao qua hàm hành sự như là một thông số **ref** hoặc thông số **out**.

10.1.1.6 Chuyển bản dãy sử dụng từ chốt params

bạn có thể tạo một hàm hành sự cho phép hiển thị lên console bất cứ bao nhiêu số nguyên cũng được bằng cách chuyển vào một bản dãy số nguyên rồi sau đó rảo qua bản dãy sử dụng câu lệnh **foreach**. Từ chốt **params** cho phép bạn đưa vào một số không cố

định thông số mà không nhất thiết phải tạo rõ ra, một cách tường minh (explicitly), một bản dãy.

Trong thí dụ sau đây, 10-4, bạn tạo ra một hàm hành sự **DisplayVals()**, chấp nhận một số thông số thay đổi, kiểu số nguyên:

```
public void DisplayVals(params int[] intVals)
```

Bản thân hàm hành sự xem bản dãy như là một bản dãy số nguyên được tạo rõ ra và được trao qua như là một thông số. Tiếp theo, bạn tha hồ rảo qua bản dãy, sử dụng câu lệnh vòng lặp foreach:

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

Tuy nhiên, *về phía hàm triệu gọi, bạn khỏi phải cần tạo rõ ra một bản dãy.* Chỉ cần trao qua những thông số số nguyên, và trình biên dịch tự động ráp các thông số này thành một bản dãy dành cho **DisplayVals()** sử dụng:

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

Nếu bạn muốn, bạn hoàn toàn tự do đưa vào một cách tường minh (explicitly) một bản dãy như sau:

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

Sau đây là đoạn mã minh hoạ việc sử dụng từ chốt **params**.

Thí dụ 10-4: Sử dụng từ chốt params

```
namespace Prog_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 xuất:

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

10.1.1.7 Chuyển bản dãy sử dụng từ chốt ref và out

Giống như với tất cả các thông số **out**, một thông số **out** kiểu bản dãy phải được gán trị trước khi đem sử dụng, nghĩa là phải được gán bởi hàm bị triệu gọi (callee). Thí dụ:

```
public static void myMethod(out int[] myArray)
{
    myArray = new int[10]; // gán myArray
}
```

Giống như tất cả các thông số **ref**, một thông số **ref** kiểu bản dãy phải được gán bởi hàm triệu gọi (caller), nghĩa là phía hàm bị gọi khỏi phải gán trị. Một thông số **ref** kiểu bản dãy có thể bị thay đổi như là kết quả của việc triệu gọi. Thí dụ, bản dãy có thể bị gán về trị **null** hoặc có thể được khởi gán về một bản dãy khác.

```
public static void myMethod(ref int[] myArray)
{
    myArray = new int[10]; // myArray được khởi gán
                           // về một bản dãy khác
}
```

Sau đây hai thí dụ minh hoạ khác biệt giữa **ref** và **out** khi được sử dụng như là thông số để trao các bản dãy cho các hàm hành sự

Trên thí dụ 10-5, bản dãy **myIntArray** được khai báo trong Main, là hàm triệu gọi (caller), và được khởi gán trong hàm hành sự **FillArray**. Sau đó, các phần tử bản dãy được trả về cho hàm triệu gọi để được hiển thị.

Thí dụ 10-5: Sử dụng từ chốt out trên thông số bản dãy

```
namespace Prog_CSharp
{
   using System;

   public class Tester
   {
      public static void FillArray(out int[] myIntArray)
      {  // khởi gán bản dãy nơi hàm bị triệu gọi
         myIntArray = new int[5] {1,2,3,4,5};
      }

      public static void Main()
      {
         int[] myIntArray;   // không cần thiết khởi gán
                             // bản dãy tại hàm caller
         FillArray(out myIntArray);// trao bản dãy cho callee
                                   // sử dụng out

         // Cho hiển thị các phần tử bản dãy:
         Console.WriteLine("Phần tử bản dãy:");
         for (int i = 0; i < myIntArray.Length; i++)
            Console.WriteLine(myIntArray[i]);
      }
   }
}
```

Kết xuất

```
Phần tử bản dãy:
1
2
3
4
5
```

Thí dụ 10-6 sau đây cho thấy bản dãy **myIntArray** được khởi gán trong hàm triệu gọi Main(), rồi được trao cho hàm hành sự **FillArray** sử dụng thông số **ref**. Một số phần tử bản dãy sẽ được nhật tu trong **FillArray**. Sau đó, các phần tử bản dãy được trả về cho hàm triệu gọi để cho hiển thị.

Thí dụ 10-6: Sử dụng từ chốt ref trên thông số bản dãy

```
namespace Prog_CSharp
{
   using System;

   public class Tester
   {
      public static void FillArray(ref int[] arr)
      {  // tạo bản dãy theo yêu cầu
         if (arr == null)
            arr = new int[10];
         // bằng không, cho điền dữ liệu lên bản dãy
         arr[0] = 123;
         arr[4] = 1024;
      }

      public static void Main()
      {  int[] myIntArray = {1,2,3,4,5};  // khởi gán bản dãy
         FillArray(ref myIntArray);       // Trao qua hàm callee sử
                                          // dụng ref
         // cho hiển thị nội dung bản dãy đã được nhật tu
         Console.WriteLine("Phần tử bản dãy:");
         for (int i = 0; i < myIntArray.Length; i++)
            Console.WriteLine(myIntArray[i]);
      }
   }
{
```

Kết xuất

```
Phần tử bản dãy:
123
2
3
4
1024
```

10.1.2 Bản dãy nhiều chiều

Bản dãy có thể có nhiều hơn một chiều. Chắc bạn đã biết bản dãy cổ điển hai chiều gồm nhiều hàng (row) và nhiều cột (column). C# hỗ trợ hai loại bản dãy nhiều chiều: bản dãy chữ nhật (rectangular array) và bản dãy "lởm chởm" (jagged array), một loại bản dãy những bản dãy (array of arrays). Trong bản dãy chữ nhật, các hàng đều cùng chiều dài, còn bản dãy "lởm chởm" mỗi hàng sẽ mang kích thước khác nhau.

Số chiều được gọi là rank.

10.1.2.1 Bản dãy hình chữ nhật

Bản dãy chữ nhật là một bản dãy gồm hai (hoặc nhiều) chiều. Theo bản dãy qui ước chiều thứ nhất cho biết số hàng, còn chiều thứ hai cho biết số cột. Muốn khai báo một bản dãy hai chiều, bạn sử dụng cú pháp sau đây:

type **[,]** *array-name*

Chỉ một cặp dấu [], nhưng trong ấy có một dấu phẩy. Thí dụ, muốn khai báo một bản dãy hai chiều **myRectArray** gồm 2 hàng 3 cột với phần tử bản dãy kiểu số nguyên, bạn viết:

```
int [,] myRectArray = new int[2,3];  // bản dãy 2 hàng 3 cột
```

Ngoài ra, bạn cũng có thể tạo một bản dãy 3 chiều, bằng cách viết như sau, với một cặp dấu [] có hai dấu phẩy trong ấy:

```
int[,,] my3DArray = new int[4, 2, 3];
```

10.1.2.1.1 Khởi gán bản dãy

Bạn có thể khởi gán bản dãy sau khi đã khai báo và thể hiện xong bản dãy 2 chiều như sau:

```
int[,] myRectArray = new int [,] {{1,2}, {3,4},
                                  {5,6}, {7,8}};// bản dãy 4 hàng 2 cột
```

Bạn cũng có thể khởi gán bản dãy không cần đến rank, như sau:

```
// bản dãy 4 hàng 2 cột
int[,] myRectArray = {{1,2}, {3,4}, {5,6}, {7,8}};
```

Nếu bạn chọn khai báo một biến bản dãy không cần khởi gán, bạn phải dùng từ chốt **new** để gán bản dãy cho biến. Thí dụ:

```
int[,] myRectArray; // khai báo nhưng chưa khởi gán (giữ chỗ trước)
myRectArray = new int [,] {{1, 2}, {3, 4}, {5, 6}, {7, 8}};// OK
myRectArray = {{1,2}, {3,4}, {5,6}, {7,8}};    // Sai rồi.
```

Bạn cũng có thể gán một trị cho một phần tử bản dãy. Thí dụ:

```
myRectArray[2, 1] = 25; // phần tử hàng 3, cột 2
```

Thí dụ 10-7 sau đây: khai báo, thể hiện và in ra nôi dung một bản dãy hai chiều myRectArray. Trong thí dụ này vòng lặp for được sử dụng để khởi gán các phần tử bản dãy.

Thí dụ 10-7: Bản dãy chữ nhật

```
namespace Prog_CSharp
{  using System;

   public class Tester
   {  static void Main()
      {  const int rows = 4;        // hằng định nghĩa số hàng
         const int columns = 3;     // hằng định nghĩa số cột

         // khai báo một bản dãy 4 hàng x 3 cột kiểu số nguyên
         int[,] myRectArray = new int[rows, columns];

         // điền các phần tử bản dãy dùng vòng lặp for
         for (int i = 0; i < rows; i++)
         {  for (int j = 0; j < columns; j++)
            {  myRectArray[i,j] = i+j;
            }
         }
         // In ra nội dung bản dãy
          for (int i = 0; i < rows; i++)
         {  for (int j = 0; j < columns; j++)
            {  Console.WriteLine("RectArray[{0},{1}] = {2}",
                                          i, j, myRectArray[i,j]);
            }
         }
      }
   }
}
```

Kết xuất

```
RectArray[0,0] = 0
RectArray[0,1] = 1
RectArray[0,2] = 2
RectArray[1,0] = 1
RectArray[1,1] = 2
RectArray[1,2] = 3
RectArray[2,0] = 2
RectArray[2,1] = 3
RectArray[2,2] = 4
RectArray[3,0] = 3
RectArray[3,1] = 4
RectArray[3,2] = 5
```

Bạn để ý, trong thí dụ trên bạn sử dụng cặp hằng trị (constant value):

```
const int rows = 4;
const int columns = 3;
```

để thiết lập kích thước bản dãy:

```
int[,] myRectArray = new int[rows, columns};
```

Bạn để ý là phải dùng hằng trị, chứ không được dùng biến. Nếu bạn viết các câu lệnh trên như sau, trình biên dịch sẽ "la làng" cho mà coi:

```
// trình biên dịch sẽ từ chối dịch
int rows = 4;
int columns = 3;
int[,] myRectArray = new int[rows, columns];
```

Bạn để ý cú pháp. Cặp dấu ngoặc vuông trong khai báo int[,] cho biết là một bản dãy số nguyên, còn dấu phẩy cho biết bản dãy có hai chiều (hai dấu phẩy cho biết là 3 chiều, v.v..). Việc thể hiện **myRectArray** thông qua **new int[rows, columns]** sẽ thiết đặt kích thước của mỗi chiều. Ở đây, việc khai báo và thể hiện bản dãy được thực hiện trên cùng một câu lệnh. Tiếp theo, hai vòng lặp **for** rảo qua mỗi cột trên mỗi hàng, và dữ liệu được đưa vào từng cột cho đến hết.

Giống như việc khởi gán bản dãy một chiều sử dụng đến cặp dấu {}, mỗi cặp chỉ một hàng. Bạn cũng có thể khởi gán bản dãy hai chiều bằng tay như sau:

```
// bản dãy 4 hàng 3 cột
int[,] myRectArray = { {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11} };
```

Trong câu lệnh trên, bạn có 4 danh sách nằm trong dấu {}, mỗi danh sách gồm 3 phần tử, xem như là bản dãy 4x3. Nếu bạn viết như sau:

```
// bản dãy 3 hàng 4 cột
int[,] myRectArray = { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} };
```

lúc này bản dãy sẽ là 3x4, nghĩa là 3 hàng 4 cột.

10.1.3 Bản dãy "lởm chởm" (Jagged Arrays)

Một bản dãy "lởm chởm" là một bản dãy mà mỗi phần tử lại là một bản dãy, với kích thước lại khác nhau, do đó không mang dáng dấp hình chữ nhật mà là răng cưa.

Khi bạn tạo một bản dãy "lởm chởm", bạn khai báo số hàng trên bản dãy. Mỗi hàng

sẽ chứa một bản dãy một chiều kích thước khác nhau. Những bản dãy con này phải được khai báo từng bản dãy một. Sau đó, bạn có thể điền trị vào các phần tử đối với các bản dãy nằm trong sâu.

Trên bản dãy "lởm chởm", mỗi chiều là một bản dãy một chiều. Muốn khai báo một bản dãy "lởm chởm" bạn sử dụng cú pháp sau đây, bao nhiêu cặp [] là bấy nhiêu chiều bản dãy:

type [] [] …

Thí dụ sau đây khai báo một bản dãy "lởm chởm" hai chiều (nghĩa là hai cặp []) gồm 4 hàng, mỗi hàng là một bản dãy một chiều:

```
int [] [] myJaggedArray = new int[4] [];
```

Bạn có thể truy xuất phần tử thứ 5 của hàng thứ 3 bằng cách viết **myJaggedArray[2] [4]**.

Trước khi bạn có thể sử dụng **myJaggedArray**, các phần tử phải được khởi gán. Bạn có thể khởi gán các phần tử như sau:

```
myJaggedArray[0] = new int[5];    // hàng 1 gồm 5 phần tử
myJaggedArray[1] = new int[2];    // hàng 2 gồm 2 phần tử bản dãy
myJaggedArray[2] = new int[3];    // hàng 3 gồm 3 phần tử bản dãy
myJaggedArray[3] = new int[5];    // hàng 4 gồm 5 phần tử bản dãy
```

Mỗi hàng là một bản dãy một chiều gồm toàn số nguyên. Hàng thứ nhất là một bản dãy 5 số nguyên, hàng thứ hai là một bản dãy 2 số nguyên, v.v.. Bạn cũng có thể sử dụng initializer (cặp dấu {} khởi gán) để điền trị vào các phần tử bản dãy, trong trường hợp này, bạn khỏi phải kê ra kích thước bản dãy. Thí dụ:

```
myJaggedArray[0] = new int[5] {1,3,5,7,9};
myJaggedArray[1] = new int[2] {0,2};
myJaggedArray[2] = new int[3] {4,6,8};
myJaggedArray[3] = new int[5] {10,11,12,13,14};
```

Bạn cũng có thể khởi gán bản dãy theo cách khai báo sau đây:

```
int [] [] myJaggedArray = new int [] []
             {   new int[] {1,3,5,7,9},
                    new int[] {0,2},
                    new int[] {4,6,8},
                    new int[] {10,11,12,13,14}
             };
```

hoặc một cách ngắn gọn hơn như sau:

```
int [] [] myJaggedArray = {
              new int[] {1,3,5,7,9},
              new int[] {0,2},
              new int[] {4,6,8},
              new int[] {10,11,12,13,14}
           };
```

Bạn để ý, trên cách viết ngắn gọn trên, bạn không thể bỏ qua tác tử **new** trong phần khởi gán các hàng bản dãy trong sâu, vì không có khởi gán mặc nhiên đối với những hàng này.

Bạn có thể truy xuất riêng rẽ các phần tử bản dãy như sau:

```
myJaggedArray[0] [1] = 33;// gán 33 cho phần tử thứ 2 bản dãy 1
myJaggedArray[2] [1] = 44;// gán 44 cho phần tử 2 bản dãy 3
```

Bạn để ý, khi bạn truy xuất các phần tử bản dãy chữ nhật, bạn ghi các chỉ số trong lòng cặp dấu [], chẳng hạn **myRectArray[i,j]**, còn với bản dãy "lởm chởm" thì bạn cần đến một cặp dấu [], chẳng hạn **myJaggedArray[3] [i]**.

Ngoài ra, bạn cũng có thể trộn bản dãy "lởm chởm" với bản dãy chữ nhật. Sau đây là khai báo và khởi gán một bản dãy "lởm chởm" một chiều chứa phần tử bản dãy hai chiều kích thước khác nhau:

```
int[] [,] myJaggedRectArray = new int[3] [,]
           {   new int[,] { {1,3}, {5,7} },
                            new int[,] { {0,2}, {4,6}, {8,10} },
                            new int[,] { {11,22}, {99,88}, {0,9} }
                         };
```

Bạn có thể truy xuất từng phần tử bản dãy **myJagRectArray**, cho hiển thị trị của phần tử [1,0] của bản dãy thứ nhất (trị 5):

```
Console.Write("{0}", myJagRectArray[0] [1,0]);
```

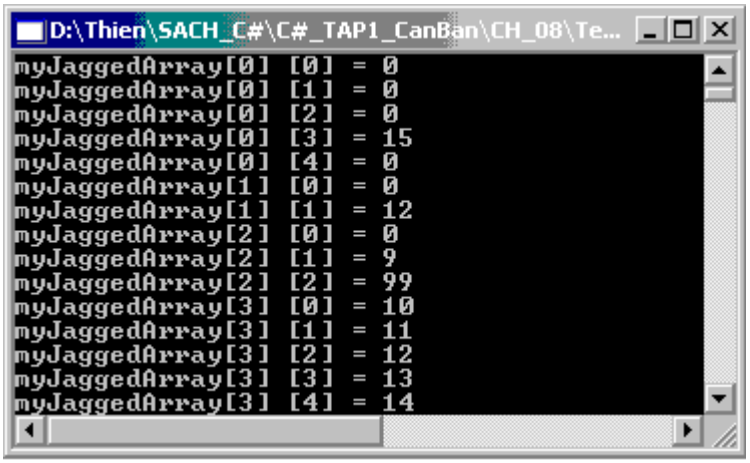
Thí dụ 10-8 sau đây tạo một bản dãy jagged **myJaggedArray**, khởi gán các phần tử bản dãy rồi sau đó in ra nội dung bản dãy. Ta tạm lợi dụng việc các số nguyên được tự động cho về zero, nên ta chỉ khởi gán một vài phần tử bản dãy mà thôi:

Thí dụ 10-8: Bản dãy "lởm chởm"

```
namespace Prog_CSharp
{  using System;
   public class Tester
```

```
   {  static void Main()
      {  const int rows = 4;     // bản dãy sẽ cao 4 hàng
         int [] [] myJaggedArray = new int[rows] [];
         myJaggedArray[0] = new int[5];// hàng 1 gồm 5 phần tử
         myJaggedArray[1] = new int[2];// hàng 2 gồm 2 phần tử bản dãy
         myJaggedArray[2] = new int[3];// hàng 3 gồm 3 phần tử bản dãy
         myJaggedArray[3] = new int[5];// hàng 4 gồm 5 phần tử bản dãy
         // điền dữ liệu vào một số phần tử bản dãy
         myJaggedArray[0] [3] = 15;
         myJaggedArray[1] [1] = 12;
         myJaggedArray[2] [1] = 9;
         myJaggedArray[2] [2] = 99;
         myJaggedArray[3] [0] = 10;
         myJaggedArray[3] [1] = 11;
         myJaggedArray[3] [2] = 12;
         myJaggedArray[3] [3] = 13;
         myJaggedArray[3] [4] = 14;

         for (int i = 0; i < 5; i++)
         {  Console.WriteLine("myJaggedArray[0] [{0}] = {1}",
               i, myJaggedArray[0] [i]);
         }

         for (int i = 0; i < 2; i++)
         {  Console.WriteLine("myJaggedArray[1] [{0}] = {1}",
               i, myJaggedArray[1] [i]);
         }

         for (int i = 0; i < 3; i++)
         {  Console.WriteLine("myJaggedArray[2] [{0}] = {1}",
               i, myJaggedArray[2] [i]);
         }

         for (int i = 0; i < 5; i++)
         {  Console.WriteLine("myJaggedArray[3] [{0}] = {1}",
               i, myJaggedArray[3] [i]);
         }
      }
   }
}
```

Kết xuất



10.1.4 Chuyển đổi giữa các bản dãy

Ta có thể tiến hành việc chuyển đổi (conversion, hoặc đảo chuyển) giữa các bản dãy với điều kiện chiều kích (dimension) bằng nhau và liệu xem việc chuyển đổi giữa kiểu dữ liệu các phần tử có thể được hay không. Một chuyển đổi hiểu ngầm có thể xảy ra nếu các phần tử bản dãy có thể được chuyển đổi một cách hiểu ngầm cũng như có thể chuyển đổi tường minh nếu các phần tử có thể được chuyển đổi một cách tường minh.

Nếu một bản dãy có chứa những qui chiếu về các đối tượng qui chiếu, ta có thể chuyển đổi đối với một bản dãy của những phải cơ bản. Thí dụ 10-9 sau đây minh hoạ việc chuyển đổi một bản dãy kiểu dữ liệu tự tạo **Button** thành một bản dãy các đối tượng**.**

Thí dụ 10-9: Chuyển đổi giữa các bản dãy

```
namespace Prog_CSharp
{   using System;

    // Tạo một object mà ta có thể trữ trên bản dãy
    public class Employee      // lớp Nhân viên
    {   public Employee(int empID)    // hàm constructor
        {   this.empID = empID;
        }

        public override string ToString()
        {   return empID.ToString();
        }
        private int empID;
    }

    public class Tester
```

```
   {   // Hàm PrintArray này lấy một bản dãy các đối tượng mà ta trao qua
       // một bản dãy Employee, rồi một bản dãy chuỗi chữ. Việc chuyển
       // đổi sẽ là hiểu ngầm vì cả hai Employee và chuỗi chữ đều được
       // dẫn xuất từ Object.
       public static void PrintArray(object [] theArray)
       {   Console.WriteLine("Nội dung bản dãy {0}", theArray.ToString());

           // rảo qua bản dãy và in ra trị
           foreach(object obj in theArray)
           {   Console.WriteLine("Trị: {0}", obj);
           }
       }

       static void Main()
       {   //Hình thành bản dãy đối tượng Employee và khởi gán rồi in ra
           Employee[] myEmployeeArray = new Employee[3];// bản dãy gồm 3
                                                        // người
           for (int i = 0; i < 3; i++)   // khởi gán trị nhân viên
           {   myEmployeeArray[i] = new Employee(i+5);
           }
           PrintArray(myEmployeeArray);

           // Tạo một bản dãy với 2 chuỗi chữ rồi in ra
           string[] myArray = { "Hello", "World" };
           PrintArray(myArray);

       }  // end Main
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

```
Nội dung bản dãy Prog_CSharp.Employee[]
Trị: 5
Trị: 6
Trị: 7
Nội dung bản dãy System.String
Trị: Hello
Trị: World
```

Thí dụ 10-9 bắt đầu bằng cách tạo một lớp **Employee** đơn giản. Lớp **Tester** bây giờ chứa một hàm static mới **PrintArray()** tiếp nhận một thông số là một bản dãy một chiều kiểu **Object**.

```
public static void PrintArray(object [] theArray)
```

Object là lớp cơ sở hiểu ngầm đối với tất cả các đối tượng trên .NET Framework, do đó Object là lớp cơ sở hiểu ngầm của cả **String** lẫn **Employee**. Hàm **PrintArray()** thi

hành hai hành động: Trước tiên, nó triệu gọi **ToString** trên bản thân bản dãy:

```
Console.WriteLine("Nội dung bản dãy {0}", theArray.ToString());
```

System.Array (mà chúng ta sẽ xem sau, 10.1.5) phủ quyết **PrintArray()** trong chiều có lợi cho bạn bằng cách in ra một tên nhận diện bản dãy:

```
Nội dung bản dãy Prog_CSharp.Employee[]
Nội dung bản dãy System.String
```

Sau đó **PrintArray()** tiếp tục triệu gọi **ToString()** trên mỗi phần tử bản dãy mà nó nhận vào như là thông số. Vì **ToString()** là một hàm virtual trong lớp cơ sở **Object**, nên nó bao giờ cũng có sẵn trên mọi lớp được dẫn xuất. Bạn đã phủ quyết hàm này một cách thích ứng trên **Employee**, do đó chạy tốt. Triệu gọi **ToString** trên một đối tượng **String** thật ra không cần thiết, vô thưởng vô phạt, nhưng nó cho phép bạn xử lý những đối tượng này một cách đa hình.

10.1.5 Lớp cơ sở System.Array

bạn nhớ cho là mỗi bản dãy bạn tạo ra đều tự động được dẫn xuất từ lớp **System.Array**. Lớp cơ sở này có một số hàm hành sự và thuộc tính rất hữu ích giúp bạn thao tác trên các bản dãy một cách thoải mái. Bảng 10-1, đã liệt kê các hàm hành sự và thuộc tính của lớp **System.Array**.

Bạn đã biết qua thuộc tính **Length** trong các thí dụ đi trước. Thí dụ, muốn biết kích thước bản dãy một chiều **myIntArray**, bạn viết:

```
int ArrayLength = myIntArray.Length;
```

Nếu bản dãy thuộc loại nhiều chiều, ta có thể biết kích thước của bất cứ chiều nào bằng cách sử dụng hàm hành sự **GetLength()**. Thí dụ, đi lấy kích thước chiều thứ 2 của bản dãy 3 chiều **myIntArray**, ta viết:

```
int ArrayLength = myIntArray.GetLength(1);
```

Ngoài ra, bạn có các hàm hành sự **Sort()** để sắp xếp các phần tử bản dãy kiểu dữ liệu "bẩm sinh", hàm hành sự **Reverse()** dùngg đảo ngược thứ tự hiện hành của các phần tử bản dãy, hoặc hàm hành sự **Clear()** cho xoá trằng các phần tử bản dãy. Thí dụ, ta có bản dãy một chiều **myHoaSi**, liệt kê tên các hoạ sĩ nổi tiếng. Ta có thể viết các câu lệnh sau đây:

```
    // Tạo một bản dãy tên các hoạ sĩ danh tiếng
    string[] myHoaSi = { "Leonardo", "Monet", "Bui Huy Phái",
                         "Van Gogh", "Gaughin" };

    // Sắp theo thứ tự abc
    Array.Sort(myHoaSi);

    // Đảo ngược theo thứ tự đi xuống
    Array.Reverse(myHoaSi);

    // In ra nôi dung bản dãy myHoaSi
    for (i = 0; i < myHoaSi.Length; i++)
    {   Console.WriteLine(myHoaSi[i]);
    }

    // Xoá các phần tử bản dãy trừ phần tử chót
    Array.Clear(myHoaSi, 1, (myHoaSi.Length - 1));
```

Chúng tôi không thể đưa ra tất cả những thí dụ sử dụng các hàm hành sự và thuộc tính của **System.Array**. Bạn có thể lên MSDN của Microsoft tìm chỉ mục Array.xxxx, với xxxx là tên hàm hành sự hoặc thuộc tính để xem ý nghĩa của từng thành viên của **System.Array**, với các thí dụ đi kèm theo.

Tạm thời, chúng tôi cho trích từ MSDN một thí dụ ứng dụng hàm hành sự **Array.Copy** lo sao chép các phần tử bản dãy không những cùng kiểu dữ liệu mà còn có thể sao chép giữa các bản dãy chuẩn kiểu dữ liệu khác nhau, hàm tự động xử lý việc "ép vào khuôn" kiểu dữ liệu (type casting), ta gọi là ép kiểu.

Thí dụ 10-10: Sử dụng hàm Array.Copy()

```
public class SamplesArray
{
    public static void Main()
    {
        // Tạo và khởi gán một bản dãy số nguyên mới
        // và một bản dãy Object mới.
    int[] myIntArray = new int[5] { 1, 2, 3, 4, 5 };
    Object[] myObjArray = new Object[5] { 26, 27, 28, 29, 30 };

   // In ra các trị ban đầu của cả hai bản dãy.
    Console.WriteLine("Khởi đi,");
    Console.Write("bản dãy số nguyên:");
    PrintValues(myIntArray);
    Console.Write("bản dãy Object: ");
    PrintValues(myObjArray);

   // Chép hai phần tử đầu tiên từ bản dãy integer về bản dãy Object.
    Array.Copy(myIntArray, myObjArray, 2);
```

```
  // In ra nội dung các bản dãy bị thay đổi.
   Console.WriteLine("\nSau khi chép hai phần tử đầu tiên của bản dãy
              số nguyên về bản dãy Object,");
   Console.Write("bản dãy số nguyên:");
   PrintValues(myIntArray);
   Console.Write("bản dãy Object: ");
   PrintValues(myObjArray);

  // Chép hai phần tử chót từ bản dãy Object lên bản dãy số nguyên.
   Array.Copy(myObjArray, myObjArray.GetUpperBound(0) - 1,
                 myIntArray, myIntArray.GetUpperBound(0) - 1, 2);

  // In ra nội dung các bản dãy bị thay đổi.
   Console.WriteLine("\nSau khi chép hai phần tử chót của bản dãy
              Object về bản dãy số nguyên,");
   Console.Write("bản dãy số nguyên:");
   PrintValues(myIntArray);
   Console.Write("bản dãy Object: ");
   PrintValues(myObjArray);
 }

 public static void PrintValues(Object[] myArr)
 {
   foreach (Object i in myArr)
   {
        Console.Write("\t{0}", i);
   }
   Console.WriteLine();
 }

 public static void PrintValues(int[] myArr)
 {
       foreach (int i in myArr)
    {
         Console.Write("\t{0}", i);
    }
    Console.WriteLine();
 }
}
```

Kết xuất.

Khởi đi,
bản dãy số nguyên: 1 2 3 4 5
bản dãy Object: 26 27 28 29 30

Sau khi chép hai phần tử đầu tiên của bản dãy số nguyên về bản dãy Object,
bản dãy số nguyên: 1 2 3 4 5

bản dãy Object: 1 2 28 29 30

Sau khi chép hai phần tử chót của bản dãy Object về bản dãy số nguyên,
bản dãy số nguyên: 1 2 3 29 30
bản dãy Object: 1 2 28 29 30

10.2 Bộ rảo chỉ mục (indexer)

Đôi lúc, bạn lại muốn truy xuất một tập hợp (collection) nằm trong lòng một lớp xem bản thân lớp như là một bản dãy. Thí dụ, giả sử bạn tạo một ô liệt kê (listbox) mang tên **myListBox**, chứa một danh sách những chuỗi chữ được trữ trên một bản dãy một chiều, và một biến thành viên private mang tên **myStrings**. Thông thường, một ô liệt kê thường có một số hàm hành sự và thuộc tính ngoài bản dãy chuỗi. Tuy nhiên, sẽ rất tiện lợi nếu ta có khả năng truy xuất bản dãy ô liệt kê với một chỉ mục, coi ô liệt kê như là một bản dãy. Thí dụ, một thuộc tính như thế cho phép bạn viết ra những câu lệnh như sau:

```
string chuoiDau = myListBox[0];
string chuoiCuoi = myListBox[Length-1];
```

Nói tóm lại, *bộ rảo chỉ mục* (indexer) là một "sáng tạo" của C# cho phép bạn truy xuất những tập hợp nằm trong lòng một lớp, sử dụng cú pháp [] thông dụng của bản dãy. *Indexer được xem như là một thuộc tính khá đặc biệt* kèm theo những hàm hành sự **get()** và **set()**. Khai báo một indexer cho phép bạn tạo những lớp (hoặc struct) hoạt động tương tự những "bản dãy ảo" (virtual array). Những thể hiện của lớp này có thể được truy xuất sử dụng tác tử truy xuất bản dãy **[]**. Đối với những lớp có mang chức năng giống như bản dãy hoặc tập hợp, sử dụng một indexer cho phép người sử dụng lớp này dùng cú pháp bản dãy để truy xuất lớp.

10.2.1 Khai báo Indexer thế nào?

Bạn khai báo một thuộc tính indexer trong lòng một lớp, sử dụng cú pháp sau đây:

type **this** [*type argument*] {**get**; **set**;}

Kiểu dữ liệu *type* được trả về xác định kiểu đối tượng mà indexer sẽ trả về, trong khi *type argument* cho biết loại đối mục nào sẽ được dùng đến để chỉ mục trên tập hợp chứa các đối tượng đích. Mặc dầu việc sử dụng số nguyên làm trị chỉ mục là phổ biến, bạn cũng có thể chỉ mục một tập hợp dựa trên một kiểu dữ liệu khác kể cả kiểu string. Bạn cũng có thể tạo một indexer với nhiều thông số để tạo một bản dãy đa chiều.

Từ chốt **this** là một qui chiếu về đối tượng theo đấy indexer sẽ xuất hiện. Giống như với thuộc tính thông thường, bạn cũng phải cho định nghĩa các hàm hành sự **get()** và **set()** cho biết các đối tượng được nhắm tới sẽ được truy xuất hoặc được gán thế nào.

Phần thân hàm **get()** của indexer cũng giống phần thân của một hàm hành sự thông thường, nghĩa là trả về kiểu của indexer, đồng thời dùng cùng danh sách các thông số hình thức (formal parameter list) như trên indexer. Thí dụ:

```
get
{
   return myArray[index];
}
```

Phần thân hàm **set()** của indexer cũng giống phần thân một hàm hành sự thông thường, đồng thời dùng cùng danh sách các thông số như trên indexer, với việc bổ sung thông số hiểu ngầm (implicit) **value**. Thí dụ:

```
set
{
   myArray[index] = value;
}
```

Số lượng thông số hình thức (formal parameter) cũng như kiểu dữ liệu của các thông số này trên một indexer được gọi là một dấu ấn (signature) của indexer. Nó không bao gồm kiểu dữ liệu của indexer cũng như tên của các thông số hình thức. Nếu bạn khai báo nhiều indexer trên cùng một lớp, thì các indexer này phải mang dấu ấn khác nhau.

Trị một indexer không được xem như là một biến. Do đó, không thể trao một trị indexer như là một thông số **ref** hoặc **out**.

Thí dụ 10-11 sau đây khai báo một ô liệt kê **ListBoxTest**, chứa một bản dãy đơn giản, **myStrings** và một indexer đơn giản lo truy xuất nội dung bản dãy này. Chúng tôi đã đơn giản hoá thí dụ, chỉ quan tâm đến danh sách các chuỗi mà ô liệt kê duy trì và các hàm hành sự thao tác trên các chuỗi này.

Thí dụ 10-11: Sử dụng một indexer đơn giản

```
namespace Prog_CSharp
{
   using System;

   // Một ListBox đã được đơn giản hoá
   public class ListBoxTest
   {  // khởi gán ô liệt kê với các chuỗi chữ trên hàm constructor
      public ListBoxTest(params string[] initialStrings)
```

```
    {   myStrings = new String[256];  // cấp phát ký ức các chuỗi chữ

        // sao các chuỗi chữ được trao qua cho hàm constructor
        foreach (string s in initialStrings)
        {   myStrings[myCtr++] = s;
        }
    }

    // thêm một chuỗi chữ đơn vào cuối ô liệt kê
    public void Add(string theString)
    {   if (myCtr >= myStrings.Length)
        {   // xử lý chỉ mục sai ở đây
        }
        else
            myStrings[myCtr++] = theString;
    }

    // định nghĩa một indexer cho phép truy xuất giống như bản dãy
    public string this[int index] // int index là danh sách
                                  // thông số hình thức
    {   get
        {   if (index < 0 || index > myStrings.Length)
            {   // xử lý chỉ mục sai
            }
            return myStrings[index];
        }
        set
        {   // chỉ thêm thông qua hàm Add()
            if (index >= myCtr)
            {   // xử lý sai lầm
            }
            else
                myStrings[index] = value;
        }
    }

    // Loan báo cho biết cầm giữ bao nhiêu chuỗi chữ
    public int GetNumEntries()
    {   return myCtr;
    }

    // các biến của lớp ListBoxTest
    private string[] myStrings;
    private int myCtr = 0;
}

public class Tester
{
    static void Main()
    {   // tạo một ô liệt kê mới và cho khởi gán bốn
        // chuỗi chữ như là thông số
        ListBoxTest lbt = new ListBoxTest("Xin", "Chào", "Bà",
                                          "Con");
```

```
            // thêm vào 6 chuỗi chữ bằng cách dùng hàm hành sự Add()
            lbt.Add("Không");
            lbt.Add("Biết");
            lbt.Add("Vì");
            lbt.Add("Sao");
            lbt.Add("Ta");
            lbt.Add("Buồn");

            // thử nghiệm việc truy xuất bằng cách thay đổi trị thứ hai
            string subst = "Vũ trụ";
            lbt[1] = subst;

            // truy xuất tất cả các chuỗi chữ
            for (int i = 0; i < lbt.GetNumEntries(); i++)
            {
                Console.WriteLine("lbt[{0}]: {1}", i, lbt[i]);
            }
        }
    }
}
```

Kết xuất

Điều đầu tiên bạn để ý là hai biến thành viên private:

```
private string[] myStrings;
private int myCtr = 0;
```

Trong chương trình này, ô liệt kê duy trì một bản dãy chuỗi chữ đơn giản, **myStrings**. Thật ra, trong thực tế một ô liệt kê có thể sẽ sử dụng một bản dãy linh động (dynamic array) như một "bảng băm" (hash table) chẳng hạn. Biến **myCtr** dùng theo dõi bao nhiêu chuỗi chữ đã được đưa vào.

Bạn khởi gán bản dãy **myStrings** trong hàm khởi dựng với câu lệnh:

```
myStrings = new String[256];
```

Phần còn lại của hàm constructor là thêm các thông số vào bản dãy sử dụng vòng lặp foreach. Bạn để ý việc sử dụng từ chốt **params**, vì không biết trước bao nhiêu chuỗi chữ sẽ được thêm vào. Để cho đơn giản, ta chỉ thêm các chuỗi chữ mới vào theo thứ tự nhận được. Trong thực tế, ta có thể chèn thêm vào một nơi nào đó giữa bản dãy chẳng hạn .

Hàm hành sự **Add()** của **ListBoxTest** đơn giản chỉ làm mỗi một việc là ghi nối đuôi chuỗi chữ mới vào bản dãy **myStrings**.

Tuy nhiên, hàm hành sự chủ chốt của **ListBoxTest** là indexer. Indexer sẽ không mang tên, do đó bạn sử dụng từ chốt **this**:

```
public string this[int index];
```

Cú pháp của indexer tương tự như cú pháp của thuộc tính, nghĩa là có một hàm hành sự **get()** hoặc một hàm hành sự **set()** hoặc cả hai. Trong trường hợp này, **get()** cho kiểm tra sơ sài giới hạn bản dãy, và khi chấp nhận chỉ số sẽ trả về trị yêu cầu. Nếu quá giới hạn, ta tung ra một câu lệnh biệt lệ (exception) như dưới đây (in đậm):

```
get
{   if index < 0 || index >= myStrings.Length)
    {   // xử lý chỉ mục sai, vượt ngoài giới hạn
        throw new IndexOutOfRangeException("Vượt quá giới hạn!");
    }
    return myStrings[index];
}
```

Còn hàm **set()** cho kiểm tra xem chỉ số bạn cung cấp đã có chuỗi chữ tương ứng trong bản dãy hay không.

```
set
{   // chỉ thêm thông qua hàm Add()
    if (index >= myCtr)
    {   xử lý sai lầm
    }
    else
        myStrings[index] = value;
}
```

Do đó, khi bạn viết câu lệnh:

```
ListBoxTest[5] = "Chào anh Ba";
```

thì trình biên dịch sẽ gọi hàm **set()** đối với đối tượng rồi chuyển chuỗi chữ "Chào anh Ba" như là thông số hiểu ngầm mang tên **value**.

10.2.2 Indexer và việc gán trị

Trong thí dụ 10-11 trên, bạn không thể gán cho một chỉ số không có trị tương ứng. Do đó, nếu bạn viết:

```
ListBoxTest[12] = "Hi hi!";
```

Bạn sẽ gây ra một biệt lệ sai lầm trong hàm hành sự **set()**, vì chỉ số (12) bạn trao qua lớn hơn trị của cái đếm **myCtr** chỉ có (10).

Lẽ dĩ nhiên, bạn có thể dùng hàm hành sự **set()** để gán gì đó; đơn giản bạn chỉ cần thao tác các chỉ số bạn nhận được. Muốn thế, bạn cho thay đổi **set()** bằng cách kiểm tra **Length** của bản dãy thay vì trị hiện hành của cái đếm. Nếu một trị được đưa vào đối với một chỉ số chưa nhận được trị, bạn chỉ cần nhật tu **myCtr**:

```
set
{   // chỉ thêm thông qua Add()
    if (index >= myStrings.Length)
    {   // xử lý sai lầm
    }
    else
    {   myStrings[index] = value;
        if (myCtr < index + 1)
            myCtr = index + 1;
    }
}
```

Việc này cho phép bạn tạo ra một bản dãy "lơ thơ" (sparse array), theo đấy bạn có thể gán cho offset 12, mà chưa bao giờ gán cho offset 11. Do đó, nếu bạn viết:

```
ListBoxTest[12] = "Hi hi!";
```

thì kết xuất sẽ như sau:

lbt[0]: Xin
lbt[1]: Vũ trụ
lbt[2]: Bà
lbt[3]: Con
lbt[4]: Không
lbt[5]: Biết
lbt[6]: Vì
lbt[7]: Sao
lbt[8]: Ta

lbt[9]: Buốn
lbt[10]:
lbt[11]:
lbt[12]: Hi hi!

Trên **Main()** bạn tạo một thể hiện của **ListBoxTest** cho mang tên **lbt** và trao cho nó bốn chuỗi chữ như là thông số:

```
ListBoxTest lbt = new ListBoxTest("Xin", "Chào", "Bà", "Con");
```

Sau đó, bạn triệu gọi hàm **Add()** để thêm vào 6 chuỗi chữ nữa:

```
lbt.Add("Không");
lbt.Add("Biết");
lbt.Add("Vì");
lbt.Add("Sao");
lbt.Add("Ta");
lbt.Add("Buồn");
```

Trước khi quan sát các trị, bạn cho thay đổi trị thứ hai (ở chỉ mục 1):

```
string subst = "Vũ trụ";
lbt[1] = subst;
```

Cuối cùng, bạn cho hiển thị các trị thông qua vòng lặp for:

```
for (int i = 0; i < lbt.GetNumEntries(); i++)
{   Console.WriteLine("lbt[{0}]: {1}", i, lbt[i]);
}
```

10.2.3 Chỉ mục dựa trên các kiểu trị khác

C# không bắt buộc bạn bao giờ cũng phải sử dụng một trị số nguyên làm chỉ số đối với một collection. Khi bạn tạo một tập hợp tự tạo (custom collection) kèm theo indexer, bạn tự do chọn indexer dùng chuỗi chữ làm chỉ mục, hoặc các kiểu dữ liệu khác. Thực tế, trị chỉ mục có thể được nạp chồng (overloaded) như vậy một collection nào đó có thể được chỉ mục, thí dụ bởi một trị số nguyên hoặc bởi một trị chuỗi chữ, tùy theo nhu cầu người sử dụng.

Trong trường hợp ô liệt kê của chúng ta, ta muốn có khả năng chỉ mục ô liệt kê dựa trên chuỗi chữ. Thí dụ 10-12 minh hoạ việc sử dụng một chỉ mục kiểu chuỗi chữ. Indexer sẽ triệu gọi một hàm hành sự hỗ trợ (helper method) **findString()** cho trả về một mẫu tin

(record) dựa trên trị của chuỗi được cung cấp. Bạn để ý là indexer được nạp chồng và indexer từ thí dụ 10-11 có khả năng sống chung hoà bình.

Thí dụ 10-12: Nạp chồng một chỉ mục

```
namespace Prog_CSharp
{  using System;

   // Một ListBox đã được đơn giản hoá
   public class ListBoxTest
   {  // khởi gán ô liệt kê với các chuỗi chữ trên hàm constructor
      public ListBoxTest(params string[] initialStrings)
      {  myStrings = new string[256];// cấp phát ký ức cho chuỗi chữ
         // sao các chuỗi chữ được trao qua cho hàm constructor
         foreach (string s in initialStrings)
         {  myStrings[myCtr++] = s;
         }
      }

      // thêm một chuỗi chữ đơn vào cuối ô liệt kê
      public void Add(string theString)
      {  if (myCtr >= myStrings.Length)
         {  // xử lý chỉ mục sai
            throw new IndexOutOfRangeException("Vượt quá giới hạn");
         }
         else
            myStrings[myCtr] = theString;
            myCtr++;
      }

      // định nghĩa một indexer kiểu số nguyên
      // cho phép truy xuất giống như bản dãy
      public string this[int index]
      {  get
         {  if (index < 0 || index > myStrings.Length)
            {  // xử lý chỉ mục sai
               throw new IndexOutOfRangeException("Vượt qúa giới hạn");
            }
            return myStrings[index];
         }
         set
         {  // chỉ thêm thông qua hàm Add()
            if (index >= myCtr)
            {  xử lý sai lầm
               throw new IndexOutOfRangeException("Vượt qúa giới hạn");
            }
            else
               myStrings[index] = value;
         }
      }
```

```
    private int findString(string searchString)
    {   for (int i = 0; i < myStrings.Length; i++)
        {   if (myStrings[i].StartsWith(searchString)
            {   return i;
            }
        }
        return -1
    }

    // indexer dựa trên chuỗi chữ
    public string this[string index]
    {   get
        {   if (index.Length == 0)
            {   // xử lý chỉ mục sai
            }
            return this[findString(index)];
        }
        set
        {   myStrings[findString(index)] = value;
        }
    }

    // Loan báo cho biết cầm giữ bao nhiêu chuỗi chữ
    public int GetNumEntries()
    {   return myCtr;
    }

    private string[] myStrings;
    private int myCtr = 0;
}

public class Tester
{   static void Main()
    {   // tạo một ô liệt kê mới và cho khởi gán
        // hai chuỗi chữ như là thông số
        ListBoxTest lbt = new ListBoxTest("Hello", "World");

        // thêm vào 6 chuỗi chữ bằng cách dùng hàm hành sự Add()
        lbt.Add("Không");
        lbt.Add("Biết");
        lbt.Add("Vì");
        lbt.Add("Sao");
        lbt.Add("Ta");
        lbt.Add("Buồn");

        // thử nghiệm việc truy xuất bằng cách thay đổi trị thứ hai
        string subst = "Universe";
        lbt[1] = subst;
        lbt["Hel"] = "GoodBye";
        // lbt["xyz"] = "oops";

        // truy xuất tất cả các chuỗi chữ
        for (int i = 0; i < lbt.GetNumEntries(); i++)
```

```
            {   Console.WriteLine("lbt[{0}]: {1}", i, lbt[i]);
            }   // end của for
        }   // end của Main
    }   // end của Tester
}   // end của namespace
```

Kết xuất:

lbt[0]: GoodBye
lbt[1]: Universe
lbt[2]: Không
lbt[3]: Biết
lbt[4]: Vì
lbt[5]: Sao
lbt[6]: Ta
lbt[7]: Buồn

Thí dụ 10-12 giống như thí dụ 10-11, ngoại trừ việc bổ sung một indexer được nạp chồng có thể so khớp một chuỗi chữ, và hàm hành sự **findString()**, được tạo ra hỗ trợ chỉ mục này.

Hàm **findString()** chỉ rảo qua (iterate) các chuỗi chữ được trữ trong **myStrings** cho tới khi tìm ra chuỗi bắt đầu đầu giống chuỗi đích ta dùng trong chỉ mục. Hàm **StartsWith()**, thuộc lớp **String**, xác định xem phần khởi đầu của myStrings[i] có giống như searchString hay không. Nếu đúng, hàm **findString()** sẽ trả về chỉ mục của chuỗi chữ này, bằng không sẽ trả về -1. Nếu -1 được dùng làm chỉ mục đối với myStrings, thì sẽ gây ra biệt lệ **System.NullReferenceException**. Bạn có thấy câu lệnh mà chúng tôi cho đánh dấu// **lbt["xyz"] = "oops";**. Nếu bạn cho chạy câu lệnh này, bằng cách bỏ dấu//, thì sẽ gây ra biệt lệ.

Bạn thấy trong **Main()**, người sử dụng trao qua một chuỗi chữ ngắn "Hel" (phần đầu của chuỗi chữ "Hello") dùng làm chỉ mục giống như với một số nguyên:

```
lbt["Hel"] = "GoodBye";
```

Câu lệnh trên triệu gọi chỉ mục bị nạp chồng. Chỉ mục này cho kiểm tra sơ (trong trường hợp này bảo đảm là chuỗi được trao qua có ít nhất một ký tự) rồi trao trị (Hel) cho **findString()**. Nó sẽ nhận lại những chỉ số và sử dụng chỉ số này để chỉ mục lên **myStrings**:

```
return this[findString(index)];
```

Trị hàm **set()** cũng hoạt động như thế:

```
myStrings[findString(index)] = value;
```

Sau đây là một thí dụ cho thấy một lớp khung lưới (grid) khổ 26 x 10 mang một indexer với 2 thông số. Thông số thứ nhất bắt buộc phải là chữ nhỏ hoặc chữ hoa đi từ A đến Z, còn thông số thứ hai phải là số nguyên nằm trong giới hạn 0-9

Thí dụ 10-13: Sử dụng indexer trên một khung lưới (grid)

```
class Grid
{   const int NumRows = 26;    // số hàng
    const int NumCols = 10;    // số cột
    int[,] cells = new int[NumRows, NumCols];  // bản dãy hai chiều

    public int this[char c, int colm]// khai báo indexer có 2 thông số
    {
        get
        {   c = Char.ToUpper(c);    // đổi thành chữ hoa
            if (c < 'A' || c > 'Z')
                throw new ArgumentException();    // xử lý biệt lệ
            if (colm < 0 || colm >= NumCols)
                throw new IndexOutOfRangeException();// biệt lệ quá giới
                                                     //hạn
            return cells[c - 'A', colm];
    }
    set
        {   c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z')
        throw new ArgumentException();
      if (colm < 0 || colm >= NumCols)
        throw new IndexOutOfRangeException();
      cells[c - 'A', colm] = value;
     }
   }
}
```

10.2.4 Indexers trên các giao diện

Indexers có thể được khai báo trên các giao diện, theo cú pháp sau đây:

type **this [***type argument***] {get; set;}**

theo đấy *type, type argument* đều giống như với indexer trên lớp. Chỉ có phần các hàm **get(), set()** là hơi khác: phần thân của **get(), set()** chỉ gồm một dấu chấm phẩy. Mục đích của các hàm này là cho biết xem indexer thuộc loại nào: read-write, read-only, hoặc write-only.

Sau đây là một thí dụ về một hàm truy xuất trên indexer giao diện (interface indexer accessor):

```
public interface IMyInterface
{
  ...
 // khai báo Indexer
  string this[int index]
  {
   get;
   set;
  }
}
```

Dấu ấn của một indexer phải khác biệt so với các dấu ấn của các indexers khác được khai báo trên cùng interface.

Thí dụ 10-14 sau đây minh hoạ việc thiết đặt interface indexers.

Thí dụ 10-14: Indexer giao diện

```
namespace Prog_CSharp
{
  using System;
 // Indexer trên một interface:
  public interface IMyInterface
  { // khai báo indexer:
     int this[int index]
     {  get;
       set;
     }
  }

 // Thiết đặt interface trên lớp IndexerClass:
  class IndexerClass: IMyInterface
  {
    private int [] myArray = new int[100];// bản dãy số nguyên
    public int this [int index] // khai báo indexer
    {
           get
      {// kiểm tra giới hạn của index
          if (index < 0 || index >= 100)
         return 0;
          else
```

```
         return myArray[index];
      }
      set
      { if (!(index < 0 || index >= 100))
         myArray[index] = value;
      }
    }
  }

  public class Tester
  {
    public static void Main()
    {  IndexerClass b = new IndexerClass();
     // triệu gọi indexer để khởi gán các phần tử #3 và #5:
      b[2] = 4;
      b[5] = 32;
      for (int i=0; i<=10; i++)
      {  Console.WriteLine("Element #{0} = {1}", i, b[i]);
      }// end for
    }// end Main
  }// end Tester
}// end namespace
```

Kết xuất

Element #0 = 0
Element #1 = 0
Element #2 = 4
Element #3 = 0
Element #4 = 0
Element #5 = 32
Element #6 = 0
Element #7 = 0
Element #8 = 0
Element #9 = 0
Element #10 = 0

Trong thí dụ trên, bạn có thể sử dụng "tên chính danh đầy đủ" (fully qualified name) của interface để thiết đặt rõ ra thành viên interface. Thí dụ:

```
public string IMyInterface.this
{
}
```

Tuy nhiên, "tên chính danh đầy đủ" chỉ cần thiết tránh nhập nhằng khi một lớp nào đó thiết đặt indexer nhiều hơn một trên một interface cùng mang một dấu ấn. Thí dụ, nếu lớp **Employee** thiết đặt hai giao diện, **ICitizen** và **IEmployee**, và cả hai interface này

cùng có một dấu ấn indexer. Lúc này, "tên chính danh đầy đủ" sẽ là cần thiết. Do đó, khai báo indexer sau đây:

```
public string IEmployee.this
{
}
```

thiết đặt indexer trên interface **IEmployee**, trong khi khai báo sau đây:

```
public string ICitizen.this
{
}
```

lo thiết đặt indexer đối với interface **ICitizen**.

10.3 Tập hợp các đối tượng

Bây giờ ta xem qua một tập hợp những đối tượng mà ta gọi là collection. Khi đặt tên ta nên thêm chữ "s" vào cuối tên để chỉ là số nhiều. Đây chẳng qua chỉ là một qui ước nhận diện, không bắt buộc.

10.3.1 Collection là gì?

Collection (mà chúng tôi tạm dịch là một "tập hợp" thay vì một "bộ sưu tập") là một tập hợp những đối tượng cùng mang một kiểu dữ liệu được "bỏ chung" lại với nhau (giống như bỏ vào một cái bao bố vậy, do đó có tác giả chỉ collection là bag), và bạn có thể tuần tự "lôi" ra mỗi đối tượng bằng cách dùng vòng lặp **foreach**. Các đối tượng thuộc bất cứ kiểu dữ liệu nào cũng có thể gom thành một tập hợp kiểu dữ liệu **Object** để có thể lợi dụng lợi thế tiềm tàng của ngôn ngữ. Nói cách khác, khi bạn có thể viết gì đó như sau:

```
foreach (string NextMessage in MessageSet)
{
    DoSomething(NextMessage);
}
```

thì chắc chắn MessageSet là một collection. Khả năng sử dụng foreach là mục tiêu chính của collection. Visual Studio .NET sử dụng khá nhiều collection.

Tuy nhiên, trong một collection kiểu dữ liệu **Object**, ta phải tiến hành xử lý thêm trên từng phần tử, chẳng hạn việc chuyển đổi sử dụng "đóng hộp" (boxing) và "rả hộp" (unboxing) sẽ ảnh hưởng lên hiệu năng của collection. Điển hình boxing và unboxing sẽ

xảy ra khi ta cho trữ rồi tìm lại một kiểu dữ liệu theo trị (value type) trên một collection kiểu **Object**.

Những collection nào kiểm tra nghiêm ngặt về kiểu dữ liệu (gọi là strongly typed collection) sẽ tránh việc giảm hiệu năng này. **StringCollection** thuộc loại này. Ngoài ra, trên một strongly typed collection người ta thường thực hiện việc kiểm tra hợp lệ kiểu dữ liệu của mỗi phần tử được thêm vào collection.

Về mặt nội tại, một đối tượng là một collection khi nào nó có khả năng cung cấp một qui chiếu về một đối tượng có liên hệ, được biết là enumerator (bộ phận liệt kê), cho phép có khả năng lần lượt rảo qua các mục tin trong collection.

Tất cả các collection được dẫn xuất trực tiếp hoặc gián tiếp từ giao diện **ICollection** đều chia sẻ một số chức năng ngoài các hàm hành sự lo việc thêm, bớt hoặc truy tìm các phần tử. Ta có thể kể:

- ***Một Enumerator.*** Một collection phải thiết đặt giao diện **System.Collection.IEnumerator**. Giao diện **IEnumerator** chỉ có định nghĩa một hàm hành sự duy nhất như sau:

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

  Hàm **GetEnumerator()** sẽ trả về một đối tượng enumerator. Một enumerator là một đối tượng lo tuần tự rảo qua (iterate) một collection được gắn liền. Có thể xem như là một con trỏ di động chĩa về bất cứ phần tử nào trên collection. Một enumerator chỉ có thể được gắn liền với một collection mà thôi, nhưng một collection có thể có nhiều enumerator. Câu lệnh **foreach** sử dụng enumerator và che đi sự phức tạp trong việc thao tác trên enumerator.

- ***Các thành viên đồng bộ hoá*** (Synchronization members). Việc đồng bộ hoá đem lại sự an toàn về thread khi truy xuất các phần tử collection. Theo mặc nhiên, collection không mang tính an toàn về thread. Chỉ có vài lớp trong namespace **Collection** là tạo một vỏ bọc (wrapper) thread-safe lên collection. Tuy nhiên, tất cả các lớp trong namespace **Collection** đều cung cấp cho bạn thuộc tính **SyncRoot** mà bạn có thể đem dùng trong các lớp dẫn xuất để tự tạo vỏ bọc thread-safe riêng cho mình. Ngoài ra, cũng có thêm một thuộc tính **IsSynchronized** giúp bạn xác định liệu xem collection có thread-safe hay không.

- Hàm hành sự **CopyTo**. Tất cả các collection có thể được sao qua một bản dãy sử dụng hàm **CopyTo**. Tuy nhiên, thứ tự các phần tử trên bản dãy mới tùy thuộc vào

trật tự mà enumerator trả về. Bản dãy kết quả bao giờ cũng là một chiều với giới hạn thấp (low bound) bằng zero.

Các chức năng sau đây được thiết đặt trên vài lớp trên namespace **Collection**

- **Khả năng và cái đếm**. Khả năng (capacity) của một collection là số phần tử mà collection *có thể* chứa. Cái đếm (count) của một collection là số phần tử hiện được trữ trong collection. **BitArray** là một trường hợp đặc biệt vì capacity và count bằng nhau. Một vài collection che dấu capacity hoặc count hoặc cả hai.

  Tất cả các collection trên namespace **Collection** tự động nới rộng khả năng khi khả năng hiện hành bị vượt quá giới hạn. Ký ức được cấp phát lại và các phần tử được chép lại từ cũ qua mới. Điều này có thể ảnh hưởng lên hiệu năng của collection. Cách hay nhất tránh giảm thiểu hiệu năng là nên ước tính kích thước ban đầu của collection để khỏi cấp phát lắt nhắt.

- ***Giới hạn thấp***. Giới hạn thấp (lower bound) trên một collection là chỉ số của phần tử đầu tiên. Tất cả các collection được chỉ mục trên namespace **Collection** đều có giới hạn thấp bằng zero. Bản dãy theo mặc nhiên có giới hạn thấp bằng zero, nhưng một giới hạn thấp khác có thể được định nghĩa khi tạo một thể hiện của lớp **Array** sử dụng hàm **Array.CreateInstance()**.

Các lớp collection có thể tổng quát phân làm 3 loại:

- ***Các collection chung chung.*** Đây là những collection dữ liệu phổ biến khác nhau, như hash tables, queue, stack, dictionaries và list.

- ***Bit collection***. Đây là những collection gồm toàn những "cờ hiệu dạng bit" (bit flag). Chúng hoạt động có hơi khác so với những collection khác.

- ***Collection chuyên môn*** Đây là những collection được thiết kế với mục đích rất đặc biệt, thường dùng thụ lý một phần tử có kiểu dữ liệu đặc thù, chẳng hạn **StringDictionary**

Bạn nên cẩn thận khi chọn sử dụng một loại collection nào đó. Vì mỗi loại collection sẽ có những chức năng riêng của nó, kèm theo những giới hạn riêng của collection. Collection nào càng chuyên môn, thì hạn chế càng cao.

10.3.2 Khảo sát namespace System.Collections

Trong phần đi trước, ta đã làm quen với **System.Array** là một kiểu tập hợp sơ đẳng. Trong phần này, chúng ta sẽ đề cập đến **System.Collections**. Namespace này định nghĩa một số giao diện chuẩn dùng để liệt kê (enumerating), so sánh và tạo những collection, cũng như truy xuất nội dung các collection này. Sau đây là một số collection interface chủ chốt:

- **ICollection:** Định nghĩa những đặc tính chung (thí dụ read-only, thread safe, v.v.) đối với một lớp collection. **ICollection** sẽ được thiết đặt bởi tất cả các collection để cung cấp hàm hành sự **CopyTo()** cũng như các thuộc tính **Count, IsReadOnly, IsSynchronized** và **SyncRoot**.

- **IComparer:** cho phép so sánh hai đối tượng nằm trong collection, như vậy collection có thể được sắp xếp. Giao diện này thường dùng phối hợp với các hàm hành sự **Array.Sort** và **Array.BinaySearch**.

- **IDictionary:** Cho phép một đối tượng collection trình bày nội dung của mình như là một cặp "mục khóa – trị" (key-value). Ta có thể những lớp collection thiết đặt giao diện như **HashTable, SortedList.**

- **IDictionaryEnumerator**: cho phép liệt kê các phần tử của một collection chịu hỗ trợ **IDictionary.** Chỉ cho phép đọc dữ liệu trong collection chứ không cho phép thay đổi dữ liệu.

- **IEnumerable:** trả về một giao diện **IEnumerator** đối với một đối tượng chỉ định. Giao diện sẽ rảo liệt kê collection sử dụng câu lệnh **foreach**.

- **IEnumerator**: thông thường giao diện này được dùng hỗ trợ kiểu liệt kê đơn giản **foreach** trên collection. Chỉ cho phép đọc dữ liệu trong collection chứ không cho phép thay đổi dữ liệu.

- **IList:** Giao diện này tượng trưng cho một collection các đối tượng mà ta có thể truy xuất riêng rẽ bằng inde10. Nghĩa là cung cấp những hàm hành sự cho phép thêm, gỡ bỏ và chỉ mục những mục dữ liệu trên một bảng danh sách các đối tượng.

Sau đây là các lớp collection mà chúng tôi sẽ lần lượt đề cập. Bảng sau đây gồm 3 cột: tên lớp collection, ý nghĩa mỗi lớp collection và những interface quan trọng được thiết đặt.

ArrayList	Một bản dãy đối tượng với kích thước động, nghĩa là chiều dài thay đổi theo nhu cầu.	**IList, ICollection, IEnumerable** và **ICloneable.**
Hashtable	Tượng trưng một collection gồm những cặp key-value được tổ chức dựa trên mã băm (hash code) của mục khoá.	Các kiểu dữ liệu được trữ trên một Hashtable bao giờ cũng phủ quyết `System.Object.GetHashCode()`. **IDictionary, ICollection, IEnumerable** và **ICloneable.**
Queue	Tượng trưng một hàng nối đuôi kiểu FIFO (first-in-first-out), vào trước ra trước.	**ICollection, IEnumerable** và **ICloneable.**
SortedList	Giống như một dictionary, tuy nhiên các phần tử cũng có thể được truy xuất theo thứ tự vị trí (ordinal position) nghĩa là dựa theo chỉ mục	**IDictionary, ICollection, IEnumerable** và **ICloneable.**
Stack	Một hàng nối đuôi kiểu LIFO (last in first out), vào sau ra trước, cung cấp những chức năng push và pop.	**ICollection,** và **IEnumerable**

10.3.2.1 IEnumerable Interface

Bạn có thể hỗ trợ câu lệnh **foreach** trên **ListBoxTest** bằng cách thiết đặt giao diện **IEnumerable**. Giao diện này chỉ có duy nhất một hàm hành sự **GetEnumerator()**, chỉ lo mỗi một việc là trả về một enumerator, một thiết đặt chuyên biệt của giao diện **IEnumerator**, và bạn có thể dùng enumerator này để rảo qua collection. Do đó, ý nghĩa của **IEnumerable** là nó có thể cung cấp một enumerator, mà ta tạm đặt tên là ListBoxEnumerator, chẳng hạn:

```
public IEnumerator GetEnumerator()
{   return (IEnumerator) new ListBoxEnumerator(this);
}
```

Enumerator phải thiết đặt các hàm hành sự (**MoveNext()** và **Reset()**) và thuộc tính **Current** của giao diện **IEnumerator**. Việc thiết đặt có thể được thực hiện trực tiếp bởi lớp "thùng chứa" (container class), ở đây là **ListBoxTest,** hoặc bởi một lớp riêng biệt. Cách tiếp cận sau được ưa thích, vì nó giao trách nhiệm về cho lớp enumerator (ở đây **ListBoxEnumerator**) thay vì dồn cục vào lớp "thùng chứa".

Vì lớp enumerator là một lớp đặc thù đối với lớp "thùng chứa" (nghĩa là **ListBoxEnumerator** phải biết cặn kẽ **ListBoxTest**), bạn sẽ thiết đặt private, nằm trong lòng **ListBoxTest**.

Bạn để ý hàm hành sự **GetEnumerator()** trao đối tượng của **ListBoxTest** (ở dây là **this**) cho enumerator, cho phép enumerator liệt kê nôi dung của đối tượng đặc biệt **ListBoxTest** này.

Lớp tạo enumerator ở đây được thiết đặt dưới cái tên là **ListBoxEnumerator**, một lớp private được định nghĩa *trong lòng* **ListBoxTest**. Nó hoạt động khá đơn giản. Nó phải thiết đặt một thuộc tính **Current** (tiếp nhận nội dung của phần tử collection hiện hành), và hai hàm hành sự **MoveNext()** (di chuyển enumerator về phần tử kế tiếp) và **Reset()** (cho enumerator về vị trí ban đầu, nghĩa là trước phần tử đầu tiên trên collection).

Đối tượng **ListBoxTest** mà ta cần liệt kê sẽ được trao cho hàm constructor như là một đối mục, được gán cho một biến thành viên **lbt**. Hàm constructor cũng sẽ cho **index** về -1 báo cho biết bạn chưa bắt đầu tiến trình liệt kê đối tượng:

```
public ListBoxEnumerator(ListBoxTest lbt)
{   this.lbt = lbt;
    index = -1;
}
```

Hàm hành sự **MoveNext()** cho tăng index, rồi cho kiểm tra bảo đảm là bạn không vượt qua cuối đối tượng mà bạn liệt kê. Nếu vượt quá, thì hàm trả về **false**, bằng không là **true**:

```
public bool MoveNext()
{   index++;
    if (index >= lbt.myStrings.Length)
        return false;
    else
        return true;
}
```

Còn hàm **Reset()** đơn giản cho index về -1.

Thuộc tính **Current** được thiết đặt trả về chuỗi hiện hành. Đây là một quyết định tự ý; trong các lớp khác, **Current** sẽ mang một ý nghĩa nào đó mà lập trình viên cho là thích ứng. Tuy nhiên, mỗi enumerator phải có khả năng trả về thành viên hiện hành, vì truy xuất thành viên hiện hành là mục tiêu của enumerator:

```
public object Current
{   get
    {   return(lbt[index]);
```

```
        }
    }
```

Đây là tất cả những gì bạn cần biết. Việc triệu gọi **foreach** sẽ đi tìm enumerator rồi dùng nó để liệt kê phần tử bản dãy. Vì **foreach** sẽ cho hiển thị mọi chuỗi chữ, cho dù bạn có thêm hay không một trị có ý nghĩa, bạn cho thay đổi việc khởi gán **myStrings** về 10 để in ra ngăn ngắn. Sau đây là thí dụ 10-15:

Thí dụ 10-15: Tạo một ListBox thành một lớp liệt kê được.

```
namespace Prog_CSharp
{  using System;
   using System.Collections;

   // Một ListBox đã được đơn giản hoá
   public class ListBoxTest: IEnumerable
   {  // thiết đặt một ListBoxEnumerator private
      private class ListBoxEnumerator: IEnumerator
      {  // public trong lòng một thiết đặt private, do đó
         // private trong lòng ListBoxTest
         public ListBoxEnumerator(ListBoxTest lbt)  // hàm constructor
         {  this.lbt = lbt;
            index = -1;
         }

         // hàm hành sự và thuộc tính của enumerator
         public bool MoveNext()
         {  index++;
            if (index >= lbt.myStrings.Length)
               return false;
            else
               return true;
         }

         public void Reset()
         {  index = -1;
         }

         // thuộc tính Current được định nghĩa là
         // chuỗi chữ chót được thêm vào
         public object Current
         {  get
            {  return(lbt[index]);
            }
         }
         private ListBoxTest lbt;
         private int index;
      }

      // Các lớp enumerator có thể trả về một enumerator
      public IEnumerator GetEnumerator()
```

```
    {   return (IEnumerator) new ListBoxEnumerator(this);
    }

    // khởi gán ô liệt kê với các chuỗi chữ trên hàm constructor
    public ListBoxTest(params string[] initialStrings)
    {   myStrings = new String[10];// cấp phát ký ức cho chuỗi chữ
        // sao các chuỗi chữ được trao qua cho hàm constructor
        foreach (string s in initialStrings)
        {   myStrings[myCtr++] = s;
        }
    }

    // thêm một chuỗi chữ đơn vào cuối ô liệt kê
    public void Add(string theString)
    {   if (myCtr >= myStrings.Length)
        {   // xử lý chỉ mục sai
        }
        else
            myStrings[myCtr++] = theString;
    }
    // định nghĩa một indexer cho phép truy xuất giống như bản dãy
    public string this[int index]
    {   get
        {   if (index < 0 || index > myStrings.Length)
            {   // xử lý chỉ mục sai
                throw new IndexOutOfRangeException("Vượt qúa giới hạn");
            }
            return myStrings[index];
        }
        set
        {   // chỉ thêm thông qua hàm Add()
            if (index >= myCtr)
            {   xử lý sai lầm
            }
            else
                myStrings[index] = value;
        }
    }
    // Loan báo cho biết cầm giữ bao nhiêu chuỗi chữ
    public int GetNumEntries()
    {   return myCtr;
    }
    private string[] myStrings;
    private int myCtr = 0;
}

public class Tester
{   static void Main()
    {   // tạo một ô liệt kê mới và cho khởi gán
        // hai chuỗi chữ như là thông số
        ListBoxTest lbt = new ListBoxTest("Hello", "World");
```

```
            // thêm vào 6 chuỗi chữ bằng cách dùng hàm hành sự Add()
            lbt.Add("Không");
            lbt.Add("Biết");
            lbt.Add("Vì");
            lbt.Add("Sao");
            lbt.Add("Ta");
            lbt.Add("Buồn");

            // thử nghiệm việc truy xuất bằng cách thay đổi trị thứ hai
            string subst = "Universe";
            lbt[1] = subst;

            // truy xuất tất cả các chuỗi chữ
            foreach (string s in lbt)
            {   Console.WriteLine("Trị: {0}", s);
            }
        }
    }
}
```

Kết xuất

```
Trị: Hello
Trị: Universe
Trị: Không
Trị: Biết
Trị: Vì
Trị: Sao
Trị: Ta
Trị: Buồn
Trị:
Trị:
```

Chương trình bắt đầu từ Main(), tạo một đối tượng **ListBoxTest** rồi trao chuỗi chữ "Hello", "World" cho hàm constructor. Khi đối tượng được tạo ra, một bản dãy **myStrings** được tạo dành trước 10 chỗ. Với các hàm **Add**, 6 chuỗi chữ được thêm vào, rồi chuỗi thứ hai được nhật tu, giống như thí dụ trước.

Khác biệt lớn trong phiên bản chương trình này là một vòng lặp **foreach** được dùng đến để truy tìm mỗi chuỗi chữ trong ô liệt kê. Vòng lặp **foreach** tự động sử dụng giao diện **IEnumerable**, triệu gọi **GetEnumerator()**, trở lui về **ListBoxEnumerator** theo đấy hàm constructor được triệu gọi, do đó khởi gán index -1. Vòng lặp **foreach** sau đó triệu gọi **MoveNext()**, cho tăng index lên 0, và trả về true. **foreach** liền dùng thuộc tính **Current** để lấy về chuỗi chữ hiện hành. Thuộc tính **Current** triệu gọi bộ indexer của ô liệt kê để lôi ra chuỗi chữ nằm ở index 0, gán chuỗi này cho biến **s**. Sau đó nội dung của **s** được hiển thị trên màn hình. Vòng lặp **foreach** lặp lại các bước trên (**MoveNext()**,

Current, hiển thị) cho tới khi nào tất cả các chuỗi chữ trên ô liệt kê hoàn toàn được hiển thị.

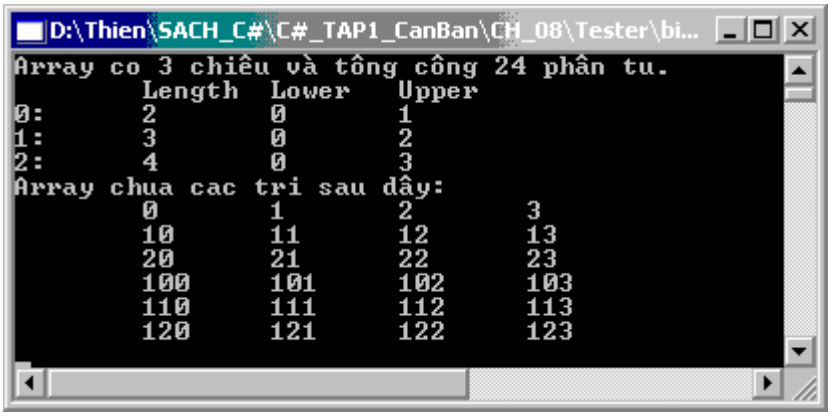
Sau đây là một thí dụ 10-16, **SamplesArray2**, được trích từ MSDN của Microsoft. Chương trình này tạo và khởi gán một bản dãy và cho hiển thị những thuộc tính và các phần tử bản dãy, sử dụng giao diện **IEnumerator**. Bạn tha hồ "ngâm cứu".

Thí dụ 10-16: Lớp SamplesArray2, sử dụng giao diện IEnumerator

```
using System;
using System.Collections;

public class SamplesArray2
{
  public static void Main()
  {
  // Tạo và khởi gán một bản dãy 3D kiểu dữ liệu Int32.
   Array myArr = Array.CreateInstance(typeof(Int32), 2, 3, 4);
   for (int i = myArr.GetLowerBound(0);
           i <= myArr.GetUpperBound(0); i++)
       for (int j = myArr.GetLowerBound(1);
                j <= myArr.GetUpperBound(1); j++)
        for (int k = myArr.GetLowerBound(2);
                  k <= myArr.GetUpperBound(2); k++)
            { myArr.SetValue((i*100)+(j*10)+k, i, j, k);
         }

  // Cho hiển thị các thuộc tính của bản dãy myArr
   Console.WriteLine("Array có {0} chiều và tổng cộng{1} phần tử.",
                  myArr.Rank, myArr.Length);
   Console.WriteLine("\tLength\tLower\tUpper");
   for (int i = 0; i < myArr.Rank; i++)
     {
      Console.Write("{0}:\t{1}", i, myArr.GetLength(i));
      Console.WriteLine("\t{0}\t{1}", myArr.GetLowerBound(i),
               myArr.GetUpperBound(i));
   }

  // Cho hiển thị nội dung của myArr
   Console.WriteLine("Array chứa các trị sau đây: ");
   PrintValues(myArr);
  }

  public static void PrintValues(Array myArr)
  {
   IEnumerator myEnumerator = myArr.GetEnumerator();
   int i = 0;
   int cols = myArr.GetLength(myArr.Rank - 1);
   while (myEnumerator.MoveNext())
   {
```

```
      if (i < cols)
      {
       i++;
      }
         else
         {
       Console.WriteLine();
       i = 1;
      }
      Console.Write("\t{0}", myEnumerator.Current);
    }
    Console.WriteLine();
   }// end while
}// end PrintValues
```

Kết xuất:.



Bạn để ý cho là enumerator không độc quyền truy xuất collection; do đó, các threads khác vẫn có thể thay đổi collection. Như vậy, enumerator sẽ tiếp tục hợp lệ khi nào collection không bị thay đổi. Nếu collection bị thay đổi, như thêm bớt phần tử hoặc phần tử bị nhật tu, thì enumerator sẽ bị bất hợp lệ, và enumerator sẽ tung ra một biệt lệ sai lầm **InvalidOperationException** vào lần tới khi enumerator được dùng đến.

10.3.2.2 ICollection Interface

Một giao diện chủ chốt khác dùng phối hợp với bản dãy, hoặc với tất cả các collection, cung cấp bởi .NET Framework, là **ICollection**. Giao diện này ấn định kích thước, enumerators và các hàm đồng bộ hoá đối với tất cả các collection. **ICollection** cung cấp 3 thuộc tính: **Count**, **IsSynchronized**, và **SyncRoot** và một hàm hành sự public

CopyTo(). Ta sẽ xem sau hàm **CopyTo()**. Thuộc tính được dùng thường xuyên là **Count**, cho biết số phần tử trong collection:

```
for (int i = 0; i < myIntArray.Count; i++)
{
    // làm gì ở đây…
}
```

Ở đây bạn sử dụng thuộc tính **Count** của **myIntArray** để xác định có bao nhiêu đối tượng để bạn có thể in ra trị.

10.3.2.3 IComparer Interface

Giao diện **IComparer**, được định nghĩa như sau:

```
// Một cách chung chung để so sánh hai đối tượng
interface IComparer
{
    int Compare(object obj1, object obj2);
}
```

cung cấp hàm hành sự **Compare()** cho phép so sánh hai đối tượng trong một collection để trả về một trị cho biết đối tượng này lớn hơn, bằng hoặc nhỏ thua đối tượng kia. Như vậy bất cứ hai mục tin trên collection có thể được sắp xếp.

Điển hình, hàm hành sự **Compare()** được thiết đặt bằng cách triệu gọi hàm hành sự **CompareTo()** của một trong những đối tượng. Hàm **CompareTo()** là một hàm hành sự của tất cả các đối tượng có thiết đặt giao diện **IComparable** được định nghĩa như sau:

```
// giao diện này cho phép một đối tượng chỉ rõ những mối tương
// quan của nó với các đối tượng khác cùng kiểu dữ liệu
interface IComparable
{
    int CompareTo(object obj);
}
```

Nếu bạn muốn tạo một lớp có thể được sắp xếp trong lòng một collection, thì bạn phải thiết đặt **IComparable**.

.NET Framework cung cấp cho bạn một lớp **Comparer** thiết đặt **IComparable**, và cung cấp một thiết đặt mặc nhiên case-sensitive. Lớp **Comparer** cú pháp như sau:

public sealed class Comparer: IComparer

Trong phần liên quan đến **ArrayList**, bạn sẽ biết cách thiết đặt giao diện **ICompa rable**.

10.3.3 Array Lists

Kiểu dữ liệu **Array** mà ta đã đề cập đến từ trước đến nay có một vấn đề: đó là kích thước cố định của bản dãy kiểu Array. Nếu bạn không biết trước bao nhiêu phần tử mà bản dãy có thể trữ, thì bạn gặp phải vấn đề hoặc khai báo một bản dãy quá nhỏ (mau cạn kiệt ký ức được cấp phát, thiếu chỗ chứa) hoặc quá lớn (phí phạm ký ức). Nghĩa là, nhiều lúc bạn không thể đoán trước bạn cần một bản dãy bao lớn.

Lớp **ArrayList** giúp bạn giải quyết vấn đề kể trên, vì đây là một loại bản dãy kích thước sẽ bành trướng một cách linh động (dynamic) theo nhu cầu. **ArrayList** cung cấp một số hàm hành sự và thuộc tính như theo bảng 10-3 sau đây:

Bảng 10-3: Các hàm hành sự và thuộc tính của lớp ArrayList

Capacity	Trả về/Cho đặt để số phần tử mà **ArrayList** có thể chứa.
Count	Trả về số phần tử hiện được trữ trên **ArrayList**.
IsFixedSize	Trả về một trị bool cho biết liệu xem **ArrayList** mang kích thước cố định hay không.
IsReadOnly	Trả về một trị bool cho biết liệu xem **ArrayList** thuộc loại read-only hay không .
IsSynchronized	Trả về một trị bool cho biết liệu xem việc truy xuất **ArrayList** có đồng bộ hay không (thread-safe).
Item	Trả về/ Cho đặt để phần tử bản dãy về chỉ mục được chỉ định. Trên C#, thuộc tính này là indexer đối với lớp **ArrayList**.
SyncRoot	Trả về một đối tượng có thể được dùng đồng bộ hoá việc truy xuất **ArrayList**.
Public Methods	
Adapter	Tạo một **ArrayList** wrapper (vỏ bọc) đối với một đặc thù **IList**
Add	Thêm một đối tượng vào cuối **ArrayList**.
AddRange	Thêm các phần tử của một ICollection vào cuối **ArrayList**.
BinarySearch	Overloaded. Sử dụng một giải thuật binary search để xác định vị trí của một phần tử hoặc một phần trên **ArrayList** đã được sắp xếp.
Clear	Gỡ bỏ tất cả các phần tử từ **ArrayList**.
Clone	Tạo một bản sao shallow của **ArrayList**.
Contains	Xác định liệu xem một phần tử có nằm trên **ArrayList** hay không.
CopyTo	Overloaded. Sao **ArrayList** hoặc một phần của nó lên bản dãy

	một chiều.
Equals (kế thừa từ Object)	Overloaded. Xác định liệu xem hai thể hiện **Object** có bằng nhau hay không .
FixedSize	Overloaded. Trả về một list wrapper với một kích thước cố định, theo đấy các phần tử được phép bị thay đổi, nhưng lại không được thêm vào hoặc gỡ bỏ.
GetEnumerator	Overloaded. Trả về một enumerator dùng rảo qua **ArrayList**.
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, dùng thích hợp trên những giải thuật băm và cấu trúc dữ liệu giống như một hash table.
GetRange	Trả về một **ArrayList** tượng trưng cho một tập hợp con (subset) của các phần tử **ArrayList** nguồn.
GetType (kế thừa từ Object)	Trả về kiểu dữ liệu **Type** của thể hiện hiện hành .
IndexOf	Overloaded. Trả về zero-based index của sự xuất hiện đầu tiên của một trị trên **ArrayList** hoặc trên một phần bản dãy.
Insert	Chèn thêm một phần tử vào **ArrayList** tại chỉ số được chỉ định.
InsertRange	Chèn các phần tử của một collection vào **ArrayList** tại chỉ số được chỉ định.
LastIndexOf	Overloaded. Trả về zero-based index của sự xuất hiện cuối cùng của một trị trên **ArrayList** hoặc trên một phần bản dãy.
ReadOnly	Overloaded. Trả về một list wrapper mang thuộc tính read-only.
Remove	Gỡ bỏ sự xuất hiện đầu tiên của một specific object khỏi **ArrayList**.
RemoveAt	Gỡ bỏ những phần tử tại chỉ số được chỉ định của **ArrayList**.
RemoveRange	Gỡ bỏ một khoảng phần tử từ **ArrayList**.
Repeat	Trả về một **ArrayList** mà các phần tử là những bản sao của trị được chỉ định.
Reverse	Overloaded. Đão ngược thứ tự của các phần tử trên **ArrayList** hoặc trên một phần bản dãy.
SetRange	Sao các phần tử của một collection chồng lên một phần phần tử trên **ArrayList**.
Sort	Overloaded. Sắp xếp các phần tử trên **ArrayList** hoặc trên một phần bản dãy
Synchronized	Overloaded. Trả về một list wrapper được đồng bộ hoá (thread-safe).
ToArray	Overloaded. Sao các phần tử của **ArrayList** về một bản dãy mới.
ToString (kế thừa từ Object)	Trả về một **String** tượng trưng cho Object hiện hành.

TrimToSize Cho đặt để khả năng về số phần tử hiện hành trên **ArrayList**.

Khi bạn tạo một **ArrayList**, bạn không xác định sẽ có bao nhiêu đối tượng trong bản dãy. Bạn thêm phần tử vào **ArrayList** bằng cách dùng hàm hành sự **Add()**, và ô liệt kê sẽ tự mình lo việc "giữ sổ sách" nội bộ đúng tình trạng, như theo thí dụ 10-17 sau đây:

Thí dụ 10-17: Làm việc với một ArrayList

```
namespace Prog_CSharp
{   using System;
    using System.Collections;

    // một lớp đơn giản Employee (nhân viên) để trữ lên bản dãy
    public class Employee
    {   public Employee(int empID) // hàm khởi dựng
        {   this.empID = empID; // mã số nhân viên
        }
        public override string ToString() // phủ quyết hàm ToString
        {   return empID.ToString();
        }
        public int EmpID    // thuộc tính EmpID, mã số nhân viên
        {   get
            {   return empID;
            }
            set
            {   empID = value;
            }
        }
        private int empID;
    }// end Employee

    public class Tester
    {   static void Main()
        {   ArrayList empArray = new ArrayList();   // bản dãy nhân viên
            ArrayList intArray = new ArrayList();   // bản dãy số nguyên
            // cho điền dữ liệu vào bản dãy
            for (int = 0; i < 5; i++)
            {   empArray.Add(new Employee(i+100));
                intArray.Add(i*5);
            }  // end for

            // in ra nội dung của intArray
            for (int i = 0; i < intArray.Count; i++)
            {   Console.Write("{0} ", intArray[i].ToString());
            }
            Console.WriteLine("\n");
            // in ra nội dung của empArray
            for (int i = 0; i < empArray.Count; i++)
            {   Console.Write("{0} ", empArray[i].ToString());
            }
```

```
            Console.WriteLine("\n");
            Console.WriteLine("empArray.Capacity: {0}",
                                              empArray.Capacity();
        }   // end Main
    }   // end Tester
}   // end namespace
```

Kết xuất

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

Với một lớp **Array**, bạn phải khai báo bao nhiêu đối tượng bản dãy sẽ chứa. Nếu bạn thêm quá sức chứa, thì **Array** sẽ tung ra một biệt lệ sai. Còn với **ArrayList** bạn không phải khai báo sức chứa, vì **ArrayList** có một thuộc tính **Capacity**, cho biết số phần tử mà bản dãy **ArrayList** có thể trữ:

```
public int Capacity {virtual get; virtual set;}
```

Khả năng mặc nhiên là 16. Khi bạn thêm phần tử thứ 17, thì khả năng sẽ tự động tăng lên xấp đôi, 32. Bạn có thể đặt để bằng tay khả năng bản dãy về bất cứ con số nào lớn hơn hoặc bằng số đếm **Count**. Nếu bạn cho về số nhỏ thua, chương trình sẽ tung ra biệt lệ sai **ArgumentOutOfRangeException**.

Sau đây là một thí dụ khác, 10-18, dùng **ArrayList** được trích từ MSDN của Microsoft, sử dụng đến enumerator:

Thí dụ 10-18: Sử dụng ArrayList với enumerator

```
using System;
using System.Collections;
public class SamplesArrayList
{   public static void Main()
   {// Tạo và khởi gán một ArrayList mới.
     ArrayList myAL = new ArrayList();
     myAL.Add("Hello");
         myAL.Add("World");
         myAL.Add("!");

         // Cho hiển thị các thuộc tính và trị của ArrayList.
         Console.WriteLine("myAL");
         Console.WriteLine("\tCount:  {0}", myAL.Count);
         Console.WriteLine("\tCapacity: {0}", myAL.Capacity);
         Console.Write("\tValues:");
         PrintValues(myAL);
   }

   public static void PrintValues(IEnumerable myList)
```

```
  {   IEnumerator myEnumerator = myList.GetEnumerator();
      while (myEnumerator.MoveNext())
         Console.Write("\t{0}", myEnumerator.Current);
     Console.WriteLine();
  }
}
```

Kết xuất:

```
myAL
 Count:  3
 Capacity: 16
 Values:  Hello  World  !
```

10.3.3.1 Thiết đặt giao diện IComparable

Giống như tất cả các collection, **ArrayList** thiết đặt hàm hành sự **Sort()** cho phép bạn sắp xếp bất cứ đối tượng nào thiết đặt giao diện **IComparable**. Trong thí dụ kế tiếp bạn sẽ thay đổi đối tượng **Employee** có thiết đặt **IComparable**:

```
public class Employee: IComparable
```

Muốn thiết đặt giao diện **IComparable**, đối tượng **Employee** phải cung cấp một hàm hành sự **CompareTo()**:

```
public int CompareTo(Object rhs)  // rhs tắt chữ righthand side
{
   Employee r = (Employee) rhs;  // ép kiểu từ Object về Employee
   return this.empID.CompareTo(r.empID);
}
```

Hàm hành sự **CompareTo()** nhận một đối tượng làm thông số; đối tượng **Employee** phải so sánh bản thân mình với đối tượng này, rồi trả về -1 nếu nó nhỏ thua đối tượng, 1 nếu lớn hơn, và 0 nếu bằng. Chính do **Employee** xác định gì là **nhỏ thua, lớn hơn** và **bằng**. Thí dụ, bạn sẽ "ép kiểu" (cast) đối tượng cho một **Employee**, rồi ủy quyền (delegate) so sánh cho thành viên số nguyên **empID** và sẽ dùng hàm hành sự **CompareTo()** mặc nhiên liên quan đến số nguyên để tiến hành so sánh hai trị số nguyên.

Bây giờ bạn sẵn sàng cho sắp xếp danh sách các nhân viên, **empArray**. Để xem việc sắp xếp thành công hay không, bạn sẽ cần thêm những số nguyên và những thể hiện đối tượng **Employee** vào các bản dãy tương ứng. Bạn sử dụng đến những trị số random thông qua lớp **Random**. Để tạo một trị random, bạn sẽ phải triệu gọi **Next()** trên đối tượng **Random**, trả về một số random. Hàm **Next()** bị nạp chồng; một phiên bản cho phép bạn trao một số nguyên tượng trưng cho một số random lớn nhất bạn muốn. Trong trường hợp này, bạn sẽ trao trị 10 để kết sinh (generate) một số random nằm giữa 0 và 10:

```
Random r = new Random();
r.Next(10);
```

Thí dụ 10-19 tạo một bản dãy số nguyên và một bản dãy **Employee**, cho điền vào cả hai bản dãy bởi số random, rồi in ra nội dung hai bản dãy này. Sau đó lại cho sắp xếp 2 bản dãy rồi lại in nội dung hai bản dãy được sắp xếp.

Thí dụ 10-19: Sắp xếp một bản dãy số nguyên và một bản dãy Employee

```
namespace Prog_CSharp
{  using System:
   using System.Collections;

   // tạo một lớp đơn giản để trữ trên bản dãy
   public class Employee: IComparable
   {  public Employee(int empID) // hàm khởi dựng
      {  this.empID = empID;     // mã số nhân viên
      }
      public override string ToString() // phủ quyết hàm ToString
      {  return empID.ToString();
      }

      // Comparer ủy quyền lui lại cho Employee. Employee sử dụng
      // hàm hành sự số nguyên CompareTo()
      public int CompareTo(Object rhs)
      {  Employee r = (Employee) rhs;
         return this.empID.CompareTo(r.empID);
      }
      private int empID;
   }  // end Employee

   public class Tester
   {  static void Main()
      {  ArrayList empArray = new ArrayList();   // bản dãy nhân viên
         ArrayList intArray = new ArrayList();   // bản dãy số nguyên
         // cho điền dữ liệu vào bản dãy thông qua số random
         Random r = new Random();
         for (int = 0; i < 5; i++)
         {  empArray.Add(new Employee(r.Next(10)+100));
            intArray.Add(r.Next(10));
         }  // end for

         // in ra nội dung của intArray
         for (int i = 0; i < intArray.Count; i++)
         {  Console.Write("{0} ", intArray[i].ToString());
         }
         Console.WriteLine("\n");

         // in ra nội dung của empArray
         for (int i = 0; i < empArray.Count; i++)
         {  Console.Write("{0} ", empArray[i].ToString());
```

```
            }
            Console.WriteLine("\n");

            // cho sắp xếp và hiển thị nội dung bản dãy intArray
            intArray.Sort();
            for (int i = 0; i < intArray.Count; i++)
            {   Console.Write("{0} ", intArray[i].ToString());
            }
            Console.WriteLine("\n");

            // cho sắp xếp và hiển thị nội dung bản dãy empArray
            empArray.Sort();
            for (int i = 0; i < empArray.Count; i++)
            {   Console.Write("{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

8 5 7 3 3
105 103 102 104 106
3 3 5 7 8
102 103 104 105 106

10.3.3.2 Thiết đặt giao diện IComparer

Khi bạn triệu gọi **Sort()** đối với **ArrayList**, thiết đặt mặc nhiên của giao diện **IComparer** được gọi vào, dùng đến **QuickSort** để triệu gọi thiết đặt **IComparable** của hàm hành sự **CompareTo()** trên mỗi phần tử bản dãy **ArrayList**.

Bạn hoàn toàn tự do tự thiết đặt **IComparer** riêng của bạn, cho phép bạn toàn quyền kiểm soát việc sắp xếp được thực hiện thế nào. Thí dụ, trong thí dụ kế tiếp bạn sẽ thêm một vùng mục tin thứ hai, **yearsOfSvc** (Years Of Service, số năm thâm niên công vụ) vào **Employee**. Bạn muốn có khả năng sắp xếp các đối tượng **Employee** trên **ArrayList**, dựa trên **empID** hoặc **yersOfSvc**.

Muốn thế, bạn sẽ tự tạo một thiết đặt giao diện **IComparer** mà bạn sẽ trao qua cho hàm hành sự **Sort()** của **ArrayList**. Lớp **IComparer**, ở đây là **EmpComparer**, sẽ biết rành các đối tượng **Employee** và biết cách sắp xếp chúng thế nào.

EmpComparer có một thuộc tính, **WhichComparison**, kiểu **Employee. EmpComparer.ComparisonType**:

```
public Employee.EmpComparer.ComparisonType WhichComparison
{   get
    {   return whichComparison;
    }
    set
    {   whichComparison = value;
    }
}
```

ComparisonType là một enumeration với hai trị: empID và yearsOfSvc (nghĩa là bạn muốn sắp xếp theo mã số nhân viên hoặc theo thâm niên công vụ):

```
public enum ComparisonType
{   EmpID;
    Yrs:
}
```

Trước khi triệu gọi **Sort()**, bạn sẽ tạo một thể hiện của **EmpComparer** và cho đặt để thuộc tính **ComparisonType**, như sau:

```
Employee.EmpComparer comp = Employee.GetComparer();
comp.WhichComparison =  Employee.EmpComparer.ComparisonType.EmpID;
empArray.Sort(comp);
```

Khi bạn triệu gọi **Sort()**, **ArrayList** sẽ triệu gọi hàm hành sự **Compare** đối với **Emp Comparer**, đến phiên lại ủy quyền so sánh cho hàm hành sự **Employee.CompareTo()** chuyển giao thuộc tính **WhichComparison**:

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

Đối tượng **Employee** phải thiết đặt một phiên bản tư tạo (custom) về **CompareTo()**, lo so sánh hai đối tượng một cách thích ứng:

```
public int CompareTo(
    Employee rhs,
    Employee.EmpComparer.ComparisonType which)
{   switch(wich)
    {   case Employee.EmpComparer.ComparisonType.EmpID:
            return this.empID.CompareTo(rhs.empID);
        case Employee.EmpComparer.ComparisonType.Yrs:
            return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
```

```
        }
        return 0;
    }
```

Thí dụ 10-20 sau đây cho thấy việc sử dụng IComparer để sắp xếp một bản dãy theo **empID** hoặc theo **yearsOfSvc.** Để đơn giản hoá, chúng tôi đã gỡ bỏ bản dãy số nguyên intArray.

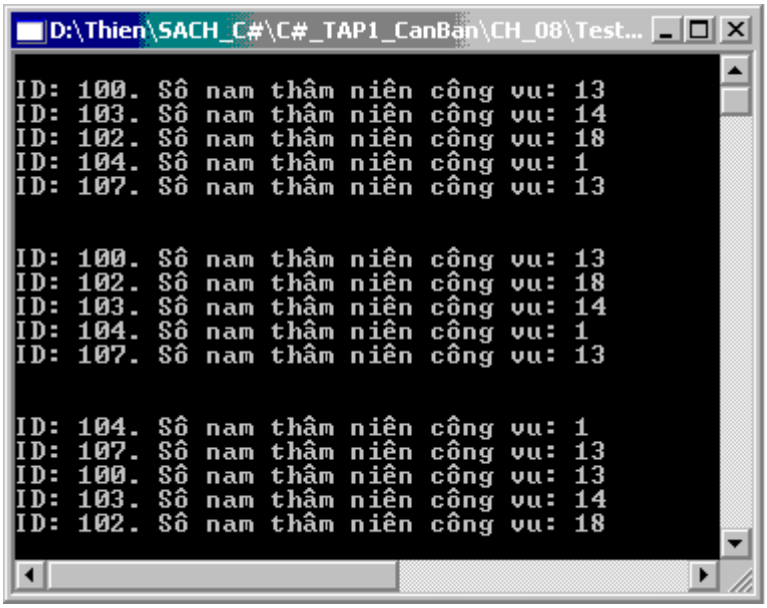
Thí dụ 10-20: Sắp xếp một bản dãy theo empID hoặc theo yearsOfSvc.

```
namespace Prog_CSharp
{   using System;
    using System.Collections;

    // tạo một lớp đơn giản để trữ trên bản dãy
    public class Employee: IComparable
    {   public Employee(int empID) // hàm khởi dựng
        {   this.empID = empID;     // mã số nhân viên
        }
        public Employee(int empID, int yearsOfSvc) // hàm khởi dựng
        {   this.empID = empID; // mã số nhân viên
            this.yearsOfSvc = yearsOfSvc; // thâm niên công vụ
        }

        public override string ToString() // phủ quyết hàm ToString
        {   return "ID: " + empID.ToString() +
                ". Số năm thâm niên công vụ: " + yearsOfSvc.ToString();
        }

        // hàm hành sự static để lấy một đối tượng Comparer
        public static EmpComparer GetComparer()
        {   return new Employeee.EmpComparer();
        }

        // Comparer ủy quyền lui lại cho Employee. Employee
        // sử dụng hàm hành sự số nguyên CompareTo()
        public int CompareTo(Object rhs)
        {   Employee r = (Employee) rhs;
            return this.empID.CompareTo(r.empID);
        }

        // thiết đặt đặc biệt để được triệu gọi bởi
        // bộ so sánh tự tạo (custom comparer)
        public int CompareTo(
            Employee rhs,
            Employee.EmpComparer.ComparisonType which)
        {   switch(which)
            {   case Employee.EmpComparer.ComparisonType.EmpID:
                    return this.empID.CompareTo(rhs.empID);
                case Employee.EmpComparer.ComparisonType.Yrs:
                    return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
```

```
        }
        return 0;
    }

    // lớp được nằm lồng dùng thiết đặt IComparer
    public class EmpComparer: IComparer
    {   // enum kiểu so sánh
        public enum ComparisonType
        {   EmpID,
            Yrs
        }

    // Yêu cầu các đối tượng Employee tự so sánh
    public int Compare(object lhs, object rhs)
    {   Employee l = (Employee) lhs;
        Employee r = (Employee) rhs;
        return l.CompareTo(r, WhichComparison);
    }

    public Employee.EmpComparer.ComparisonType WhichComparison
    {   get
        {   return whichComparison;
        }
        set
        {   whichComparison = value;
        }
    }

    // private state variable
    private Employee.EmpComparer.ComparisonType whichComparison;
}


    private int yearsOfSvc = 1;
    private int empID;
}   // end Employee

public class Tester
{   static void Main()
    {   ArrayList empArray = new ArrayList();   // bản dãy nhân viên
        // cho điền dữ liệu vào bản dãy thông qua số random
        Random r = new Random();
        for (int i = 0; i < 5; i++)
        {   empArray.Add(new Employee(r.Next(10)+100, r.Next(20)));
        }   // end for

        // in ra nội dung của empArray
        for (int i = 0; i < empArray.Count; i++)
        {   Console.Write("\n{0} ", empArray[i].ToString());
        }
        Console.WriteLine("\n");

        // cho sắp xếp và hiển thị nội dung bản dãy empArray
        Employee.EmpComparer comp = Employee.GetComparer();
```

```
            comp.WhichComparison =
                      Employee.EmpComparer.ComparisonType.EmpID;
            empArray.Sort(comp);
            for (int i = 0; i < empArray.Count; i++)
            {   Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
            comp.WhichComparison =
                   Employee.EmpComparer.ComparisonType.Yrs;
            empArray.Sort(comp);
            for (int i = 0; i < empArray.Count; i++)
            {   Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
        }   // end Main
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất



Phần kết xuất gồm 3 khối: khối thứ nhất cho thấy các đối tượng **Employee** như được thêm vào **ArrayList**. Trị empID và yearsOfSvc theo thứ tự ngẫu nhiên (random order). Khối thứ hai cho thấy sắp xếp theo thứ tự empID, còn khối thứ ba thì sắp xếp theo thứ tự năm thâm niên công vụ.

10.4 Hàng nối đuôi (Queue)

Một ***queue*** (hàng nối đuôi) là một collection kiểu FIFO (tắt chữ first-in-first-out) nghĩa là vào trước ra trước giống như hàng nối đuôi trước quầy bán vé các rạp xi nê.

Queue là một collection rất tốt khi bạn cần quản lý một nguồn lực (resource) hạn chế. Thí dụ, có thể bạn muốn chuyển một thông điệp cho một nguồn lực chỉ có thể giải quyết một thông điệp trong một lúc. Lúc này, bạn có thể tạo một queue để có thể bảo khách hàng: "Thông điệp của bạn rất ư quan trọng, vì ở đây thông điệp được giải quyết theo thứ tự nhận được".

Lớp **Queue** có một số hàm hành sự cũng như thuộc tính được liệt kê sau đây:

Public Properties	
Count	Trả về số phần tử hiện được trữ trên **Queue**.
IsSynchronized	Trả về một trị bool cho biết liệu xem việc truy xuất **Queue** có đồng bộ hay không (thread-safe).
SyncRoot	Trả về một đối tượng có thể được dùng đồng bộ hoá việc truy xuất **Queue**.
Public Methods	
Clear	Gỡ bỏ tất cả các phần tử từ **Queue**.
Clone	Tạo một bản sao shallow của **Queue**.
Contains	Xác định liệu xem một phần tử có nằm trên **Queue** hay không.
CopyTo	Sao các phần tử của **Queue** hoặc một phần của nó lên bản dãy một chiều hiện hữu khởi đi từ một chỉ số chỉ định.
Dequeue	Gỡ bỏ và trả về đối tượng về đầu hàng của **Queue**.
Enqueue	Thêm một đối tượng vào cuối hàng của **Queue**.
Equals (kế thừa từ Object)	Overloaded. Xác định liệu xem hai thể hiện Object có bằng nhau hay không.
GetEnumerator	Trả về một enumerator cho phép rảo qua hàng nối đuôi **Queue**.
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng cho việc sử dụng trong các giải thuật băm và các cấu trúc dữ liệu giống như một hash table.
GetType (kế thừa từ Object)	Trả về kiểu dữ liệu Type của thể hiện hiện hành.
Peek	Trả đối tượng về đầu hàng nối đuôi **Queue** không gỡ bỏ nó đi.
Synchronized	Trả về một **Queue** wrapper được đồng bộ hóa (thread-safe).

ToArray	Sao các phần tử của **Queue** qua một bản dãy mới.
ToString (kế thừa từ Object)	Trả về một String tượng trưng cho Object hiện hành.
TrimToSize	Cho đặt để khả năng về số phần tử hiện hành trên **Queue**.

Bạn thêm các phần tử vào queue dùng hàm hành sự **Enqueue**, cũng như gỡ bỏ các phần tử này khỏi queue thông qua hàm hành sự **Dequeue**, hoặc bằng cách dùng một enumerator. Thí dụ 10-21 sau đây: minh hoạ việc sử dụng một hàng nối đuôi:

Thí dụ 10-21: Cách dùng một hàng nối đuôi thế nào

```
namespace Prog_CSharp
{  using System;
   using System.Collections;

   public class Tester
   {  static void Main()
      {  Queue myIntQueue = new Queue();
         // điền dữ liệu vào
         for (int = 0; i < 5; i++)
         {  myIntQueue.Enqueue(i*5);   // qua mỗi tua thì nhân 5
                                       // cho chỉ số
         }

         // Cho hiển thị nội dung của myIntQueue
         Console.Write("Trị của myIntQueue: \t");
         PrintValues(myIntQueue);

         // Cho gỡ bỏ một phần tử khỏi myIntQueue
         Console.WriteLine("\n(Dequeue)\t{0}", myIntQueue.Dequeue());

         // Cho hiển thị nội dung của myIntQueue sau Dequeue
         Console.Write("Trị của myIntQueue: \t");
         PrintValues(myIntQueue);

         // Cho gỡ bỏ thêm một phần tử khỏi myIntQueue
         Console.WriteLine("\n(Dequeue)\t{0}", myIntQueue.Dequeue());

         // Cho hiển thị lại nội dung của myIntQueue sau Dequeue
         Console.Write("Trị của myIntQueue: \t");
         PrintValues(myIntQueue);

         // Nhìn lén phần tử đầu tiên trên myIntQueue
         // nhưng không được gỡ bỏ
         Console.WriteLine("\n(Peek) \t{0}", myIntQueue.Peek());

         // Cho hiển thị lại nội dung của myIntQueue sau Peek
         Console.Write("Trị của myIntQueue: \t");
         PrintValues(myIntQueue);
      }  // end Main
```

```
        public static void PrintValues(IEnumerable myCollection)
        {   IEnumerator myEnumerator = myCollection.GetEnumerator();
            while (myEnumerator.MoveNext())
                Console.Write("{0} ", myEnumerator.Current);
            Console.WriteLine();
        }  // end PrintValues
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Trị của myIntQueue: 0 5 10 15 20

(Dequeue) 0
Trị của myIntQueue: 5 10 15 20

(Dequeue) 5
Trị của myIntQueue: 10 15 20

(Peek) 10
Trị của myIntQueue: 10 15 20

Chắc bạn hiểu thấu ý nghĩa chương trình này, khỏi giải thích chi thêm. Vì lớp **Queue** là enumerable, nên bạn có thể trao nó qua cho hàm hành sự **PrintValues()**, được cung cấp như là một giao diện **IEnumerable**. Việc chuyển đổi (conversion) được xem như là hiểu ngầm. Trong hàm **PrintValues()**, bạn triệu gọi **GetEnumerator()**, như bạn đã biết là hàm duy nhất của tất cả các lớp **IEnumerable**. Như vậy bạn nhận về một enumerator, **myEnumerator**, cho phép bạn liệt kê ra tất cả các đối tượng trên collection.

10.5 Cái ngăn chồng (Stacks)

Stack, mà chúng tôi tạm dịch là "cái ngăn chồng" (một cái ngăn mà các mục tin được chồng lên nhau), là một collection kiểu LIFO (last-in-first-out), nghĩa là vào sau nhưng muốn được phục vụ trước (một kiểu mà nhiều người thích xử sự khi ở nơi công cộng), giống như một chồng đĩa trong bếp tiệm ăn. Người rửa lau đĩa xong đặt đĩa lên đầu chồng, còn người đầu bếp thì lấy đĩa đầu tiên trên chồng.

Các hàm hành sự chủ chốt dùng thêm vào hoặc rút khỏi stack là **Push** (ấn xuống) và **Pop()** (tống ra).

Sau đây là bảng liệt kê các hàm hành sự và thuộc tính của **Stack:**

Public Properties	
Count	Trả về số phần tử hiện được trữ trên **Stack**.
IsSynchronized	Trả về một trị bool cho biết liệu xem việc truy xuất **Stack** có đồng bộ hay không (thread-safe).
SyncRoot	Trả về một đối tượng có thể được dùng đồng bộ hoá việc truy xuất **Stack**.
Public Methods	
Clear	Gỡ bỏ tất cả các phần tử từ **Stack**.
Clone	Tạo một bản sao shallow của **Stack**.
Contains	Xác định liệu xem một phần tử có nằm trên **Stack** hay không.
CopyTo	Sao các phần tử của **Stack** hoặc một phần của nó lên bản dãy một chiều hiện hữu khởi đi từ một chỉ số chỉ định.
Equals (kế thừa từ Object)	Overloaded. Xác định liệu xem hai thể hiện Object có bằng nhau hay không.
GetEnumerator	Trả về một enumerator cho phép rảo qua hàng nối đuôi **Stack**.
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng cho việc sử dụng trong các giải thuật băm và các cấu trúc dữ liệu giống như một hash table.
GetType (kế thừa từ Object)	Trả về kiểu dữ liệu Type của thể hiện hiện hành.
Peek	Trả đối tượng về đầu hàng nối đuôi **Stack** không gỡ bỏ nó đi.
Pop	Gỡ bỏ và trả về một đối tượng về đầu hàng của **Stack**.
Push	Thêm một đối tượng vào đầu hàng của **Stack**.
Synchronized	Trả về một **Stack** wrapper được đồng bộ hóa (thread-safe).
ToArray	Sao các phần tử của **Stack** qua một bản dãy mới.
ToString (kế thừa từ Object)	Trả về một String tượng trưng cho Object hiện hành.

Hầu hết các hàm hành sự và thuộc tính của **Stack** đều giống như của **Queue**.

Các lớp **ArrayList, Queue,** và **Stack** đều có hàm hành sự overloaded **CopyTo()** và **ToArray(),** dùng sao những phần tử collection về một bản dãy. Trong trường hợp **Stack()**, hàm **CopyTo()** sẽ chép những phần tử của stack lên một bản dãy một chiều hiện hữu, viết đè lên nội dung của bản dãy bắt đầu đi từ chỉ mục được chỉ định. Hàm

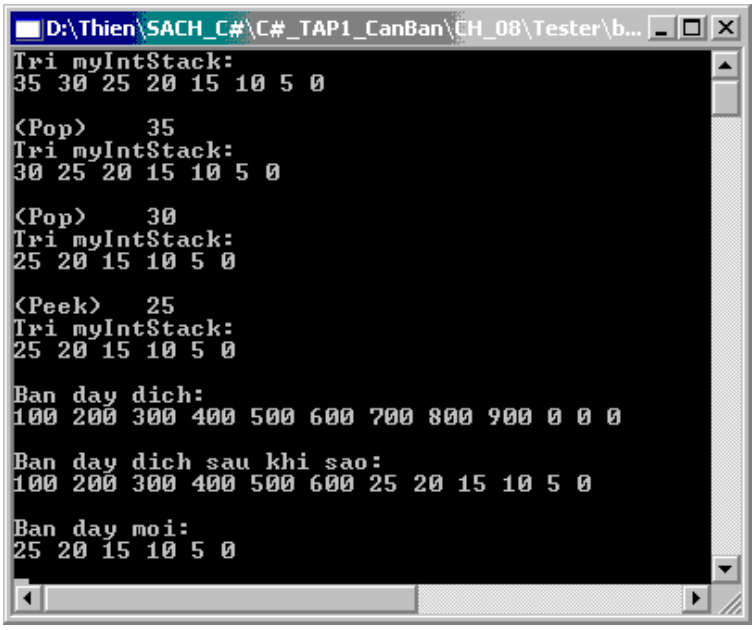
ToArray() trả về một bản dãy mới với nội dung lấy từ các phần tử stack. Thí dụ 10-22 minh hoạ việc sử dụng một stack.

Thí dụ 10-22: Sử dụng một Stack thế nào?

```
namespace Prog_CSharp
{   using System;
    using System.Collections;

    public class Tester
    {   static void Main()
        {   // Tạo và khởi gán một stack, myIntStack
            Stack myIntStack = new Stack();
            for (int i = 0; i < 8; i++)
            {   myIntStack.Push(i*5);
            }// end for

            // Cho hiển thị nội dung của myIntStack sau Push
            Console.WriteLine("Tri myIntStack: \t");
            PrintValues(myIntStack);

            // Tống một phần tử khỏi myIntStack
            Console.WriteLine("\n(Pop) \t {0}", myIntStack.Pop());


            // Cho hiển thị nội dung của myIntStack sau Pop
            Console.WriteLine("Tri myIntStack: \t");
            PrintValues(myIntStack);

            // Tống một phần tử thứ hai khỏi myIntStack
            Console.WriteLine("\n(Pop) \t {0}", myIntStack.Pop());


            // Cho hiển thị nội dung của myIntStack sau Pop
            Console.WriteLine("Trị myIntStack: \t");
            PrintValues(myIntStack);

            // Xem lén phần tử đầu tiên, nhưng không
            // gỡ bỏ khỏi myIntStack sau Peek
            Console.WriteLine("\n(Peek) \t {0}", myIntStack.Peek());


            // Cho hiển thị nội dung của myIntStack
            Console.WriteLine("Trị myIntStack: \t");
            PrintValues(myIntStack);

            // Khai báo một bản dãy đích, myTargetArray, trữ 12 số nguyên
            Array myTargetArray = Array.CreateInstance(typeof(int), 12);
            myTargetArray.SetValue(100, 0);  // cho gán trị
            myTargetArray.SetValue(200, 1);
            myTargetArray.SetValue(300, 2);
            myTargetArray.SetValue(400, 3);
            myTargetArray.SetValue(500, 4);
            myTargetArray.SetValue(600, 5);
```

```
            myTargetArray.SetValue(700, 6);
            myTargetArray.SetValue(800, 7);
            myTargetArray.SetValue(900, 8);

            // cho hiển thị trị của myTargetArray
            Console.WriteLine("\nBản dãy đích: ");
            PrintValues(myTargetArray);

            // Chép nội dung của myIntStack lên myTargetArray,
            // đi từ chỉ số 6
            myIntStack.CopyTo(myTargetArray, 6);

            // cho hiển thị trị của myTargetArray sau khi sao
            Console.WriteLine("\nBản dãy đích sau khi sao: ");
            PrintValues(myTargetArray);

            // Chép toàn bộ nội dung myIntStack lên một bản dãy chuẩn
            Object[] myStdArray = myIntStack.ToArray();

            // cho hiển thị trị của myStdArray sau khi sao
            Console.WriteLine("\nBản dãy mới: ");
            PrintValues(myStdArray);
        }  // end Main

        public static void PrintValues(IEnumerable myCollection)
        {  IEnumerator myEnumerator = myCollection.GetEnumerator();
            while (myEnumerator.MoveNext())
                Console.Write("{0} ", myEnumerator.Current);
            Console.WriteLine();
        }  // end PrintValues
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất



Bạn để ý kết xuất các phần tử bị ấn vào myIntStack phản ảnh thứ tự ngược lại, đây là bản chất của kiểu LIFO.

Trong thí dụ này sử dụng hàm hành sự static **CreateInstance()** để tạo một bản dãy 12 số nguyên. Hàm này nhận 2 đối mục: một kiểu dữ liệu (ở đây là int), và con số cho biết kích thước bản dãy. Ngoài ra, hàm hành sự **SetValue()** cho điền dữ liệu vào bản dãy cũng nhận hai đối mục: đối tượng được thêm vào, và di số (offset) thêm vào từ đó. Cả hai hàm này thuộc lớp Array.

Bạn để ý hàm **ToArray()** dùng trả về một bản dãy các đối tượng, do đó **myStdArray** được khai báo một cách thích ứng:

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

10.6 Collection tự điển (Dictionary)

Một ***dictionary*** là một collection gắn liền một ***mục khoá*** (key) với một ***trị*** (value), do đó thường gọi cặp "key-value". Một ngôn ngữ tự điển thường gắn liền một từ (là mục khoá) với một định nghĩa (là trị). Do đó có từ dictionary cho loại kiểu dữ liệu collection này.

Muốn biết ý nghĩa của dictionary, bạn thử tưởng tượng lập một danh sách tên thủ đô các nước. Cách tiếp cận thứ nhất là đưa chúng vào một bản dãy:

```
string[] stateCapitals = new string[160];
```

stateCapitals có thể chứa tối đa 160 tên thủ đô. Ta chỉ có thể truy xuất tên mỗi thủ đô thông qua một di số (offset). Thí dụ, muốn truy xuất tên thủ đô Việt Nam, ta phải biết Hà Nội là thủ đô thứ 70 (chẳng hạn) theo thứ tự ABC:

```
string capOfVietNam = stateCapitals[69];
```

Tuy nhiên, điều tiện lợi để truy xuất tên thủ đô một quốc gia là sử dụng ký hiệu bản dãy. Nói cho cùng, không có cách gì dễ dàng để xác định, Thái Lan là quốc gia thứ 120 chẳng hạn, theo thứ tự ABC. Do đó, tiện nhất là cho trữ tên thủ đô đi kèm theo tên quốc gia. Trong kiểu dữ liệu dictionary, ta được phép trữ một trị (trong trường hợp này là tên thủ đô) với một mục khoá (ở đây là tên quốc gia).

Dictionary của .NET Framework có thể gắn liền bất cứ kiểu dữ liệu nào (string, integer, object, v.v.) cho mục khoá với bất cứ loại trị nào (string, integer, object, v.v.). Điển hình, mục khoá phải ngắn gọn, còn trị tương ứng là phức tạp.

Đặc tính quan trọng của một dictionary tốt là có thể thêm dễ dàng những trị và có thể truy xuất nhanh chóng. Một số dictionary chạy nhanh khi thêm trị mới, một số khác tối ưu hoá việc truy xuất. Một thí dụ về dictionary là hash table (bảng băm).

10.6.1 Bảng băm (Hashtables)

Một ***hashtable*** là một dictionary được tối ưu hoá về mặt truy xuất. Sau đây chúng tôi cho liệt kê các hàm hành sự và thuộc tính của **Hashtable**.

Public Properties

Count	Trả về con số của những cặp key-value được trữ trong **Hashtable**.
IsFixedSize	Trả về một trị cho biết liệu xem **Hashtable** có thể cố định hay không .
IsReadOnly	Trả về một trị cho biết liệu xem **Hashtable** thuộc loại read-only hay không
IsSynchronized	Trả về một trị cho biết liệu xem việc truy xuất **Hashtable** có được đồng bộ hoá hay không (thread-safe).
Item	Trả về/Đặt để trị được gắn liền với mục khóa chỉ định. Trên C#, thuộc tính này là indexer đối với lớp **Hashtable**.

Keys	Trả về một giao diện ICollection chứa các mục khóa trên **Hashtable**.
SyncRoot	Trả về một đối tượng mà ta có thể dùng để đồng bộ hoá truy xuất **Hashtable**.
Values	Trả về một giao diện ICollection chứa những trị trên **Hashtable**.
Public Methods	
Add	Thêm một phần tử với mục khoá chỉ định và trị đưa vào **Hashtable**.
Clear	Gỡ bỏ tất cả các phần tử khỏi **Hashtable**.
Clone	Tạo một bản sao shallow của **Hashtable**.
Contains	Xác định liệu xem **Hashtable** có chứa một mục khóa chỉ định.
ContainsKey	Xác định liệu xem **Hashtable** có chứa một mục khóa chỉ định.
ContainsValue	Xác định liệu xem **Hashtable** có chứa một trị chỉ định.
CopyTo	Sao các phần tử của **Hashtable** về một một bản dãy một chiều thể hiện Array đi từ chỉ mục chỉ định.
Equals (kế thừa từ Object)	Overloaded. Xác định liệu xem hai thể hiện Object có bằng nhau không .
GetEnumerator	Trả về một giao diện IDictionaryEnumerator cho phép rảo qua **Hashtable**.
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng cho việc sử dụng trong các giải thuật băm và các cấu trúc dữ liệu giống như một hash table.
GetObjectData	Cho thiết đặt giao diện ISerializable và trả về dữ liệu cần để serialize **Hashtable**.
GetType (kế thừa từ Object)	Trả về kiểu dữ liệu Type của thể hiện hiện hành .
OnDeserialization	Cho thiết đặt giao diện ISerializable và phát đi tình huống deserialization khi deserialization hoàn tất.
Remove	Gỡ bỏ phần tử mang mục khoá chỉ định khỏi **Hashtable**.
Synchronized	Trả về một "vỏ bọc" được đồng bộ hoá (synchronized wrapper - thread-safe) đối với **Hashtable**.
ToString (kế thừa từ Object)	Trả về một String tượng trưng cho Object hiện hành.

Trên một **Hashtable**, mỗi trị sẽ được trữ trong một "xô" (bucket, giống như xô nước), và xô này được đánh số tương tự như một di số trên một một bản dãy.

Vì mục khóa có thể không phải là một số nguyên, nên ta phải có khả năng dịch mục khóa (nghĩa là Viêt Nam chẳng hạn) thành số xô. Mỗi mục khóa phải cung cấp một hàm hành sự **GetHashCode()** lo phần dịch này.

Chắc bạn đã biết mọi thứ đều được dẫn xuất từ **Object**. Lớp **Object** cung cấp một hàm hành sự virtual **GetHashCode()**, mà các kiểu dữ liệu được dẫn xuất tự do kế thừa hoặc cho phủ quyết (overriden).

Khi bạn đưa những trị (tên thủ dô) vào **Hashtable**, thì **Hashtable** triệu gọi **GetHash Code()** trên mỗi mục khóa ta cung cấp. Hàm này trả về một **int** nhận diện mã số xô trữ tên thủ đô.

Lẽ dĩ nhiên, có khả năng là nhiều mục khóa cùng cho về một số xô. Ta gọi đây là ***collision*** (đụng độ). Có nhiều cách giải quyết một cuộc đụng độ. Cách thông dụng nhất, mà CLR chấp nhận, là cho mỗi xô duy trì một danh sách các trị được sắp xếp thứ tự.

Khi bạn muốn tìm lại một trị từ **Hashtable**, bạn phải cung cấp một mục khóa. Một lần nữa, **Hashtable** triệu gọi **GetHashCode()** dựa trên mục khóa và dùng số nguyên **int** để đi tìm xô thích ứng**.** Nếu chỉ có một trị, thì trị này được trả về, bằng không một cuộc truy lùng kiểu nhị phân sẽ được thực hiện. Vì không có bao nhiêu trị nên việc truy tìm sẽ rất nhanh.

Mục khóa có thể thuộc loại "bẩm sinh", hoặc có thể là loại tự tạo (user defined) (một object). Các đối tượng tự tạo dùng làm mục khóa đối với **Hashtable** cũng phải thiết đặt **GetCashCode()** cũng như **Equals()**. Trong đa số trường hợp, bạn chỉ cần sử dụng thiết đặt được kế thừa từ **Object**.

10.6.2 Giao diện IDictionary

Hashtable là những dictionary vì chúng thiết đặt giao diện **IDictionary**. Giao diện này cung cấp một thuộc tính public mang tên **Item**. Thuộc tính **Item** tìm lại trị dựa trên mục khóa được khai báo. Trên C#, việc khai báo thuộc tính Item diễn ra như sau:

```
object this[object key] {get; set;}
```

Thuộc tính **Item** được thiết đặt trên C# sử dụng tác tử chỉ mục []. Do đó, bạn truy xuất mọi mục tin (item) trên bất cứ đối tượng **Dictionary** nào sử dụng cú pháp offset, như với một bản dãy.

Thí dụ 10-23 minh hoạ việc thêm các mục tin vào **Hashtable** rồi sau đó cho tìm lại các mục tin này thông qua thuộc tính **Item**.

Thí dụ 10-23: Thuộc tính Item dùng như tác tử di số (offset operator)

```
namespace Prog_CSharp
{   using System;
```

```
   using System.Collections;

   public class Tester
   {  static void Main()
      {  // Tạo và khởi gán một Hashtable mới
         Hashtable myHashTable = new Hashtable();
         myHashTable.Add("000440312", "Đoàn Dự");
         myHashTable.Add("000123933", "Tiêu Phong");
         myHashTable.Add("000145938", "Dương Khang");
         myHashTable.Add("000773394", "Quách Tĩnh");

         // truy xuất một mục tin đặc biệt nào đó
         Console.WriteLine("myHashTable["000145938"]: {0}",
            myHashTable["000145938"];
      }  // end Main
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

myHashTable[000145938]: Dương Khang

Thí dụ 10-23 bắt đầu thể hiện một **Hashtable** mới, myHashTable. Chúng tôi sử dụng hàm constructor đơn giản nhất chấp nhận khả năng ban đầu và load factor (hệ số nạp) mặc nhiên, hash code provider mặc nhiên và comparer mặc nhiên .

Sau đó, ta thêm 4 cặp key/value, mã số bảo hiểm y tế - tên nhân vật Kim Dung (nếu họ còn sống chắc phải có con số này). Một khi các mục tin được đưa vào, bạn có thể truy xuất mục tin thứ 3 sử dụng mục khóa của nó.

10.6.3 Các collection mục khóa và trị

Collection **Dictionary** còn bổ sung hai thuộc tính **Keys** và **Values**. Thuộc tính **Keys** cho tìm lại một đối tượng **ICollection** với tất cả các mục khóa trong **Hashtable**, cũng như thuộc tính **Values** thì lại cho tìm một đối tượng **ICollection** với tất cả các trị. Thí dụ 10-24, minh hoạ việc sử dụng hai collection **Keys** và **Values** này.

Thí dụ 10-24: Sử dụng hai collection Keys và Values

```
namespace Prog_CSharp
{  using System;
   using System.Collections;

   public class Tester
   {  static void Main()
      {  // Tạo và khởi gán một Hashtable mới
         Hashtable myHashTable = new Hashtable();
```

```
            myHashTable.Add("000440312", "Đoàn Dự");
            myHashTable.Add("000123933", "Tiêu Phong");
            myHashTable.Add("000145938", "Dương Khanh");
            myHashTable.Add("000773394", "Quách Tĩnh");

            // đi lấy những mục khóa và trị từ myHashTable
            ICollection myKeys = myHashTable.Keys;
            ICollection myValues = myHashTable.Values;

            // Cho rảo qua myKeys collection và myValues collection
            foreach(string key in myKeys)
            {  Console.WriteLine("{0} ", key);
            }

            foreach(string val in myValues)
            {  Console.WriteLine("{0} ", val);
            }
         }  // end Main
      }  // end Tester
   }  // end Prog_CSharp
```

Kết xuất

000440312
000123933
000145938
000773394
Đoàn Dự
Tiêu Phong
Dương Khang
Quách Tĩnh

10.6.4 Giao diện IDictionayEnumerator

Các đối tượng **IDictionary** cũng hỗ trợ **foreach** bằng cách thiết đặt hàm hành sự **GetEnumerator** cho trả về một **IDictionaryEnumerator**. Giao diện này dùng để rảo qua bất cứ đối tượng **IDictionary** nào, vì giao diện cung cấp 3 thuộc tính (**Entry, Key** và **Value)** cho phép cả mục khóa lẫn trị đối với mỗi mục tin trên dictionary. Thí dụ 10-25 minh hoạ việc sử dụng giao diện IDictionaryEnumerator này.

Thí dụ 10-25: Cách dùng giao diện IDictionaryEnumerator thế nào.

```
namespace Prog_CSharp
{  using System;
   using System.Collections;
```

```
   public class Tester
   {  static void Main()
      {  // Tạo và khởi gán một Hashtable mới
         Hashtable myHashTable = new Hashtable();
         myHashTable.Add("000440312", "Đoàn Dự");
         myHashTable.Add("000123933", "Tiêu Phong");
         myHashTable.Add("000145938", "Dương Khang");
         myHashTable.Add("000773394", "Quách Tĩnh");

         // Cho hiển thị các thuộc tính và trị của myHashTable
         Console.WriteLine("myHashTable");
         Console.WriteLine(" Count:  {0}", myHashTable.Count);
         Console.WriteLine(" Mục khoá và Trị:");
         PrintKeysAndValues(myHashTable);
      }  // end Main

      public static void PrintKeysAndValues(HashTable table)
      {  IDictionaryEnumerator myEnumerator = table.GetEnumerator();
         while (myEnumerator.MoveNext())
            Console.WriteLine("\t{0}:\t{1}", myEnumerator.Key,
               myEnumerator.Value);
         Console.WriteLine();
      }  // end Print…
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

myHashTable
Count: 4
Mục khoá và Trị
    000440312: Đoàn Dự
    000123933: Tiêu Phong
    000145938: Dương Khang
    000773394: Quách Tĩnh

Chương 11

Chuỗi chữ và Biểu thức regular

Có thời người ta nghĩ rằng máy điện toán chỉ toàn xử lý những trị số. Vào thời kỳ đầu máy điện toán dùng ưu tiên để tính ra hành trình các hoả tiễn, và ngành lập trình được dạy trong bộ môn toán ở các đại học (ở VN ta, vẫn còn vô số người đinh ninh cho rằng muốn học vi tính phải rành toán. Thật ra điều ấy không đúng. Người viết rất dốt toán).

Ngày nay, phần lớn các chương trình đều dính dáng nhiều đến các chuỗi chữ (string of characters) hơn là các con số. Điển hình là những chuỗi chữ thường được dùng trong việc xử lý văn bản, trong thao tác tài liệu, cũng như tạo những trang web.

C# cung cấp việc hỗ trợ bẩm sinh đối với toàn bộ chức năng kiểu **string**. Quan trọng nhất là C# coi chuỗi chữ như là những đối tượng gói ghém (encapsulating) tất cả các hàm hành sự xử lý**,** sắp xếp và truy tìm (searching) đối với các chuỗi chữ.

Việc xử lý chuỗi chữ phức tạp và so khớp mẫu dáng (pattern) được hỗ trợ bằng cách dùng những biểu thức được gọi là ***regular expression***[43]**.** C# phối hợp sức mạnh và sự phức tạp của cú pháp regular expression, mà ta thường thấy xử lý trong các ngôn ngữ xử lý chuỗi chữ như **awk** và **Perl** chẳng hạn, với thiết kế hoàn toàn thiên đối tượng.

Trong chương này, bạn sẽ học làm việc với kiểu dữ liệu C# **string**, cũng như với lớp **System.String** của.NET Framework. Bạn sẽ thấy làm thế nào để truy tìm và trích các chuỗi con, thao tác và ghép chuỗi (concatenate), và tạo những chuỗi mới thông qua lớp **StringBuilder**. Ngoài ra, bạn sẽ học cách sử dụng lớp **RegEx** (tắt chữ Regular Expression) để so khớp (match) các chuỗi dựa trên biểu thức regular phức tạp.

11.1 Chuỗi chữ

C# coi chuỗi chữ như là kiểu dữ liệu cao cấp rất uyển chuyển, cực mạnh và dễ dùng. Mỗi đối tượng **string** là một loạt những ký tự Unicode "bất di bất dịch" (immutable)

[43] Tác giả chịu thua chưa dịch được từ này. Các bạn có thể cho ý kiến.

nghĩa là trị nó không thể bị thay đổi một khi dã được tạo ra. Nói cách khác, các hàm hành sự xem như thay đổi một đối tượng **String** nhưng thật ra trả về một **String** mới, nghĩa là một bản sao bị thay đổi, còn bản chính vẫn còn nguyên vẹn. Bạn sử dụng lớp **System.Text.StringBuilder** nếu thấy cần thiết phải thay đổi nội dung của một đối tượng giống chuỗi.

Khi bạn khai báo một chuỗi chữ, C# sử dụng từ chốt **string**, thật ra bạn đang khai báo đối tượng thuộc kiểu **System.String**, một trong những kiểu dữ liệu bẩm sinh cung cấp bởi.NET Framework Class Library. Một kiểu dữ liệu C# **string** là một kiểu dữ liệu **System.String**.

Khai báo lớp **System.String** như sau:

```
public sealed class String: IComparable, ICloneable, IConvertible,
       IEnumerable
```

Khai báo trên cho thấy lớp **String** thuộc loại vô sinh, nghĩa là ta không thể dẫn xuất từ lớp **string**. Ngoài ra, **string** thiết đặt 4 giao diện **IComparable, ICloneable, IConvertible** và **IEnumerable**, cho thấy **System.String** chia sẻ chức năng với các lớp khác trên.NET Framework.

Giao diện **IComparable** được thiết đặt bởi những kiểu dữ liệu mà trị cần được sắp xếp, những lớp số hoặc chuỗi chữ. Thí dụ, chuỗi chữ có thể được sắp xếp theo thứ tự ABC; bất cứ một chuỗi nào cũng có thể đem so sánh với một chuỗi chữ khác để xác định chuỗi nào nằm trước trên một danh sách đã theo thứ tự. **IComparable** thiết đặt hàm hành sự **CompareTo()**.

Giao diện **ICloneable** hỗ trợ việc tạo những thể hiện mới cùng trị với thể hiện hiện hành. Trong trường hợp này, có khả năng "nhân bản" (clone) một chuỗi chữ tạo ra một chuỗi mới mang cùng trị (ký tự) như nguyên bản. **ICloneable** thiết đặt hàm hành sự **Clone()**.

Giao diện **IConvertible** cung cấp những hàm hành sự cho phép chuyển đổi trị của thiết đặt dữ liệu kiểu trị hoặc qui chiếu về những kiểu dữ liệu sơ đẳng khác như **ToInt32(), ToDouble(), ToDecimal()** chẳng hạn.

Giao diện **IEnumerable** cung cấp một enumerator, cho phép bạn rảo qua trên một collection.

11.1.1 Tạo những chuỗi chữ

Thể thức thông dụng nhất để tạo ra một chuỗi chữ là gán một chuỗi chữ được đặt nằm trong hai dấu nháy “”, được biết đến như là một ***string literal*** (trực kiện chuỗi), về một biến tự tạo (user-defined variable) kiểu **string**:

```
string newString = "Đây là một chuỗi trực kiện";
```

Trong chuỗi trực kiện này ta có thể cho bao gồm các ký tự escape, chẳng hạn “\n” hoặc “\t”, bắt đầu bởi dấu “\” (backslash), cho biết khi nào nhảy hàng hoặc nhảy canh cột (tab). Nếu backslash được dùng trong cú pháp các dòng lệnh, chẳng hạn URL hoặc lối tìm về thư mục (directory path), thì ta phải tăng đôi backslash cho mỗi backslash.

Chuỗi cũng có thể được tạo ra sử dụng chuỗi trực kiện *đúng nguyên văn* (verbatim string literal), bắt đầu bởi dấu a vòng @. Dấu này cho hàm constructor của **String** biết chuỗi chữ phải sử dụng verbatim, cho dù chuỗi trải dài trên nhiều hàng hoặc bao gồm những ký tự escape. Trong một chuỗi trực kiện verbatim, backslash và những ký tự theo sau được xem như là những ký tự bổ sung vào chuỗi. Do đó, hai khai báo sau đây xem như tương đương:

```
string literalOne = "\\\\MySystem\\MyDirectory\\ProgrammingC#.cs";
string verbatimLiteralOne =
                        @"\\MySystem\MyDirectory\ProgrammingC#.cs";
```

Trên hàng đầu tiên, một chuỗi trực kiện nonverbatim được khai báo, do đó backslash là một ký tự escape, nên phải viết tăng đôi. Còn trên hàng verbatim thì backslash phụ thêm khỏi cần đến. Thí dụ thứ hai dưới đây minh họa chuỗi verbatim trải dài trên nhiều hàng:

```
string literalTwo = "Line One\nLine Two";
string verbatimliteralTwo = @"Line One
Line Two";
```

Bạn có thể chọn sử dụng theo tùy thích một trong hai cách viết kể trên.

11.1.2 Hàm hành sự ToString()

Một thể thức thông dụng khác để tạo một chuỗi là cho triệu gọi hàm **ToString** tác động lên một đối tượng rồi gán kết quả cho một biến chuỗi. Tất cả các kiểu dữ liệu bẩm sinh đều phủ quyết hàm này để đơn giản hoá việc chuyển đổi một trị (thường là một trị số) thành một biểu diễn chuỗi chữ của trị này. Thí dụ sau đây cho thấy hàm **ToString** của một kiểu dữ liệu số nguyên được gọi vào để trữ trị nó trên một chuỗi chữ:

```
int myInteger = 5;
string myIntString = myInteger.ToString();
```

Việc triệu gọi **myInteger.ToString()** trả về một đối tượng **String**, để sau đó được đem gán cho **myIntString,** kiểu string.

11.1.3 Hàm constructor kiểu String

Lớp.NET **String** cung cấp một lô hàm constructor được phủ quyết hỗ trợ những kỹ thuật khác nhau để gán những trị chuỗi lên các kiểu dữ liệu **string**. Một vài hàm constructor cho phép bạn tạo một chuỗi bằng cách trao một bản dãy ký tự hoặc con trỏ ký tự (character pointer). Trao một bản dãy ký tự như là một thông số cho hàm constructor của **String** sẽ tạo một thể hiện mới của một chuỗi "ăn ý" với CLR (CLR-compliant). Còn trao một con trỏ ký tự sẽ tạo một thể hiện noncompliant, không an toàn (unsafe). Sau đây liệt kê các hàm constructor được nạp chồng (overloaded) lo khởi gán một thể hiện mới của lớp **String**:

- **unsafe public String(char*);**
  Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một con trỏ được chỉ định về một bản dãy ký tự Unicode. Hàm constructor này là không CLS-compliant.

- **public String(char[]);**
  Khởi gán một thể hiện mới của lớp String về trị được chỉ định bởi một bản dãy ký tự Unicode.

- **unsafe public String(sbyte*);**
  Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một con trỏ về một bản dãy số nguyên có dấu 8-bit. Hàm constructor này là không CLS-compliant.

- **public String(char, int);**
  Khởi gán một thể hiện mới của lớp String về trị được chỉ định bởi một ký tự Unicode **char** được khai báo lặp lại một số lần **int**.

- **unsafe public String(char*, int, int);**
  Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một con trỏ chỉ về một bản dãy ký tự Unicode **char***, với một vị trí ký tự khởi **int** trong lòng bản dãy này, và một chiều dài **int**. Hàm constructor này là không CLS-compliant.

- **public String(char[], int, int);**
  Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một bản dãy ký tự Unicode **char[]**, với một vị trí ký tự khởi đi **int** trong lòng bản dãy này, và một chiều dài **int**.

- **unsafe public String(sbyte*, int, int);**

Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một con trỏ chỉ về một bản dãy số nguyên 8-bit **sbyte***, với một vị trí ký tự khởi đi **int** trong lòng bản dãy này, và một chiều dài **int**. Hàm constructor này là không CLS-compliant.

- **unsafe public String(sbyte*, int, int, Encoding);**
  Khởi gán một thể hiện mới của lớp String về trị được khai báo bởi một con trỏ chỉ về một bản dãy số nguyên 8-bit **sbyte***, với một vị trí ký tự khởi **int** trong lòng bản dãy này, một chiều dài **int**, và một đối tượng **Encoding.** Hàm constructor này là không CLS-compliant.

11.1.4 Thao tác trên các chuỗi

Lớp **String** cung cấp cho bạn vô số hàm hành sự cho phép bạn tiến hành những so sánh, truy tìm và thao tác trên các chuỗi chữ, như theo bảng liệt kê dưới đây:

Public Fields	
Empty	Tượng trưng cho một chuỗi rỗng. Vùng mục tin này mang tính read-only.
Public Properties	
Chars	Đi lấy ký tự nằm tại vị trí ký tự chỉ định trên chuỗi này. Trên C#, thuộc tính này là indexer đối với lớp **String**.
Length	Đi lấy số ký tự trong chuỗi này.
Public Methods	
Clone	Trả về một qui chiếu về chuỗi này của **String**.
Compare	Overloaded. So sánh hai đối tượng **String** được chỉ định.
CompareOrdinal	Overloaded. So sánh hai đối tượng **String,** không kể đến ngôn ngữ bản địa hoặc văn hoá.
CompareTo	Overloaded. So sánh chuỗi này với một đối tượng được chỉ định.
Concat	Overloaded. Ghép một hoặc nhiều chuỗi của **String**, hoặc biểu diễn trị **String** của một hoặc nhiều chuỗi của Object.
Copy	Tạo một chuỗi mới của **String** mang cùng trị như **String** được chỉ định.
CopyTo	Trên chuỗi này, sao một số ký tự được chỉ định từ một vị trí được chỉ định về một vị trí được chỉ định trên một bản dãy ký tự Unicode.
EndsWith	Xác định liệu xem phần cuối của chuỗi này khớp với **String** được chỉ định hay không.
Equals	Overloaded. Overridden. Xác định liệu xem 2 đối tượng **String** có mang cùng trị hay không.
Format	Overloaded. Thay thế mỗi đặc tả định dạng (format

	specification) trên một **String** được chỉ định với văn bản tương đương của một trị đối tượng tương ứng.
GetEnumerator	Tìm lại một đối tượng mà ta có thể rảo qua ký tự riêng rẽ trên chuỗi này.
GetHashCode	Overridden. Trả về mã băm (hash code) đối với chuỗi này.
GetType (kế thừa từ Object)	Đi lấy Type của chuỗi hiện hành.
GetTypeCode	Trả về TypeCode đối với lớp **String**.
IndexOf	Overloaded. Báo cáo cho biết chỉ mục (index) của xuất hiện (occurrence) đầu tiên của một **String**, hoặc một hoặc nhiều ký tự, trong lòng chuỗi này.
IndexOfAny	Overloaded. Báo cáo cho biết index của xuất hiện đầu tiên trên chuỗi này của bất cứ ký tự nào trên một bản dãy ký tự Unicode được chỉ định.
Insert	Trả về một chuỗi mới với chuỗi chỉ định được chèn vào từ một vị trí index được chỉ định.
Intern	Trả về một qui chiếu hệ thống chỉ về chuỗi được chỉ định của một đối tượng **String**.
IsInterned	Trả về một qui chiếu đối với một đối tượng **String** được chỉ định.
Join	Overloaded. Ghép một chuỗi **String** được chỉ định phân cách giữa mỗi phần tử bản dãy **String** được chỉ định, biến thành một chuỗi duy nhất được ghép.
LastIndexOf	Overloaded. Báo cáo cho biết vị trí index của xuất hiện cuối cùng trên chuỗi này của một ký tự Unicode được chỉ định hoặc của **String**.
LastIndexOfAny	Overloaded. Báo cáo cho biết vị trí index của xuất hiện cuối cùng trên chuỗi này của một hoặc nhiều ký tự được chỉ định trên một bản dãy Unicode.
PadLeft	Overloaded. Canh phải (right-aligns) các ký tự trong chuỗi này, cho điền (padding) về phía trái bởi ký tự trắng (spaces) hoặc bởi một ký tự Unicode được chỉ định, theo tổng chiều dài được chỉ định.
PadRight	Overloaded. Canh trái (left-aligns) các ký tự trong chuỗi này, cho điền (padding) về phía phải bởi ký tự trắng (spaces) hoặc bởi một ký tự Unicode được chỉ định, theo tổng chiều dài được chỉ định.
Remove	Xoá bỏ một số ký tự được chỉ định khỏi chuỗi này bắt đầu từ một vị trí được chỉ định.
Replace	Overloaded. Thay thế tất cả các xuất hiện của một ký tự Unicode được chỉ định hoặc **String** trong chuỗi này, bởi một

	ký tự Unicode hoặc **String** được chỉ định.
Split	Overloaded. Nhận diện những chuỗi con (substrings) trên chuỗi này được giới hạn bởi một hoặc nhiều ký tự được chỉ định trên một bản dãy, rồi sau đó đưa những chuỗi con này vào một bản dãy kiểu **String**.
StartsWith	Xác định liệu xem khởi đi chuỗi này có khớp với đối tượng **String** được chỉ định hay không.
Substring	Overloaded. Trả về một chuỗi con (substring) từ chuỗi này.
ToCharArray	Overloaded. Chép các ký tự trên chuỗi này lên một bản dãy ký tự Unicode.
ToLower	Overloaded. Trả về một bản sao đối tượng **String** này ghi theo chữ thường (lowercase).
ToString	Overloaded. Overridden. Chuyển đổi (convert) trị của chuỗi này thành một **String**.
ToUpper	Overloaded. Trả về một bản sao đối tượng **String** này viết theo chữ hoa (uppercase).
Trim	Overloaded. Gỡ bỏ tất cả các xuất hiện của một nhóm ký tự được chỉ định khỏi phần đầu và đuôi của chuỗi này.
TrimEnd	Gỡ bỏ tất cả các xuất hiện của một nhóm ký tự được chỉ định trên một bản dãy ký tự Unicode khỏi phần đuôi của chuỗi này.
TrimStart	Gỡ bỏ tất cả các xuất hiện của một nhóm ký tự được chỉ định trên một bản dãy ký tự Unicode khỏi phần đầu của chuỗi này.

Bạn thấy là số thành viên của lớp **String** khá phong phú. Bạn nên vào MSDN của Microsoft tra cứu từng hàm một để xem công dụng của từng hàm cũng như những thí dụ sử dụng của mỗi hàm. Thí dụ 11-1 chỉ minh họa việc sử dụng của một vài hàm, như chẳng hạn **Compare()**, **Concat()**, **Copy()**, **Insert()**, **EndsWith()** và **IndexOf**.

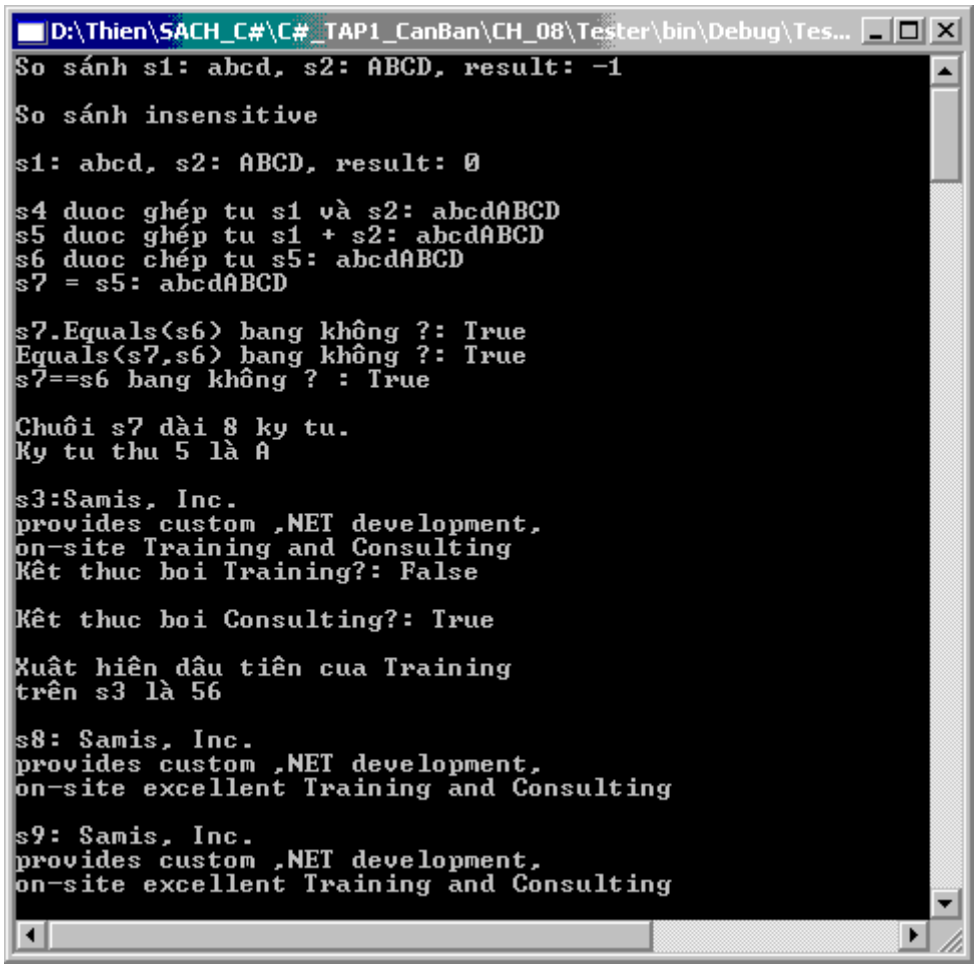
Thí dụ 11-1: Làm việc với các chuỗi.

```
namespace Prog_CSharp
{  using System;
   public class StringTester
   {
      static void Main()
      {  // thử tạo một vài chuỗi để mà_u8220 ?vọc”
         string s1 = "abcd";
         string s2 = "ABCD";
         string s3 = @"Samis, Inc.
                      provides custom ,NET development,
                      on-site Training and Consulting";
         int result;  // để chứa kết quả so sánh

         // so sánh 2 chuỗi, case sensitive
         result = string.Compare(s1, s2);
```

```
Console.WriteLine("So sánh s1: {0}, s2: {1}, result:
                                  {2}\n", s1, s2, result);

// Compare overloaded, để ý đến trị bool "ignore case"
// (true = ignore case, không quan tâm chữ hoa chữ thường)
result = string.Compare(s1, s2, true);
Console.WriteLine("So sánh insensitive\n");
Console.WriteLine("s1: {0}, s2: {1}, result: {2}\n", s1,
                                          s2, result);

// dùng hàm Concat() để ghép chuỗi
string s4 = string.Concat(s1, s2);
Console.WriteLine("s4 được ghép từ s1 và s2: {0}", s4);

// dùng tác tử + được nạp chồng để ghép hai chuỗi
string s5 = s1 + s2;
Console.WriteLine("s5 được ghép từ s1 + s2: {0}", s4);

// dùng hàm sao Copy()
string s6 = string.Copy(s5);
Console.WriteLine("s6 được chép từ s5: {0}", s6);

// dùng tác tử gán để sao chép
string s7 = s5;
Console.WriteLine("s7 = s5: {0}", s7);

// Có 3 cách để so sánh xem có bằng nhau hay không
Console.WriteLine("\ns7.Equals(s6) bằng không ?: {0}",
                    s7.Equals(s6));
Console.WriteLine("Equals(s7,s6) bằng không ?: {0}",
                      string.Equals(s7,s6));
Console.WriteLine("s7==s6 bằng không ?: {0}", s7 == s6);

// Hai thuộc tính hữu ích: index và chiều dài
Console.WriteLine("\nChuỗi s7 dài {0} ký tự.", s7.Length);
Console.WriteLine("Ký tự thứ 5 là {1}\n",s7.Length,s7[4]);

// Trắc nghiệm liệu xem một chuỗi kết thúc với một lô ký tự
Console.WriteLine("s3:{0}\nKết thúc bởi Training?:
                   {1}\n",s3, s3.EndsWith("Training"));
Console.WriteLine("Kết thúc bởi Consulting?:
                        {0}",s3.EndsWith("Consulting"));

// Trả về chỉ mục của chuỗi con
Console.WriteLine("\nXuất hiện đầu tiên của Training ");
Console.WriteLine("trên s3 là {0}\n",
                    s3.IndexOf("Training"));

// Chèn từ "excellent" trước từ "Training"
string s8 = s3.Insert(56, "excellent ");
Console.WriteLine("s8: {0}\n", s8);

   // Bạn có thể phối hợp lại hai hàm Insert
   // và IndexOf như sau:
```

```
                string s9 = s3.Insert(s3.IndexOf("Training"),
                                      "excellent ");
                Console.WriteLine("s9: {0}\n", s9);
                Console.ReadLine();
            }   // end Main
        }   // end StringTester
    }   // end Prog_CSharp
```

Kết xuất



Thí dụ 11-1 bắt đầu bằng một khai báo 3 chuỗi s1, s2 và s3:

```
string s1 = "abcd";
string s2 = "ABCD";
```

```
string s3 = @"SAMIS, Inc.
             provides custom ,NET development,
             on-site Training and Consulting";
```

Hai chuỗi đầu tiên **s1, s2** là những chuỗi trực kiện (literal) thông thường, còn chuỗi thứ ba **s3** là một chuỗi trực kiện verbatim. Ta bắt đầu so sánh **s1** với **s2**. Hàm **Compare()** là một hàm hành sự public static thuộc lớp **string**, và nó bị nạp chồng (overload). Phiên bản nạp chồng đầu tiên, chỉ lấy 2 chuỗi, rồi đem ra so sánh với nhau:

```
// so sánh 2 chuỗi, case sensitive (để ý đến chữ hoa chữ thường)
result = string.Compare(s1, s2);
Console.WriteLine("So sánh s1: {0}, s2: {1}, result: {2}\n",
                  s1, s2, result);
```

Đây là một so sánh có để ý đến chữ hoa chữ thường (case-sensitive), và trả về những trị khác nhau, dựa trên kết quả so sánh: số nguyên âm (-1) nếu chuỗi đầu tiên nhỏ thua chuỗi thứ hai, 0 nếu cả hai chuỗi bằng nhau, số nguyên dương (1) nếu chuỗi đầu tiên lớn hơn chuỗi thứ hai. Trong trường hợp này, **s1** cho thấy nhỏ thua **s2**. Theo mã Unicode, chữ thường mang trị nhỏ thua chữ hoa:

```
So sánh s1: abdc, s2: ABCD, result: -1
```

So sánh thứ hai dùng đến phiên bản nạp chồng của **Compare**, lấy vào một thông số thứ ba, kiểu Bool, cho biết có quan tâm đến chữ hoa chữ thường hay không. Nếu trị thông số này là true có nghĩa là "ignore case", không quan tâm chữ hoa chữ thường, như theo các câu lệnh sau đây:

```
result = string.Compare(s1, s2, true);
Console.WriteLine("So sánh insensitive\n");
Console.WriteLine("s1: {0}, s2: {1}, result: {2}\n", s1, s2, result);
```

Lần này, không quan tâm đến chữ hoa, chữ thường, nên **result** mang trị zero, cho biết hai chuỗi giống nhau:

```
So sánh insensitive

s1: abdc, s2: ABCD, result: 0
```

Thí dụ 11-1, sau đó cho ghép (concatenate) một số chuỗi. Có nhiều cách để thực hiện điều này. Bạn có thể sử dụng hàm hành sự **Concat()**, là một hàm public static của lớp **string**:

```
string s4 = string.Concat(s1, s2);
Console.WriteLine("s4 được ghép từ s1 và s2: {0}", s4);
```

hoặc đơn giản dùng tác tử ghép được nạp chồng (+):

```
string s4 = s1 + s2;
Console.WriteLine("s4 được ghép từ s1 + s2: {0}", s4);
```

Trong cả hai trường hợp, kết xuất phản ảnh việc ghép chuỗi thành công:

```
s4 được ghép từ s1 và s2: abcdABCD
s5 được ghép từ s1 + s2: abcdABCD
```

Cũng tương tự như thế, tạo một bản sao (copy) mới một chuỗi có thể được thực hiện theo hai cách. Trước tiên, bạn dùng hàm static **Copy()**, sau đó nếu muốn bạn sử dụng tác tử gán (=) được nạp chồng:

```
string s6 = string.Copy(s5);
Console.WriteLine("s6 được chép từ s5: {0}", s6);
```

hoặc

```
string s7 = s5;
Console.WriteLine("s7 = s5: {0}, s7);
```

Một lần nữa, kết xuất phản ảnh hai thể thức sao chép hoạt động giống nhau:

```
s6 được chép từ s5: abcdABCD
s7 = s5: abcdABCD
```

Lớp.NET **String** cung cấp 3 thể thức để trắc nghiệm sự bằng nhau (equality) của hai chuỗi. Trước tiên, bạn dùng hàm overloaded **Equals()** hỏi trực tiếp **s6** liệu xem **s5** có mang trị bằng nhau hay không:

```
Console.WriteLine("\ns7.Equals(s6) bằng không ?: {0}",
                  s7.Equals(s6));
```

Thể thức thứ hai là trao cả hai chuỗi cho hàm static **Equals()**:

```
Console.WriteLine("Equals(s7,s6) bằng không ?: {0}",
                 string.Equals(s7,s6));
```

Và thể thức cuối cùng là sử dụng tác tử equality (==) được nạp chồng của **String**:

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

Cả 3 thể thức trên đều trả về trị bool **true**, như theo kết xuất sau đây:

```
s7.Equals(s6) bằng không ?: True
Equals(s7,s6) bằng không ?: True
s7==s6 bằng không ?: True
```

Nhiều hàng kế tiếp trên thí dụ 11-1 sử dụng tác tử chỉ mục [] để tìm ra một ký tự đặc biệt nào đó trong lòng một chuỗi, còn **Length** trả về chiều dài của chuỗi:

```
Console.WriteLine("\nChuỗi s7 dài {0} ký tự.", s7.Length);
Console.WriteLine("Ký tự thứ 5 là {1}\n",s7.Length, s7[4]);
```

và sau đây: là kết xuất:

```
Chuỗi s7 dài 8 ký tự.
Ký tự thứ 5 là A
```

Hàm **EndsWith()** hỏi liệu xem một chuỗi con sẽ được tìm thấy ở cuối chuỗi hay không. Do đó, có thể bạn muốn hỏi **s3** trước tiên nó có kết thúc bởi chuỗi con "Training" hay không (không đúng) , rồi sau đó xem nó có kết thúc bởi chuỗi con "Consulting" (đúng)

```
Console.WriteLine("s3:{0}\nKết thúc bởi Training?: {1}\n",
          s3, s3.EndsWith("Training");
Console.WriteLine("Kết thúc bởi Consulting?: {0}",
          s3.EndsWith("Consulting");
```

Kết xuất cho thấy trắc nghiệm đầu thất bại, trắc nghiệm sau thành công:

```
s3:SAMIS, Inc.
provides custom ,NET development,
on-site Training and Consulting";
Kết thúc bởi Training?: False
Kết thúc bởi Consulting?: True
```

Hàm **IndexOf()** cho biết vị trí của một chuỗi con trong lòng một chuỗi, còn hàm **Insert()** thì lại chèn một chuỗi con mới vào một bản sao của chuỗi nguyên thủy. Đoạn mã sau đây xác định vị trí của xuất hiện đầu tiên trên s3 của chuỗi con "Training":

```
Console.WriteLine("\nXuất hiện đầu tiên của Training ");
Console.WriteLine("trên s3 là {0}\n", s3.IndexOf("Training"));
```

và kết xuất cho thấy di số (offset) là 56:

```
Xuất hiện đầu tiên của Training
trên s3 là 56
```

Sau đó, bạn có thể dùng di số này để chèn từ "excellent", theo bởi một ký tự trắng, vào chuỗi **s3** này. Hiện thời, việc chèn vào được thực hiện trên một bản sao của chuỗi nguyên thủy, và chuỗi mới trả về bởi hàm **Insert()** sẽ được gán cho s7.

```
string s8 = s3.Insert(103, "excellent ");
Console.WriteLine("s8: {0}\n", s8);
```

và sau đây: là kết xuất:

```
s8:SAMIS, Inc.
   provides custom ,NET development,
   on-site excellent Training and Consulting";
```

Cuối cùng, bạn có thể phối hợp các tác vụ (operation) kể trên để cho ra một câu lệnh chèn hữu hiệu hơn:

```
string s9 = s3.Insert(s3.IndexOf("Training"), "excellent ");
   Console.WriteLine("s9: {0}\n", s9);
```

với kết xuất tương tự:

```
s9:SAMIS, Inc.
   provides custom ,NET development,
   on-site excellent Training and Consulting";
```

11.1.5 Đi tìm các chuỗi con

Lớp **String** cung cấp cho bạn một hàm nạp chồng **Substring()** dùng trích những chuỗi con từ những chuỗi. Cả hai phiên bản đều dùng một chỉ mục cho biết việc trích bắt đầu từ đâu, và một trong hai phiên bản lại có thêm một chỉ mục thứ hai cho biết việc truy tìm kết thúc ở đâu, thí dụ 11-2 minh họa việc sử dụng hàm **Substring()**.

Thí dụ 11-2: Dùng hàm Substring()

```
namespace Prog_CSharp
{  using System;
   using System.Text:

   public class Tester
   {  static void Main()
      {  // Tạo một chuỗi gì đó để mà làm việc chứ!
         string s1 = "One Two Three Four";
         int ix:

         // Đi lấy chỉ mục của ký tự trắng chót và từ chót
         ix = s1.LastIndexOf(" ");
         string s2 = s1.Substring(ix + 1);

         // Gán cho s1 về chuỗi con bắt đầu từ 0 và kết thúc ở ix
         // (khởi đầu từ chót như vậy s1 sẽ chỉ còn One Two Three:
         s1 = s1.Substring(0, ix);

         // Đi lấy chỉ mục của ký tự trắng chót trên s1 (sau Two)
         ix = s1.LastIndexOf(" ");
```

```
            // Gán cho s3 về chuỗi con bắt đầu từ ix+1,
            // như vậy s3 = Three
            string s3 = s1.Substring(ix+1);

            // Gán lại s1 về chuỗi con bắt đầu từ 0
            // và kết thúc bởi ix. Như vậy s1 chứa "One Two"
            s1 = s1.Substring(0, ix);

            // Đi lấy chỉ mục của ký tự trắng chót
            // trên s1 (giữa One và Two)
            ix = s1.LastIndexOf(" ");

            // Gán cho s4 về chuỗi con bắt đầu từ ix+1,
            // như vậy s4 = Two
            string s4 = s1.Substring(ix+1);

            // Gán lại s1 về chuỗi con bắt đầu từ 0
            // và kết thúc bởi ix. Như vậy s1 chứa "One"
            s1 = s1.Substring(0, ix);

            // Đi lấy chỉ mục của ký tự trắng chót trên s1,
            // nhưng không có ix chứa -1
            ix = s1.LastIndexOf(" ");

            // Gán cho s5 về chuỗi con bắt đầu từ ix+1, nhưng không có
            // ký tự trắng, do đó cho s5 về chuỗi con bắt đầu từ 0
            string s5 = s1.Substring(ix+1);

            Console.WriteLine("s2: {0}\ns3: {1}", s2, s3);
            Console.WriteLine("s4: {0}\ns5: {1}", s4, s5);
            Console.WriteLine("s1: {0}\n", s1);
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

s2: Four
s3: Three
s4: Two
s5: One

s1: One

Thí dụ 11-2 không phải là một bài giải "ngon lành" trong việc trích từng từ một trong một câu văn chẳng hạn, nhưng đây là bước khởi đầu tốt minh họa một kỹ thuật hữu ích. Ta bắt đầu tạo một chuỗi **s1**:

```
string s1 = "One Two Three Four";
```

Tiếp theo, **ix** được gán trị của ký tự trắng (space) chót trên chuỗi **s1**:

```
ix = s1.LastIndexOf(" ");
```

Sau đó, chuỗi con nằm sau **ix+1** được gán cho một chuỗi con mới **s2**:

```
string s2 = s1.Substring(ix + 1);
```

như vậy, sẽ trích từ vị trí **ix+1** cho đến cuối hàng, gán cho **s2** trị Four. Bước kế tiếp là gỡ bỏ Four khỏi **s1**. Bạn có thể làm điều này bằng cách gán cho **s1** chuỗi con của **s1** trích bắt đầu từ 0 cho đến **ix**:

```
s1 = s1.Substring(0, ix);
```

Ta lại bắt đầu gán **ix** cho ký tự trắng chót, chỉ về đầu từ **Three**, và ta trích từ này cho vào **s3**. Ta tiếp tục tương tự như thế cho tới khi ta điền **s4** và **s5**. Cuối cùng, ta cho in ra kết quả:

```
s2: Four
s3: Three
s4: Two
s5: One

s1: One
```

Giải thuật này không mấy đẹp đẽ, nhưng chạy được, minh họa việc sử dụng **Substring()**. Ta không dùng đến con trỏ (pointer) như với C++ làm cho đoạn mã không an toàn.

11.1.6 Chẻ chuỗi (Splitting string)

Một giải pháp hữu hiệu hơn đối với bài toán thí dụ 11-2 là thử dùng hàm hành sự **Split()** của lớp **String**. Hàm này lo phân tích ngữ nghĩa (parsing) một chuỗi nào đó thành những chuỗi con. Muốn sử dụng **Split()**, bạn đưa vào một bản dãy gồm toàn những ký tự phân cách (delimiter) nghĩa là những ký tự báo cho biết việc chẻ (split) thành những từ, và hàm sẽ trả về một bản dãy chuỗi con. Thí dụ 11-3, minh họa điều này:

Thí dụ 11-3: Sử dụng hàm Split()

```
namespace Prog_CSharp
{   using System;
    using System.Text;

    public class StringTester
```

```
   {  static void Main()
      {  // Tạo một chuỗi s1 để mà làm việc
         string s1 = "One,Two,Three Samis, Inc.";

         // Các hằng kiện (constant) đối với dấu phẩy (,)
         // và ký tự trắng (space).
         const char Space = ' ';
         const char Comma = ',';

         // Tạo một bản dãy gồm toàn các ký tự phân cách để chẻ câu văn
         char[] delimiters = new char[] {Space, Comma};

         string output = "";
         int ctr = 1;

         // tách chuỗi rồi rảo qua bản dãy chuỗi kết xuất
         foreach (string subString in s1.Split(delimiters))
         {  output += ctr++;
            output += ": ";
            output += subString;
            output += "\n";
         }
         Console.WriteLine(output);
      }  // end Main
   }  // end StringTester
}  // end Prog_CSharp
```

Kết xuất

1: One
2: Two
3: Three
4: Samis
5:
6: Inc.

Bạn bắt đầu tạo một chuỗi để tiến hành phân tích ngữ nghĩa:

```
string s1 = "One, Two, Three Samis, Inc.";
```

Các dấu phân cách là nhóm ký tự dấu phẩy (,) và ký tự trắng (space). Sau đó bạn triệu gọi **Split()** đối với chuỗi này, rồi trao kết quả cho một vòng lặp **foreach**:

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

Bạn bắt đầu khởi gán **output** cho về một chuỗi rỗng. Sau đó, bạn xây dựng chuỗi **output** theo 4 bước. Bạn ghép (concatenate) trị của **ctr**, tiếp theo bạn thêm dấu hai chấm (:), sau đó là chuỗi con hàm **Split()** trả về, và cuối cùng nhảy sang hàng mới (\n). Với mỗi lần concatenation, thì một chuỗi mới được tạo ra, và 4 bước kể trên sẽ được lặp lại cho

mỗi chuỗi con tìm thấy được bởi hàm **Split()**. Việc sao **string** được lặp lại tỏ ra kém hiệu quả.

Vấn đề là kiểu dữ liệu chuỗi chữ không được thiết kế cho loại tác vụ này. Vấn đề của bạn là tạo một chuỗi mới bằng cách cho nối đuôi (appending) một chuỗi được định dạng mỗi lần rảo qua vòng lặp. Lớp bạn cần đến là **StringBuilder**.

11.1.7 Thao tác trên các chuỗi động

Lớp **StringBuilder** được sử dụng để tạo và thay đổi các chuỗi. Về mặt ý nghĩa, đây là việc gói ghém (encapsulation) của một hàm constructor đối với một chuỗi **String**. Sau đây là các thành viên của lớp **StringBuilder**.

Bảng 11-2: Các hàm thành viên của lớp StringBuilder

Public Properties	
Capacity	Đi lấy/Đặt để số ký tự tối đa có thể được chứa trên ký ức được cấp phát bởi thể hiện hiện hành.
Chars	Đi lấy/Đặt để ký tự tại vị trí được chỉ định trên chuỗi này. Trên C#, thuộc tính này là indexer đối với lớp **StringBuilder**.
Length	Đi lấy/Đặt để chiều dài của chuỗi này.
MaxCapacity	Đi lấy khả năng tối đa của chuỗi này.
Public Methods	
Append	*Overloaded*. Cho nối đuôi biểu diễn chuỗi của một đối tượng được chỉ định về cuối chuỗi này.
AppendFormat	*Overloaded*. Cho nối đuôi một chuỗi đã được định dạng (formatted string), có chứa zero hoặc nhiều đặc tả định dạng (format specifications), về chuỗi này. Mỗi format specification sẽ được thay thế bởi biểu diễn chuỗi của một đối tượng đối mục tương ứng.
EnsureCapacity	Bảo đảm khả năng của chuỗi **StringBuilder** ít nhất là ở trị được chỉ định.
Equals	*Overloaded*. Trả về một trị cho biết liệu xem chuỗi này có bằng đối tượng được chỉ định hay không.
GetHashCode (kế thừa từ Object)	Dùng làm một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng dùng trên các giải thuật băm và trên các cấu trúc dữ liệu tương tự như một hash table.
GetType (kế thừa từ Object)	Đi lấy Type của chuỗi hiện hành.
Insert	Overloaded. Cho chèn biểu diễn chuỗi của một đối tượng được chỉ định lên chuỗi này tại một vị trí được chỉ định.

Remove	Gỡ bỏ một khoảng ký tự khỏi chuỗi này.
Replace	Overloaded. Thay thế tất cả các xuất hiện của một ký tự được chỉ định bởi một ký tự hoặc chuỗi được chỉ định.
ToString	Overloaded. Overridden. Chuyển đổi một **StringBuilder** thành một String.

Khác với **String**, chuỗi **StringBuilder** không bất di bất dịch (mutable); khi bạn thay đổi một **StringBuilder**, bạn thay đổi chuỗi hiện hành, chứ không phải trên bản sao. Thí dụ 11-4 thay thế đối tượng **String** trên thí dụ 11-3 bởi một đối tượng **StringBuilder**.

Thí dụ 11-4: Sử dụng một chuỗi StringBuilder

```
namespace Prog_CSharp
{  using System;
   using System.Text;

   public class StringTester
   {  static void Main()
      {  // Tạo một chuỗi s1 để mà làm việc
         string s1 = "One,Two,Three Samis, Inc.";

         // Các hằng kiện (constant) đối với dấu phẩy (,)
         // và ký tự trắng (space).
         const char Space = ' ';
         const char Comma = ',';

         // Tạo một bản dãy gồm toàn các ký tự phân cách để chẻ câu văn
         char[] delimiters = new char[] {Space, Comma};

         // Dùng lớp StringBuilder để tạo chuỗi kết xuất output
         StringBuilder output = new StringBuilder();
         int ctr = 1;

         // tách chuỗi rồi rảo qua bản dãy chuỗi kết xuất
         foreach (string subString in s1.Split(delimiters))
         {  // Hàm AppendFormat() ghi nối đuôi (append)
            // một chuỗi được định dạng
            output.AppendFormat("{0}: {1}\n", ctr++, subString);
         }
         Console.WriteLine(output);
      }  // end Main
   }  // end StringTester
}  // end Prog_CSharp
```

Chỉ có phần cuối chương trình là bị thay đổi. Thay vì dùng tác tử concatenation để thay đổi chuỗi, bạn dùng hàm **AppendFormat** của lớp **StringBuilder** để ghi nối đuôi những chuỗi mới được định dạng giống như bạn tạo ra. Việc này xem ra dễ và hữu hiệu hơn nhiều. Kết xuất cũng giống y chang:

Kết xuất

1: One
2: Two
3: Three
4: Samis
5:
6: Inc.

Bạn để ý Vì bạn trao cho bản dãy **delimiter,** cả dấu phẩy lẫn ký tự trắng, do đó ký tự trắng nằm sau dấu phẩy giữa "Samis" và "Inc." được trả về như là một từ, được đánh số 6 như bạn thấy ở kết xuất trên. Đây là điều bạn không muốn thế. Muốn loại bỏ điều này, bạn phải bảo hàm **Split()** so khớp một dấu phẩy (như giữa One, Two và Three) hoặc một ký tự trắng hoặc một dấu phẩy theo sau bởi ký tự trắng. Điểm sau cùng hơi rắc rối, đòi hỏi bạn phải dùng đến regular expression.

11.2 Regular Expressions

Tạm thời chúng tôi không dịch từ "regular expression", vì dịch theo từng từ một nghe nó vô duyên thế nào đấy! Nếu bạn chưa biết ý nghĩa của từ này, thì ráng chờ đọc cho hết chương này, bạn sẽ biết ngay ý nghĩa của nó.

Regular expression là một ngôn ngữ cực mạnh dùng mô tả văn bản cũng như thao tác trên văn bản. Một regular expression thường được ứng dụng lên một chuỗi, nghĩa là lên một nhóm ký tự. Thường xuyên, chuỗi này có thể là toàn bộ một tài liệu văn bản.

Bằng cách áp dụng một regular expression được xây dựng một cách thích ứng lên chuỗi sau đây:

```
One,Two,Three Samis, Inc.
```

Bạn có thể trả về bất cứ hoặc tất cả các chuỗi con của nó (**Samis** hoặc **One** chẳng hạn) hoặc thay đổi phiên bản của những chuỗi con của nó (SAmIS hoặc OnE chẳng hạn). Điều gì regular expression *sẽ làm* sẽ được xác định bởi cú pháp bản thân regular expression.

Một regular expression là một mẫu dáng văn bản (pattern of text) thường gồm 2 kiểu ký tự: ***literals*** (trực kiện) và ***metacharacters*** (ký tự siêu). Một **literal** đơn thuần chỉ là một ký tự (thí dụ, mẫu tự từ a đến z) mà bạn muốn đem so khớp với chuỗi đích. Còn **metacharacter** là một ký hiệu đặc biệt hoạt động như là một mệnh lệnh đối với bộ phận phân tích ngữ nghĩa (parser) của regular expression. Parser là "guồng máy" (engine) chịu trách nhiệm hiểu regular expression. Pattern mô tả một hoặc nhiều chuỗi phải so khớp khi

bạn truy tìm một "thân văn bản" (body of text). Regular expression dùng làm như một khuôn mẫu (template) để so khớp một mẫu dáng ký tự lên chuỗi cần được dò tìm. Thí dụ, nếu bạn tạo một regular expression:

```
^(From|To|Subject|Date):
```

nó sẽ khớp với bất cứ chuỗi con với các chữ "**From**" hoặc các chữ "**To**" hoặc các chữ "**Subject**" hoặc các chữ "**Date**" miễn là các chữ này bắt đầu bởi một hàng mới (^) và kết thúc bởi dấu hai chấm (:).

Dấu mũ (^, carrot) trong trường hợp này cho bộ parser của regular expression biết chuỗi mà bạn đang truy tìm phải bắt đầu bởi một hàng mới. Các chữ "**From**" và "**to**" là những literal, và những metacharacter "(", ")", và "|" được dùng để tạo nhóm literal và cho biết bất cứ những lựa chọn nào cũng phải khớp. Bạn để ý là dấu ^ cũng là một metacharacter cho biết là khởi đầu một hàng mới. Do đó, bạn đọc hàng sau đây:

```
^(From|To|Subject|Date):
```

như sau: "cho khớp bất cứ chuỗi con nào bắt đầu bởi một hàng mới theo sau bởi bất cứ 4 chuỗi literal **From, To, Subject** và **Date** theo sau bởi dấu hai chấm.

Bạn để ý: Nếu bạn muốn tìm hiểu sâu regular expression thì mời lên MSDN của Microsoft. Trong phạm vi chương này, chúng tôi chỉ đề cập đến thế thôi. Bạn cũng có thể đọc sách *Mastering Regular Expressions* của Jeffrey E.E. Friedl, Nxb O'Reilly & Associate Inc (Mỹ).

11.2.1 Các lớp Regular Expression

.NET Framework cung cấp một cách tiếp cận thiên đối tượng về việc so khớp và thay thế theo regular expression. **System.Text.RegularExpressions** là namespace trên thư viện Base Class Library liên quan đến tất cả các đối tượng.NET Framework được gắn liền với regular expression. Lớp chủ chốt hỗ trợ regular expression là **RegEx**, tượng trưng cho một regular expression bất di bất dịch được biên dịch.

Phần sau đây sẽ mô tả những lớp regular expression mà.NET Framework cung cấp cho bạn.

11.2.1.1 Lớp Regex và các thành viên

Lớp **Regex** tượng trưng cho một regular expression bất di bất dịch (immutable) nghĩa là read-only. Nó cũng chứa một số hàm hành sự static cho phép bạn sử dụng những lớp rex khác mà khỏi phải hiển lộ một cách tường minh các đối tượng của các lớp khác. Thí dụ: Sau đây là một thí dụ nhỏ sử dụng **Regex**

```
Regex r; // khai báo biến đối tượng kiểu Regex
r = new Regex("\\s2000"); // hiển lộ một đối tượng Regex, r,
                          // và khởi gán regular expression
```

Bạn để ý cách sử dụng thêm một backslash như là ký tự escape để chỉ backslash trong lớp ký tự so khớp \s như là một literal.

Lớp **Regex** là lớp quan trọng nhất trong việc khai thác xử lý regular expression, nên chúng tôi cho liệt kê sau đây các hàm hành sự và thuộc tính của lớp **Regex** mà ta sẽ sử dụng trong các thí dụ sau:

Public Properties

Options	Trả về những mục chọn (options) được trao qua cho hàm constructor **Regex**.
RightToLeft	Nhận một trị cho biết liệu xem regular expression dò tìm từ phải qua trái hay không

Public Methods

CompileToAssembly	*Overloaded*. Biên dịch các regular expression rồi cho cất trữ lên đĩa theo một assembly duy nhất.
Equals (kế thừa từ Object)	*Overloaded*. Xác định liệu xem hai thể hiện **Object** có bằng nhau hay không.
Escape	Cho thoát (escape) một bộ tối thiểu những ký tự metacharacters (\, *, +, ?, |, {, [, (,), ^, $,., #, and white space) bằng cách thay thế chúng bởi mã escape code của chúng.
GetGroupNames	Trả về một bản dãy gồm toàn tên nhóm thu lượm (capturing group name) đối với regular expression.
GetGroupNumbers	Trả về một bản dãy gồm toàn số nhóm thu lượm (capturing group number) tương ứng với tên nhóm trên một bản dãy.
GetHashCode (kế thừa từ Object)	Dùng như là một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, phù hợp với việc sử dụng trên các giải thuật băm và cấu trúc dữ liệu giống như một hash table.
GetType (kế thừa từ Object)	Đi lấy **Type** của thể hiện hiện hành.
GroupNameFromNumber	Đi lấy tên nhóm tương ứng với số nhóm được khai báo.
GroupNumberFromName	Đi lấy số nhóm tương ứng với tên nhóm được khai báo.
IsMatch	*Overloaded*. Trả về một trị bool cho biết liệu xem regular expression có tìm thấy một so khớp hay không trên chuỗi nhập liệu.
Match	*Overloaded*. Dò tìm trên một chuỗi nhập xem có xuất hiện một regular expression hay không rồi trả về kết quả chính

	xác như là một đối tượng **Match** duy nhất.
Matches	*Overloaded*. Dò tìm trên một chuỗi nhập xem tất cả các xuất hiện của một regular expression có hay không rồi trả về tất cả những so khớp thành công xem như **Match** được gọi nhiều lần.
Replace	*Overloaded*. Cho thay thế những xuất hiện của một mẫu dáng ký tự (character pattern) được định nghĩa bởi một regular expression bởi một chuỗi ký tự thay thế được chỉ định.
Split	*Overloaded*. Chẻ một chuỗi nhập liệu thành một bản dãy gồm những chuỗi con ở những vị trí được chỉ định bởi một so khớp trên một regular expression.
ToString	*Overridden*. Trả về mẫu dáng regular expression được trao qua cho hàm constructor của **Regex**.
Unescape	Cho unescape bất cứ những ký tự nào được escape trên chuỗi nhập liệu.

11.2.1.1.1 Sử dụng lớp RegEx

Mặc dầu có thể tạo những thể hiện **Regex**, lớp này cũng cung cấp một số hàm hành sự static hữu ích. Thí dụ 11-5 minh họa việc sử dụng **RegEx**.

Thí dụ 11-5: Sử dụng lớp Regex đối với regular expression

```
namespace Prog_CSharp
{   using System;
    using System.Text;  using System.Text.RegularExpressions;

    public class Tester
    {   static void Main()
        {   string s1 = "One, Two, Three Samis, Inc.";
            Regex theRegex = new Regex(" |, ");
            StringBuilder sBuilder = new StringBuilder();
            int id = 1;

            foreach (string subString in theRegex.Split(s1))
            {   sBuilder.AppendFormat("{0}: {1}\n", id++, subString);
            }   // end foreach
            Console.WriteLine("{0}", sBuilder);
        }   // end Main
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất

```
1: One
2: Two
```

3: Three
4: Samis
5: Inc.

Thí dụ 11-5 bắt đầu bằng cách tạo một chuỗi **s1**, tương tự như chuỗi dùng trên thí dụ 11-4:

```
string s1 = "One, Two, Three Samis, Inc.";
```

và một regular expression mà ta sẽ dùng truy tìm chuỗi này:

```
Regex theRegex = new Regex(" |, ");
```

Một trong những hàm constructor đối với **Regex** sẽ lấy một chuỗi regular expression làm đối mục. Có vẻ hơi khó hiểu. Trong phạm trù của một chương trình C#, đâu là regular expression: văn bản được trao cho hàm constructor, hay là bản thân đối tượng **Regex**? Đúng là chuỗi văn bản được trao cho hàm constructor là một regular expression theo kiểu cổ điển của từ. Tuy nhiên, theo quan điểm C# thiên đối tượng , đối mục của hàm constructor chỉ đơn giản là một chuỗi ký tự; chính **theRegex** là đối tượng regular expression.

Phần còn lại chương trình hoạt động tương tự như thí dụ 11-4 trang trước; ngoại trừ thay vì triệu gọi **Split()** trên **s1**, thì hàm **Split()** của **Regex** sẽ được triệu gọi. **Regex.Split()** hoạt động cũng giống như **String.Split()**, trả về một bản dãy chuỗi như là kết quả việc so khớp pattern của regular expression trong lòng **theRegex**.

Regex.Split() được nạp chồng (overloaded). Phiên bản đơn giản nhất sẽ được triệu gọi vào trên một thể hiện của **Regex** như ta đã thấy trên thí dụ 11-5. Cũng có một phiên bản static đối với hàm này; hàm này nhận một chuỗi để truy tìm và pattern dùng kèm để truy tìm, như được minh họa trên thí dụ 11-6 sau đây:

Thí dụ 11-6: Sử dụng static Regex.Split() đối với regular expression

```
namespace Prog_CSharp
{  using System;
   using System.Text:
   using System.Text.RegularExpressions;
   public class Tester
   {  static void Main()
      {  string s1 = "One, Two, Three Samis, Inc.";
         StringBuilder sBuilder = new StringBuilder();
         int id = 1;

         foreach (string subString in Regex.Split(s1, " |, "))
         {  sBuilder.AppendFormat("{0}: {1}\n", id++, subString);
```

```
            }  // end foreach
            Console.WriteLine("{0}", sBuilder);
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

1: One
2: Two
3: Three
4: Samis
5: Inc.

Thí dụ 11-6 cũng tương tự như thí dụ 11-5, ngoại trừ thí dụ 11-6 không tạo một thể hiện đối tượng **Regex**. Thay vào đó, nó dùng phiên bản static của **Split()**, tiếp nhận hai đối mục: một chuỗi phải truy tìm, **s1**, và chuỗi regular expression tượng trưng cho pattern phải so khớp.

Hàm thể hiện của **Split()** cũng bị nạp chồng với những phiên bản giới hạn số lần việc tách chẻ xảy ra, và còn xác định vị trí trong lòng chuỗi đích mà việc truy tìm sẽ bắt đầu.

11.2.1.2 Lớp Match

Lớp **Match** tượng trưng cho những kết quả duy nhất của một tác vụ so khớp (match) regular expression. Sau đây là một thí dụ nhỏ sử dụng hàm hành sự **Match** của lớp **Regex** để trả về một đối tượng kiểu **Match** để có thể tìm ra so khớp đầu tiên trên chuỗi chữ nhập. Thí dụ sử dụng thuộc tính **Match.Success** của lớp **Match** báo cho biết liệu xem đã tìm ra một so khớp hay chưa.

```
Regex r = new Regex("abc"); // hiển lộ một đối tượng Regex
// Tìm ra một match duy nhất trên chuỗi.
Match m = r.Match("123abc456");
if (m.Success)
{
 // In ra vị trí của ký tự khớp (trong trường hợp này: 3
 Console.WriteLine("Tìm thấy khớp ở vị trí " + m.Index);
}
```

11.2.1.3 Lớp MatchCollection

Hai lớp bổ sung trong namespace.NET **RegularExpressions** cho phép bạn lặp lại việc truy tìm, và trả về kết quả lên một tập hợp. Collection được trả về thuộc kiểu dữ liệu **MatchCollection**, bao gồm zero hoặc nhiều đối tượng **Match**.

Lớp **MatchCollection** tượng trưng cho một loạt những so khớp thành công không đè chồng lên nhau (non-overlapping) tạo thành một collection bất di bất dịch và lớp này không có hàm public constructor. Những thể hiện của lớp **MatchCollection** sẽ do thuộc tính **Regex.Matches** của lớp **Regex** trả về.

Thí dụ nhỏ sử dụng sử dụng hàm hành sự **Matches** của lớp **Regex** để hình thành một **MatchCollection** với tất cả các so khớp được tìm thấy trong chuỗi chữ nhập. Ta cho chép collection lên một bản dãy kiểu string để cầm giữ mỗi so khớp thành công, và một bản dãy số nguyên cho biết vị trí của mỗi so khớp.

```
MatchCollection mc;   // collection chứa những so khớp
String[] results = new String[20]; // bản dãy kết quả so khớp
int[] matchposition = new int[20]; // bản dãy vị trí so khớp
Regex r = new Regex("abc");      // hiển lộ một đối tượng Regex
                                    // và định nghĩa một rex
// Dùng hàm hành sự Matches để tìm ra tất cả các matches.
mc = r.Matches("123abc4abcd");
// Rảo tìm match collection để tìm ra matches và positions.
for (int i = 0; i < mc.Count; i++)
{
      // Thêm vào bản dãy match string.
      results[i] = mc[i].Value;
      // Ghi nhận character position nơi tìm ra match.
      matchposition[i] = mc[i].Index;
}
```

11.2.1.3.1 Sử dụng Regex Match Collections

Hai thuộc tính quan trọng của đối tượng **Match** là chiều dài và trị của nó, mà ta có thể đọc như theo Thí dụ 11-7 sau đây:

Thí dụ 11-7: Sử dụng MatchCollection và Match

```
namespace Prog_CSharp
{   using System;
    using System.Text.RegularExpressions:

    public class Tester
    {   public static void Main()
        {   string string1 = "This is a test string";

            // cho tìm ra bất cứ nonwhitespace theo sau bởi whitespace
            Regex theRegex = new Regex(@"(\S+)\s");

            // đi lấy collection những gì so khớp
            MatchCollection theMatches = theRegex.Matches(string1);

            // rảo qua collection
            foreach (Match theMatch in theMatches)
            {   Console.WriteLine("theMatch.Length: {0}", theMatch.Length);
```

```
            if (theMatch.Length != 0)
            {   Console.WriteLine("theMatch: {0}", theMatch.ToString());
            }
         }  // end for
      }  // end Main
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

theMatch.Length: 5
theMatch: This
theMatch.Length: 3
theMatch: is
theMatch.Length: 2
theMatch: a
theMatch.Length: 5
theMatch: test
theMatch.Length: 6
theMatch: string

Bạn để ý: Các ký tự Space, tab, linefeed, carriage-return, formfeed, vertical-tab, và newline characters đều được gọi là "white-space characters".

Thí dụ 11-7 cho tạo một chuỗi đơn giản để tiến hành truy tìm:

```
string string1 = "This is a test string";
```

và một regular expression làm "khuôn giá truy tìm":

```
Regex theRegex = new Regex(@"(\S+)\s");
```

Chuỗi \S đi tìm nonwhitespace, và dấu + cho biết một hoặc nhiều. Còn chuỗi \s (Bạn để ý chữ thường) cho biết là whitespace. Do đó, gộp lại, chuỗi này đi tìm bất cứ ký tự nonwhitespace theo sau bởi whitespace.

Kết xuất cho thấy là đã tìm thấy 4 từ đầu. Từ chót không tìm thấy vì nó không theo sau bởi ký tự trắng. Nếu bạn chèn thêm một ký tự trắng sau từ string, và trước dấu nháy ", thì chương trình sẽ tìm thấy từ string này.

Thuộc tính Length là chiều dài của chuỗi con nắm được, và sẽ được đề cập ở mục "Sử dụng CaptureCollection" cuối chương này.

11.2.1.4 Lớp Group

Đôi khi người ta cho là rất tiện khi cho gộp lại (grouping) những biểu thức con (subexpression) so khớp với nhau như vậy bạn có thể phân tích ngữ nghĩa những đoạn của chuỗi khớp. Thí dụ, có thể bạn muốn so khớp dựa trên địa chỉ Internet IP và cho gộp lại tất cả các **IPAddresses** tìm thấy được bất cứ nơi nào trên đoạn chuỗi.

Bạn để ý: IP Address được dùng để nhận diện một máy tính trên mạng, và thường có dạng sau đây: 123.456.789.012.

Lớp **Group** cho phép bạn tạo những nhóm so khớp (group of matches) dựa trên cú pháp regular expression, và tượng trưng cho kết quả từ một biểu thức gộp nhóm (grouping expression) duy nhất.

Một biểu thức gộp nhóm đặt tên cho một nhóm và cung cấp một regular expression; bất cứ chuỗi con nào khớp với regular expression sẽ được đưa vào nhóm. Thí dụ, muốn tạo một nhóm **ip**, bạn có thể viết một regular expression cho biết một hoặc nhiều digit hoặc dot theo sau bởi space như sau:

```
@"(?<ip>(\d|\.)+)\s"
```

Lớp **Match** được dẫn xuất từ **Group**, và có một collection mang tên "**Groups"** chứa tất cả các nhóm mà **Match** tìm thấy.

Lớp **Group** tượng trưng cho những kết quả thu hoạch được từ một thu lượm nhóm (capturing group) duy nhất. Vì **Group** có thể thu lượm 0, 1 hoặc nhiều chuỗi chữ trong một lần so khớp duy nhất (dùng đến quantifiers), nó chứa một collection gồm những đối tượng của **Capture**. Vì **Group** kế thừa từ **Capture**, substring chót bị thu lượm có thể được truy xuất trực tiếp (bản thân thể hiện **Group** tương đương với mục tin chót của collection được trả về bởi thuộc tính **Captures**).

Các thể hiện của **Group** sẽ được trả về bởi thuộc tính **Match.Groups**(*groupnum*), hoặc **Match.Groups**("*groupname*") nếu cấu trúc gộp nhóm "(?<*groupname*>)" được dùng đến.

Thí dụ sau đây sử dụng kiến trúc gộp nhóm lồng nhau để thu lượm những chuỗi con gộp thành nhóm.

```
int[] matchposition = new int[20];   // bản dãy vị trí
```

```
String[] results = new String[20];   // bản dãy kết quả so khớp
// Định nghĩa những substrings abc, ab, b.
Regex r = new Regex("(a(b))c");
Match m = r.Match("abdabc");
for (int i = 0; m.Groups[i].Value != ""; i++)
{
   results[i]=m.Groups[i].Value; // chép nhóm lên bản dãy chuỗi
   matchposition[i] = m.Groups[i].Index; // chép vị trí lên bản dãy
}
```

Thí dụ trên trả về kết xuất như sau:

```
 results[0] = "abc" matchposition[0] = 3
 results[1] = "ab" matchposition[1] = 3
 results[2] = "b"  matchposition[2] = 4
```

Đoạn mã sau đây sử dụng kiến trúc gộp nhóm có mang tên (name và value) để thu lượm những substrings từ một chuỗi chứa dữ liệu trên một dạng thức "DATANAME:VALUE" mà regular expression bị chẻ ở dấu hai chấm (":").

```
Regex r = new Regex("^(?<name>\\w+):(?<value>\\w+)");
Match m = r.Match("Section1:119900");
```

Regular expression trả về kết xuất sau đây:

```
m.Groups["name"].Value = "Section1"
m.Groups["value"].Value = "119900"
```

11.2.1.4.1 Sử dụng cụ thể lớp Group

Thí dụ 11-8 minh họa việc sử dụng collection **Groups** và lớp **Group**.

Thí dụ 11-8: Sử dụng lớp Group

```
namespace Prog_CSharp
{  using System;
   using System.Text.RegularExpressions;

   public class Tester
   {  static void Main()
      {  // Tạo một chuỗi để làm việc
         string string1 = "04:03:27 127.0.0.0 Samis.com";

         // group time = một hoặc nhiều digit hoặc dấu hai chấm
         // theo sau bởi space
         Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
         // ip address = một hoặc nhiều digit hoặc dấu chấm
         // theo sau bởi space
         @"(?<ip>(\d|\.)+)\s" +
```

```
            // site = một hoặc nhiều ký tự
            @"(?<site>\S)");

            // đi lấy collection những so khớp
            MatchCollectiopn theMatches = theReg.Matches(string1);

            // rảo qua collection
            foreach (Match theMatch in theMatches)
            {   if (theMatch.Length != 0)
                {   Console.WriteLine("\ntheMatch: {0},
                                      theMatch.ToString());
                    Console.WriteLine("\time: {0}, theMatch.Groups["time"]);
                    Console.WriteLine("\ip: {0}, theMatch.Groups["ip"]);
                    Console.WriteLine("\site: {0}, theMatch.Groups[site]);
                }  // end if
            }  // end foreach
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Một lần nữa, thí dụ 11-8 bắt đầu tạo một chuỗi để tiến hành dò tìm:

```
string string1 = "04:03:27 127.0.0.0 Samis.com";
```

Chuỗi này có thể là một trong nhiều chuỗi được ghi nhận trên một tập tin log của web server hoặc như là kết quả dò tìm của database. Trong thí dụ đơn giản này có 3 cột: một cho thời gian (time), một cho địa chỉ Internet IP, và một cho site, mỗi cột cách nhau bởi một ký tự trắng. Trong thực tế, vấn đề có thể phức tạp hơn. Có thể bạn muốn dò tìm với độ phức tạp cao hơn dùng đến nhiều ký tự phân cách (delimiter).

Trong thí dụ 11-8, bạn muốn tạo một đối tượng **Regex** duy nhất để dò tìm những chuỗi kiểu này, và chặt chúng thành 3 nhóm: **time, ip** và **site**. Regular expression khá đơn giản, do đó thí dụ dễ hiểu:

```
// group time = một hoặc nhiều digit hoặc dấu hai chấm
// theo sau bởi space
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
// ip address = một hoặc nhiều digit hoặc dấu chấm theo sau bởi space
@"(?<ip>(\d|\.)+)\s" +
// site = một hoặc nhiều ký tự
@"(?<site>\S)");
```

Ta tập trung xem các ký tự hình thành nhóm:

```
(?<time>
```

Các dấu ngoặc () lo tạo một nhóm. Những gì nằm giữa dấu ngoặc mở (ngay trước dấu hỏi ?) và dấu ngoặc đóng (sau dấu cộng + trong trường hợp này) là một nhóm đơn độc chưa mang tên.

```
(@(?<time>(\d|\:)+)
```

Chuỗi **?<time>** đặt tên cho nhóm là **time** và nhóm được gắn liền với đoạn văn bản so khớp (matching text), là regular expression **(\d|\:)+)\s"**. Regular expression này được suy diễn như sau: "một hoặc nhiều digit hoặc dấu hai chấm theo sau bởi một space".

Cũng tương tự như thế, chuỗi **?<ip>** đặt tên cho nhóm **ip**, và **?<site>** đặt tên cho nhóm **site**. Giống như thí dụ 11-7 đã làm, thí dụ 11-8 cũng đòi hỏi một collection của tất cả các đoạn khớp:

```
MatchCollectiopn theMatches = theReg.Matches(string1);
```

Tiếp theo, cho rảo qua collection **theMatches** để lôi ra mỗi đối tượng **Match**

```
foreach (Match theMatch in theMatches)
```

Nếu chiều dài **Length** của **Match** lớn hơn 0, có nghĩa là đã tìm thấy một so khớp. Sau đó, thì cho in ra toàn bộ những mục so khớp:

```
Console.WriteLine("\ntheMatch: {0}, theMatch.ToString());
```

và sau đây: là kết xuất: `theMatch: 04:03:27 127.0.0.0 Samis.com`

Tiếp theo, là đi lấy nhóm "time" từ collection **Groups** của **Match**, cho in ra nội dung

```
Console.WriteLine("\time: {0}, theMatch.Groups["time"]);
```

với kết xuất là: `time: 04:03:27`

Tiếp theo, chương trình lôi ra các nhóm **ip** và **site**:

```
Console.WriteLine("\ip: {0}, theMatch.Groups["ip"]);
Console.WriteLine("\site: {0}, theMatch.Groups[site]);
```

với kết xuất: ip: 127.0.0.0
site: Samis.com

Trong thí dụ 11-8, collection **theMatches** chỉ có một **Match**. Tuy nhiên, ta có khả năng cho khớp nhiều hơn một regular expression trong lòng một chuỗi. Muốn thế, bạn thử thay đổi **string1** trên thí dụ 11-8, cung cấp nhiều mục vào **logFile** thay vì chỉ một, như sau:

```
string string1 = "04:03:27 127.0.0.0 Samis.com " +
"04:03:28 127.0.0.0 foo.com " + "04:03:29 127.0.0.0 bar.com";
```

Lần này, có 3 match trong collection **theMatches** của lớp **MatchCollection**. Và sau đây là phần kết xuất:

```
theMatch: 04:03:27 127.0.0.0 Samis.com
time: 04:03:27
ip: 127.0.0.0
site: Samis.com

theMatch: 04:03:28 127.0.0.0 foo.com
time: 04:03:28
ip: 127.0.0.0
site: foo.com

theMatch: 04;03;29 127.0.0.0 bar.com
time: 04:03:29
ip: 127.0.0.0
site: bar.com
```

Trong thí dụ này, collection **theMatches** chứa 3 đối tượng **Match**. Mỗi lần rảo qua khỏi vòng ngoài của vòng lặp **foreach**, ta tìm thấy đối tượng **Match** kế tiếp trong collection, rồi cho hiển thị nội dung.

```
foreach (Match theMatch in theMatches)
```

Đối với mỗi mục **Match** tìm thấy, bạn có thể in ra toàn bộ match, các nhóm khác nhau hoặc cả hai.

11.2.1.5 Lớp GroupCollection

Lớp **GroupCollection** tượng trưng một collection gồm toàn những nhóm được thu lượm và trả về một lô những nhóm được thu lượm trong một lần so khớp duy nhất. Collection này thuộc loại read-only và không có hàm public constructor. Các thể hiện của lớp **GroupCollection** được trả về trong collection mà thuộc tính **Match.Groups** trả về.

Thí dụ sau đây dò tìm và in ra số những nhóm được thu lượm bởi một regular expression. Làm thế nào để trích từng thu lượm riêng rẽ trên mỗi thành viên của một group collection, đề nghị bạn xem thí dụ về lớp **CaptureCollection** trong phần kế tiếp.

```
using System.Text.RegularExpressions;

public class RegexTest
{
   public static void RunTest()
   {
   // Định nghĩa những nhóm "abc", "ab", và "b".
   Regex r = new Regex("(a(b))c");
```

```
    Match m = r.Match("abdabc");
    Console.WriteLine("Số nhóm tìm thấy = " + m.Groups.Count);
    }
    public static void Main()
    {
    RunTest();
    }
}
```

Thí dụ này cho ra kết xuất sau đây:

```
Số nhóm tìm thấy = 3
```

11.2.1.6 Lớp Capture

Lớp **Capture** chứa những kết quả từ một thu lượm duy nhất dựa trên một subexpression.

Thí dụ sau đây rảo qua một collection **Group,** trích ra collection **Capture** từ mỗi thành viên của **Group**, rồi gán những biến ***posn*** và ***length*** lần lượt cho biết vị trí của ký tự trên chuỗi chữ nguyên thủy nơi mà mỗi chuỗi được tìm thấy kèm theo chiều dài.

```
Regex r;
Match m;
CaptureCollection cc;
int posn, length;

r = new Regex("(abc)*");
m = r.Match("bcabcabc");
for (int i=0; m.Groups[i].Value != ""; i++)
{
    // Thu lượm Collection đối với Group(i).
    cc = m.Groups[i].Captures;
    for (int j = 0; j < cc.Count; j++)
    {
    // Báo vị trí của Capture object.
    posn = cc[j].Index;
    // Cho biết chiều dài của Capture object.
    length = cc[j].Length;
    }
}
```

11.2.1.7 Lớp CaptureCollection

Mỗi lần một đối tượng **Regex** khớp với một subexpression, một thể hiện **Capture** sẽ được tạo ra, và được thêm vào một collection **CaptureCollection**. Mỗi đối tượng

Capture tượng trưng cho một thu lượm (capture) đơn lẻ. Mỗi nhóm sẽ có riêng cho mình một capture collection những mục khớp với subexpression được gắn liền với nhóm.

Lớp **CaptureCollection** tượng trưng cho một loạt những chuỗi con được thu lượm và trả về một lô những thu lượm được thực hiện chỉ qua một nhóm thu lượm duy nhất. Thuộc tính **Captures,** một đối tượng của lớp **CaptureCollection**, được cung cấp như là một thành viên của các lớp **Match** và **Group** giúp truy xuất dễ dàng lô các chuỗi con được thu lượm.

Thí dụ, nếu bạn sử dụng regular expression **((a(b))c)+** (theo đấy dấu + được gọi là ***quantifier*** cho biết là một hoặc nhiều so khớp) để thu lượm những so khớp từ chuỗi chữ "abcabcabc", **CaptureCollection** đối với mỗi matching **Group** của những substring sẽ chứa 3 thành viên.

Thí dụ sau đây dùng đến regular expression **(Abc)+** để tìm ra một hoặc nhiều so khớp trên chuỗi chữ **"XYZAbcAbcAbcXYZAbcAb**". Thí dụ minh họa việc sử dụng thuộc tính **Captures** để trả về nhiều nhóm các chuỗi con bị thu lượm.

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

public class RegexTest
{
    public static void RunTest()
    {
    int counter;
    Match m;
    CaptureCollection cc;
    GroupCollection gc;

    // Dò tìm những nhóm "Abc".
    Regex r = new Regex("(Abc)+");
    // Định nghĩa chuỗi chữ
    m = r.Match("XYZAbcAbcAbcXYZAbcAb");
    gc = m.Groups;

    // In ra số nhóm
    Console.WriteLine("Nhóm thu lượm được = " +
                    gc.Count.ToString());

    // Rảo qua mỗi group.
    for (int i=0; i < gc.Count; i++)
    {
        cc = gc[i].Captures;
        counter = cc.Count;

        // In ra số captures trong nhóm này.
        Console.WriteLine("Đếm số captures = " +
                          counter.ToString());
```

```
        // Rảo qua mỗi capture trong group.
        for (int ii = 0; ii < counter; ii++)
        {
            // In ra capture và position.
            Console.WriteLine(cc[ii] + "Bắt đầu từ ký tự " +
                                       cc[ii].Index);
        }
    }
    }

    public static void Main() {
    RunTest();
    }
}
```

Kết xuất

Nhóm thu lượm được = = 2
Đếm số captures = = 1
AbcAbcAbc Bắt đầu từ ký tự 3
Đếm số captures = 3
Abc Bắt đầu từ ký tự 3
Abc Bắt đầu từ ký tự 6
Abc Bắt đầu từ ký tự 9

11.2.1.7.1 Sử dụng lớp CaptureCollection

Thuộc tính chủ chốt của đối tượng **Capture** là **length**, cho biết chiều dài của chuỗi con bị thu lượm. Khi bạn yêu cầu **Match** cho biết chiều dài, thì chính **Capture.Length** bạn tìm thấy, vì **Match** được dẫn xuất từ **Group**, và **Group** lại được dẫn xuất từ **Capture**.

Điển hình, bạn chỉ sẽ tìm thấy một **Capture** đơn độc trong một **CaptureCollection**; nhưng điều này không buộc phải như thế. Điều gì sẽ xảy ra nếu bạn phân tích ngữ nghĩa một chuỗi trong ấy tên công ty (company) có thể xuất hiện hoặc ở hai nơi. Muốn gộp các tên này vào chung thành một match đơn lẻ, bạn tạo nhóm **?<company>** ở hai nơi trong mẫu dáng của regular expression:

```
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
       @"(?<company>\S+)\s" +
       @"(?<ip>(\d|\.)+)\s" +
       @"(?<company>\S+)\s");
```

Regular expression nhóm chận bắt bất cứ chuỗi ký tự nào khớp theo sau **time**, cũng như sau **ip**. Với regular expression này, bạn sẵn sàng phân tích chuỗi sau đây:

```
string string1 = "04:03:27 Thien 0.0.0.127 Samis";
```

chuỗi trên bao gồm những tên ở hai nơi được chỉ định. Sau đây là kết xuất:

```
theMatch: 04:03:27 Thien 0.0.0.127 Samis
time:04:03::27
ip: 0.0.0.127
company: Samis
```

Việc gì đã xảy ra? Vì sao nhóm Company chỉ cho thấy Samis mà thôi. Mục đầu tiên Thien, cũng khớp, biến đâu mất rồi. Câu trả lời là mục hai đè lên mục đi trước. Tuy nhiên, nhóm đã chận bắt được cả hai; và collection **Captures** có thể cho bạn thấy, như theo thí dụ 11-9.

Thí dụ 11-9: Thử quan sát capture collection

```
namespace Prog_CSharp
{   using System;
    using System.Text.RegularExpressions;

    public class Tester
    {   static void Main()
        {   // Tạo một chuỗi để làm việc, với hai tên ở hai nơi
            string string1 = "04:03:27 Thien 0.0.0.127 Samis ";

            // regular expression với nhóm company ở hai nơi
            Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
            @"(?<company>\S+)\s" +
            @"(?<ip>(\d|\.)+)\s" +
            @"(?<company>\S+)\s");

            // đi lấy collection những so khớp
            MatchCollectiopn theMatches = theReg.Matches(string1);

            // rảo qua collection
            foreach (Match theMatch in theMatches)
            {   if (theMatch.Length != 0)
                {   Console.WriteLine("\ntheMatch: {0},
                                       theMatch.ToString());
                    Console.WriteLine("\time: {0}, theMatch.Groups["time"]);
                    Console.WriteLine("\ip: {0}, theMatch.Groups["ip"]);
                    Console.WriteLine("\Company: {0},
                                       theMatch.Groups[company]);

                    // rảo qua capture collection trên nhóm company trong
                    // lóng collection Groups trong Match
                    foreach (Capture cap in
                              theMatch.Groups["company"].Captures)
                    {   Console.WriteLine("cap: {0}", cap.ToString());
                    }   // end foreach trong
                }   // end if
            }   // end foreach ngoài
        }   // end Main
```

```
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất

```
theMatch: 04:03:27 Thien 0.0.0.127 Samis
time:04:03:27
ip: 0.0.0.127
Company: Samis
cap: Thien
cap: Samis
```

Đoạn mã in đậm cho rảo qua collection **Captures** đối với nhóm **Company**.

```
foreach (Capture cap in theMatch.Groups["company"].Captures)
```

Ta thử xem lại hàng này được phân tích ngữ nghĩa thế nào. Trình biên dịch bắt đầu bằng cách tìm ra collection mà ta sẽ rảo qua trên ấy. **theMatch** là một đối tượng có một collection mang tên **Groups**. Collection **Groups** có một indexer cho phép trích một chuỗi và trả về một đối tượng **Group** đơn lẻ. Do đó, lệnh sau đây: trả về một đối tượng **Group** đơn lẻ:

```
theMatch.Groups["company"].Captures
```

Đến phiên, vòng lặp **foreach** rảo qua collection **Captures**, trích mỗi phần tử collection và gán cho biến cục bộ **cap**, thuộc kiểu dữ liệu **Capture**. Bạn có thể là trên kết xuất có hai phần tử capture: Thien và Samis. Phần tử thứ hai đè chồng lên phần tử đầu trên nhóm, do đó chỉ in ra Samis, nhưng khi quan sát collection **Captures** thì bạn thấy có hai trị bị thu lượm.

Chương 12

Thụ lý các biệt lệ

C#, giống như nhiều ngôn ngữ thiên đối tượng khác, xử lý những sai lầm và những điều kiện bất bình thường thông qua các *biệt lệ* [44](exception). Trên C#, biệt lệ cung cấp cho bạn một thể thức (an toàn, đồng nhất và có cấu trúc) để thụ lý những trường hợp sai lầm ở cấp hệ thống lẫn cấp ứng dụng. Trên C#, tất cả các biệt lệ đều phải được tượng trưng bởi một đối tượng của một lớp được dẫn xuất từ **System.Exception.** Đối tượng này gói ghém mọi thông tin liên quan đến một việc xảy ra bất bình thường trong chương trình.

Điều quan trọng là bạn phải phân biệt giữa bug, error và exception. ***Bug*** là lỗi chương trình mà bạn phải cho sửa chữa trước khi đem giao cho khách hàng sử dụng. **Exception** không thể chống đở tránh khỏi bug. Mặc dù một bug có thể gây ra một biệt lệ, bạn không nên chờ đợi biệt lệ sẽ xử lý bug của bạn. Thật thế, bạn phải tự mình sửa chữa các bug. Thông thường, bạn sửa chữa bug vào lúc biên dịch.

Một **sai lầm** thường được gây ra bởi hành động của người sử dụng. Thí dụ, thay vì khỏ vào một con số như chờ đợi, người sử dụng có thể khỏ vào một chữ. Một lần nữa, một sai lầm có thể gây ra một biệt lệ, nhưng bạn có thể ngăn ngừa việc này xảy ra bằng cách viết những đoạn mã kiểm tra hợp lệ (validation code). Nếu có thể được, thì bao giờ bạn cũng phải tiên liệu những sai lầm có thể xảy ra trong mọi tình huống để viết những đoạn mã kiểm tra hợp lệ ngăn ngừa sai lầm.

Cho dù bạn đã sửa chữa bug và tiên liệu mọi sai lầm từ phía người sử dụng để ngăn ngừa trước, bạn vẫn có thể bị lôi cuốn vào những vấn đề có thể tiên liệu nhưng lại không thể ngăn ngừa được, chẳng hạn trường hợp thiếu ký ức (out of memory) hoặc cố mở một tập tin không hề có (có thể do khỏ sai). Bạn không thể ngăn ngừa các biệt lệ, nhưng bạn có thể xử lý chúng để chương trình không bị "ngủm" nữa chừng.

Khi chương trình của bạn gặp phải một tình huống khác thường, chẳng hạn không đủ ký ức (out of memory), thì nó *tống ra* (throw) một biệt lệ. Khi một biệt lệ được tống ra, việc thi hành hàm hiện hành tạm ngưng, và call stack được trải trưng ra (unwound) cho tới khi nào bạn tìm thấy một hàm thụ lý biệt lệ (exception handler) thích ứng.

---

[44] Chúng tôi không dịch là "ngoại lệ" như một số người đã dịch.

Điều này có nghĩa là nếu hàm hiện đang chạy không thể xử lý biệt lệ, thì hàm hiện hành sẽ bị chấm dứt, và hàm triệu gọi (calling function) sẽ có cơ may xử lý biệt lệ. Nếu không hàm triệu gọi nào có thể xử lý biệt lệ, thì cuối cùng biệt lệ sẽ được xử lý bởi CLR, và CLR có thể cho ngưng "cái rụp" chương trình của bạn.

Đoạn mã dùng xử lý biệt lệ được tống ra được gọi là *exception handler*, tạm dịch là hàm *thụ lý biệt lệ,* và được thiết đặt như là lệnh **catch**. Thật lý tưởng nếu biệt lệ được "tóm gọn" và được xử lý, chương trình có thể sửa chữa vấn đề và tiếp tục chạy. Cho dù chương trình của bạn không thể tiếp tục, bằng cách tóm biệt lệ bạn có cơ may in ra một thông điệp sai lầm mang ý nghĩa và cho chương trình "hạ cánh" một cách đẹp đẽ và an toàn.

Nếu trong hàm, có đoạn mã buộc phải chạy cho dù có biệt lệ hay không (nghĩa là phải giải phóng các nguồn lực bạn đã cấp phát), bạn có thể đặt đoạn mã này nằm ở khối *finally*, biết rằng tại đây đoạn mã sẽ được thi hành cho dù có biệt lệ.

Biệt lệ có thể được tống ra theo hai cách khác nhau:

- Một lệnh **throw** sẽ tung ra lập tức một biệt lệ và vô điều kiện. Quyền điều khiển không bao giờ đạt đến lệnh nằm ngay liền sau lệnh throw.

- Khi chương trình của bạn gặp phải một tình huống khác thường, khi đang xử lý các lệnh C# và các biểu thức thì sẽ gây ra một biệt lệ khi chương trình không thể kết thúc bình thường. Thí dụ chia một số cho zero sẽ tung ra biệt lệ mang tên **System.DivideByZeroException. Bạn** xem sau các lớp biệt lệ của hệ thống.

12.1 Các lớp biệt lệ

Sau đây là những biệt lệ mà vài tác vụ C# sẽ tung ra. Bạn để ý các biệt lệ này bao giờ cũng kết thúc bởi từ Exception.

System.ArithmeticException	Một lớp căn bản đối với những biệt lệ xảy ra khi tiến hành những tính toán số học, chẳng hạn **System.DivideByZero Exception** và **System.OverflowException**.
System.ArrayTypeMismatchException	Biệt lệ được tung ra khi việc trữ lên bản dãy thất bại vì kiểu dữ liệu hiện thời của phần tử được trữ không tương thích với kiểu dữ liệu hiện thời của bản dãy.
System.DivideByZeroException	Biệt lệ được tung ra khi cố chia một trị số

	nguyên cho zero.
System.IndexOutOfRangeException	Biệt lệ được tung ra khi cố công chỉ mục một bản dãy thông qua một chỉ số nhỏ thua zero hoặc ngoài phạm vi của bản dãy.
System.InvalidCastException	Biệt lệ được tung ra khi một chuyển đổi tường minh từ một kiểu dữ liệu căn bản hoặc giao diện về một kiểu dữ liệu được dẫn xuất không thành công vào lúc chạy.
System.MulticastNotSupportedException	Biệt lệ được tung ra khi cố gắng phối hợp hai non-null delegates bất thành, vì kiểu dữ liệu của delegate không có kiểu dữ liệu trả về void.
System.NullReferenceException	Biệt lệ được tung ra khi một null reference được sử dụng theo một thể thức gây ra là cần phải có đối tượng được qui chiếu.
System.OutOfMemoryException	Biệt lệ được tung ra khi thất bại trong việc cấp phát ký ức (thông qua tác tử **new**).
System.OverflowException	Biệt lệ được tung ra khi một tác vụ toán số học bị tràn năng (overflows).
System.StackOverflowException	Biệt lệ được tung ra khi stack thi hành chương trình bị cạn kiệt ký ức vì có quá nhiều triệu gọi hàm nằm treo, do việc đệ quy quá sâu.
System.TypeInitializationException	Biệt lệ được tung ra khi một static constructor tung ra một biệt lệ, và không hiện hữu một catch clause để chận bắt.

12.2 Tống và tóm bắt biệt lệ

Trên C#, bạn chỉ có thể tống ra những đối tượng kiểu **System.Exception**, hoặc những đối tượng được dẫn xuất từ kiểu dữ liệu này. Namespace **System** bao gồm một số kiểu biệt lệ mà bạn có thể dùng trong chương trình của bạn. Những kiểu biệt lệ này gồm **ArgumentNullException** (biệt lệ do đối mục null), **InvalidCastException** (biệt lệ do ép kiểu bất hợp lệ) và **OverflowException** (biệt lệ do tràn năng) cũng như nhiều biệt lệ khác.

12.2.1 Câu lệnh *throw*

Trong lớp C#, để báo động một điều kiện bất bình thường khi chương trình đang thi hành, bạn cho tống ra một biệt lệ, bằng cách dùng câu lệnh **throw** mang cú pháp sau đây:

throw [*expression*];

với *expression* là một đối tượng biệt lệ (được dẫn xuất từ **System.Exception**). Bạn để ý *expression* là tùy chọn, nghĩa là không ghi ra khi tống ra lại đối tượng biệt lệ hiện hành trong một điều khoản **catch.**

Câu lệnh sau đây tạo một thể hiện của **System.Exception**, rồi cho tống nó ra:

```
throw new System.Exception(); hoặc

class MyException: System.Exception {}
throw new MyException();
```

Thông thường câu lệnh **throw** được dùng phối hợp với khối câu lệnh **try-catch** hoặc **try-finally**. Khi biệt lệ được tung ra, chương trình sẽ đi tìm câu lệnh **catch** lo thụ lý biệt lệ này.

Việc tống ra một biệt lệ sẽ lập tức ngưng việc thi hành chương trình, trong khi CLR đi lùng một bộ thụ lý biệt lệ (exception handler). Nếu bộ phận này không tìm thấy trong hàm hành sự hiện hành, thì cửa sổ call stack sẽ được *trải ra* (unwind), vọt lên rảo qua các hàm triệu gọi cho tới khi tìm thấy một bộ thụ lý biệt lệ. Nếu vọt leo tuốt đến Main() mà không thấy tăm hơi một bộ thụ lý biệt lệ, thì chương trình liền bị ngưng. Thí dụ 12-1 minh họa một biệt lệ.

Thí dụ 12-1: Tống ra một biệt lệ

```
namespace Prog_CSharp
{   using System;

    public class Tester
    {
        public static void Main()
        {   Console.WriteLine("Vào Main…");
            Tester t = new Tester();
            t.Func1();
            Console.WriteLine(Ra khỏi Main…");
        }   // end Main
```

```
        public void Func1()
        {   Console.WriteLine("Vào Func1…");
            Func2();
            Console.WriteLine("Ra khỏi Func1…");
        }   // end Func1

        public void Func2()
        {   Console.WriteLine("Vào Func2…");
            throw new System.Exception();
            Console.WriteLine("Ra khỏi Func2…");
        }   // end Func2
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất

```
Vào Main…
Vào Func1…
Vào Func2…

Exception occurred: System.Exception: An exception of type
System.Exception was thrown.
    at Prog_CSharp.Tester.Func2()
        in …exceptions01.cs:line 26
    at Prog_CSharp.Tester.Func1()
        in …exceptions01.cs:line 20
    at Prog_CSharp.Tester.Main()
        in …exceptions01.cs:line 12
```

Thí dụ đơn giản trên viết ra lên màn hình khi Vào hoặc Ra khỏi từng hàm một. **Main()** tạo một đối tượng kiểu **Tester** và triệu gọi hàm **Func1()**. Sau khi in ra thông điệp "Vào Func1…", hàm **Func1()** liền triệu gọi hàm **Func2()**. Hàm này in ra thông điệp "Vào Func2…" rồi tống ra một đối tượng biệt lệ kiểu **System.Exception**.

Việc thi hành chương trình liền ngưng lập tức và CLR tìm xem trên **Func2()** có một bộ thụ lý biệt lệ hay không. Không có, CLR cho trải ra cửa sổ call stack (không bao giờ in ra thông điệp Ra khỏi…) về **Func1()**. Ở đây, lại không có nốt bộ thụ lý biệt lệ, và CLR lại cho trải stack lui về Main(). Vì ở đây cũng không có bộ thụ lý biệt lệ, nên bộ thụ lý biệt lệ mặc nhiên được gọi vào, cho in ra thông điệp sai lầm.

12.2.2 Câu lệnh *try-catch*

Trên C#, bộ thụ lý biệt lệ thường được gọi là **khối catch** và được tạo bởi câu lệnh **try-catch** bao gồm một khối **try** theo sau bởi một hoặc nhiều điều khoản (clause) **catch,** cho biết những bộ thụ lý biệt lệ khác nhau. Cú pháp câu lệnh **try-catch** có thể mang một trong những dạng thức sau đây:

Dạng 1
```
try try-block
catch (exception-declaration-1) catch-block-1
catch (exception-declaration-2) catch-block-2
```

Dạng 2
```
try try-block catch catch-block
```

theo đấy:

try-block: Chứa đoạn mã mà ta chờ đợi xảy ra biệt lệ.
exception-declaration-1, *exception-declaration-2*: Khai báo đối tượng biệt lệ.
catch-block, *catch-block-1*, *catch-block-2*: Chứa bộ thụ lý biệt lệ.

Khối ***try-block*** chứa đoạn mã phòng ngừa có thể gây ra biệt lệ. Khối này được thi hành cho tới khi biệt lệ được tung ra hoặc được thành công mỹ mãn. Thí dụ sau đây cố ép kiểu (cast) một đối tượng null sẽ gây ra biệt lệ **NullReference Exception** (biệt lệ do qui chiếu null):

```
 object o2 = null;
try
{
 int i2 = (int) o2;  // Sai bét!
}
```

Còn **catch** clause có thể dùng có thông số hoặc không thông số. Nếu không thông số, thì khối này chận bắt bất cứ loại biệt lệ nào cũng được; coi như là điều khoản catch tổng quát. Còn nếu dùng có thông số, thì nó nhận một đối tượng được dẫn xuất từ **System.Exception**, và xem đây như là thụ lý một biệt lệ đặc biệt. Thí dụ:

```
catch (InvalidCastException e)
{
}
```

Ta có thể dùng nhiều **catch** clause trong cùng câu lệnh **try-catch**. Trong trường hợp này, thứ tự bố trí các **catch** clause rất quan trọng, vì các **catch** clause được xét theo thứ tự vào. Những biệt lệ đặc thù nhất sẽ được chận bắt trước những biệt lệ ít đặc thù. Nói cách

khác, những biệt lệ nào hay xảy ra nhiều cho nằm trước, rồi theo tuần tự những biệt lệ ít xảy ra cho đặt sau.

Câu lệnh **throw** có thể được dùng trong khối **catch** để tung ra lại biệt lệ đã bị chận bắt bởi câu lệnh **catch**. Thí dụ:

```
catch (InvalidCastException e)
{
 throw (e);  // Tung ra lại biệt lệ e
}
```

Nếu bạn muốn tung ra lại một biệt lệ hiện được thụ lý bởi một **catch** clause không thông số, thì nên dùng câu lệnh **throw** không đối mục, như theo thí dụ sau đây:

```
catch
{
 throw;
}
```

Trong khối **try** chớ nên khởi gán các biến, vì chưa chắc khối **try** sẽ được thi hành. Một biệt lệ có thể xảy ra trước khi khối try hoàn tất công việc, như theo thí dụ sau đây:

```
public static void Main()
{
  int x;
  try
  {
     x = 123;  //  không được làm thế
   // ...
  }
  catch
  {
   // ...
  }
  Console.Write(x);  // Error: Use of unassigned local variable 'x'.
}
```

Trong thí dụ trên, biến **x** được khởi gán trong lòng khối **try. x** lại được sử dụng ngoài khối **try**, chẳng hạn trên câu lệnh **Write(x).** Điều này sẽ phát sinh sai lầm biên dịch: **Use of unassigned local variable**.

Trên thí dụ 12-2 sau đây lệnh **throw** được thi hành trong lòng khối **try** (thử), và một khối **catch** (chận bắt) được dùng báo cho biết sai lầm đã được thụ lý. Trong trường hợp này chỉ có một khối catch.

Thí dụ 12-2: Chận bắt một biệt lệ

```
namespace Prog_CSharp
{  using System;
   public class Tester
   {  public static void Main()
      {  Console.WriteLine("Vào Main…");
         Tester t = new Tester();
         t.Func1();
         Console.WriteLine(Ra khỏi Main…");
      }  // end Main

      public void Func1()
      {  Console.WriteLine("Vào Func1…");
         Func2();
         Console.WriteLine("Ra khỏi Func1…");
      }  // end Func1

      public void Func2()
      {  Console.WriteLine("Vào Func2…");
         try
         {  Console.WriteLine("Đi vào khối try …");
            throw new System.Exception();
            Console.WriteLine("Ra khỏi khối try …");
         }
         catch
         {  Console.WriteLine("biệt lệ bị chận bắt và thụ lý.");
         }
         Console.WriteLine("Ra khỏi Func2…");
      }  // end Func2
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Vào Main…
Vào Func1…
Vào Func2…
Đi vào khối try …
Biệt lệ bị chận bắt và thụ lý
Ra khỏi Func2…
Ra khỏi Func1…
Ra khỏi Main…

Thí dụ 12-2 cũng giống như thí dụ 12-1, ngoại trừ bây giờ chương trình có bao gồm một khối **try/catch**. Điển hình, bạn nên đặt khối **try** xung quanh câu lệnh mang tiềm năng "gây rối" nguy hiểm, chẳng hạn truy xuất một tập tin, cấp phát ký ức, v.v.

Theo sau khối **try**, là lệnh chung chung **catch**. Trong thí dụ 12-2, **catch** là lệnh chung chung (generic) vì bạn không cho biết loại biệt lệ nào phải chận bắt. Trong trường hợp này, lệnh sẽ chận chận bắt bất cứ biệt lệ nào được tung qua. Chúng tôi sẽ đề cập sau trong chương này việc sử dụng lệnh **catch** để chận bắt những loại biệt lệ đặc thù.

12.2.2.1 Tiến hành hành động sửa sai

Trong thí dụ 12-2, lệnh **catch** chỉ đơn giản báo cáo là biệt lệ đã được chận bắt và dã được thụ lý. Trong thực tế, có thể bạn phải tiến hành những bước sửa chữa vấn đề gây ra biệt lệ. Thí dụ, nếu người sử dụng cố mở một tập tin read-only, có thể bạn cho triệu gọi một hàm hành sự cho phép người sử dụng thay đổi thuộc tính của một tập tin. Nếu chương trình báo là thiếu ký ức, có thể bạn cho người sử dụng cơ hội đóng lại các ứng dụng khác. Nếu tất cả đều thất bại, khối **catch** sẽ cho in ra một thông điệp sai lầm để người sử dụng biết đường mà lần.

12.2.2.2 Cho trải call stack

bạn quan sát kỹ phần kết xuất của thí dụ 12-2. Bạn thấy chương trình đi vào **Main(), Func1(), Func2()** và khối **try**. Bạn không bao giờ thấy nó thoát khỏi **try**, mặc dù nó thoát **Func2(), Func1()** và **Main()**. Việc gì đã xảy ra?

Khi một biệt lệ được tung ra, việc thi hành chương trình ngừng ngay lập tức và được trao quyền điều khiển cho khối **catch**, giống như kẻ tung người hứng. Nó không bao giờ trở về lối đi nguyên thủy của đoạn mã. Nó sẽ không bao giờ đi đến dòng lệnh in ra lệnh **exit** đối với khối **try**. Khối **catch** sẽ thụ lý sai lầm, rồi sau đó việc thi hành nhảy về đoạn mã theo sau **catch**.

Không có **catch**, call stack sẽ cho trải ra (unwind), nhưng khi có **catch** thì call stack sẽ không trải ra như là kết quả của biệt lệ. Giờ đây, biệt lệ được xử lý, không còn vấn đề và chương trình có thể tiếp tục. Điều này sẽ sáng tỏ hơn, nếu bạn di chuyển khối **try/ catch** lên trên **Func1(),** như theo thí dụ 12-3.

Thí dụ 12-3: Khối catch trên một hàm triệu gọi.

```
namespace Prog_CSharp
{   using System;
    public class Tester
    {   public static void Main()
```

```
      {  Console.WriteLine("Vào Main…");
         Tester t = new Tester();
         t.Func1();
         Console.WriteLine(Ra khỏi Main…");
      }  // end Main

      public void Func1()
      {  Console.WriteLine("Vào Func1…");
         try
         {  Console.WriteLine("Đi vào khối try…");
            Func2();
            Console.WriteLine("Ra khỏi khối try…");
         }
         catch
         {  Console.WriteLine("Biệt lệ bị chận bắt và thụ lý.");
         }
         Console.WriteLine("Ra khỏi Func1…");
      }  // end Func1

      public void Func2()
      {  Console.WriteLine("Vào Func2…");
         throw new System.Exception();
         Console.WriteLine("Ra khỏi Func2…");
      }  // end Func2
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Vào Main…
Vào Func1…
Đi vào khối try…
Vào Func2…
Biệt lệ bị chận bắt và thụ lý
Ra khỏi Func1…
Ra khỏi Main…

Lần này biệt lệ không được thụ lý trong **Func2()** mà trong hàm **Func1()**. Khi **Func2()** được triệu gọi, nó in ra Vào Func2… rồi tung ra một biệt lệ. Chương trình ngưng thi hành, và CLR đi tìm một bộ thụ lý biệt lệ, nhưng không có trong **Func2()**. Call stack được cho trải ra, và CLR tìm thấy bộ thụ lý biệt lệ ở **Func1().** Lệnh **catch** được gọi vào, và việc thi hành chương trình tiếp tục ngay liền sau **catch**, in ra Ra khỏi Func1.. rồi Ra khỏi Main…bạn chắc vì sao **Đi vào khối try…** và **Ra khỏi Func2…** không được in ra.

12.2.2.3 Tạo những lệnh catch "có địa chỉ"

Tới đây, bạn chỉ mới làm quen với lệnh **catch** mang tính chung chung. Bạn có thể tạo những lệnh **catch** "có địa chỉ" (dedicated), cho phép chỉ thụ lý một vài biệt lệ nhưng lại không đối với những biệt lệ khác, dựa trên loại biệt lệ được tung ra. Thí dụ 12-4 minh họa làm thế nào khai báo biệt lệ mà bạn muốn thụ lý.

Thí dụ 12-4: Khai báo loại biệt lệ muốn thụ lý.

```
namespace Prog_CSharp
{   using System;

    public class Tester
    {   public static void Main()
        {   Tester t = new Tester();
            t.TestFunc();
        }   // end Main

        // Thử chia hai con số, và xử lý những biệt lệ có thể xảy ra
        public void TestFunc()
        {   try
            {   double a = 5;
                double b = 0;
                Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
            }

            // đầu tiên những biệt lệ xảy ra nhiều nhất
            catch (System.DivideByZeroException)
            {   Console.WriteLine("Chận đúng DivideByZeroException!");
            }

            catch (System.ArithmeticException)
            {   Console.WriteLine(Chận đúng System.ArithmeticException!");
            }

            // cuối cùng là exception generic
            catch
            {   Console.WriteLine("Chận đúng biệt lệ vô danh !");
            }
        }   // end TestFunc

        // Làm bài tóan 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;
        }   // end DoDivide
    }   // end Tester
```

```
}   // end Prog_CSharp
```

Kết xuất

Chận đúng DivideByZeroException!

Trong thí dụ này, hàm **DoDivide()** sẽ không để cho bạn chia zero bởi một con số khác, mà cũng không để cho bạn chia một số cho zero. Nó sẽ tống ra một thể hiện **DivideByZeroException** nếu bạn cố chia cho zero. Nếu bạn cố chia zero cho một số khác, sẽ không có biệt lệ thích ứng, vì chia zero cho một số là hợp lệ, nên sẽ không có biệt lệ. Để làm vui lòng bạn, chúng tôi giả định là bạn không muốn có việc chia zero cho một số, nếu có thì cho tung ra biệt lệ **Arithmetic Exception**.

Khi một biệt lệ được tung ra, CLR sẽ quan sát từng bộ thụ lý biệt lệ một *theo thứ tự*, và so khớp với biệt lệ đầu tiên nếu có thể được. Khi bạn cho chạy chương trình với **a=5** và **b=7**, kết quả sẽ là: 5/7 = 0.7142857142857143.

Như chờ đợi, không biệt lệ nào được tung ra. Tuy nhiên, khi bạn cho **a=0**, thì kết quả sẽ là: **ArithmeticException caught!**

Biệt lệ được tung ra, CLR sẽ quan sát biệt lệ đầu tiên **DivideZeroException**. Thấy không khớp, CLR xuống bộ thụ lý biệt lệ kế tiếp, **ArithmeticException**, thì thấy khớp.

Cuối cùng, nếu bạn đổi **a=7** và **b=0**, thì lần này biệt lệ được tung ra là **DivideByZero Exception.**

Bạn để ý: bạn đặc biệt cẩn thận với thứ tự các lệnh **catch**, vì **DivideZeroException** được dẫn xuất từ **ArithmeticException**. Nếu bạn đảo ngược các lệnh **catch**, **Divide ZeroException** sẽ khớp với bộ thụ lý biệt lệ **ArithmeticException** và biệt lệ sẽ chả bao giờ đến bộ thụ lý biệt lệ **DivideZeroException.** Thật thế, nếu thứ tự các lệnh **catch** bị đảo ngược, thì khó lòng bất cứ biệt lệ nào có thể đi đến bộ thụ lý biệt lệ **DivideZero Exception.** Trình biên dịch sẽ nhận ra là **DivideZeroException** không thể đạt được và sẽ báo cáo sai lầm biên dịch.

Bạn có thể phân bố các lệnh **try/catch**, chận hứng một vài biệt lệ đặc thù trong một hàm, và những biệt lệ chung chung trong những hàm triệu gọi cao hơn. Việc sắp đặt thế nào tùy theo mục tiêu thiết kế của bạn.

Giả sử bạn có một hàm A triệu gọi một hàm B khác, và B lại triệu gọi hàm C. C lại gọi D, và D lại gọi E. Hàm E nằm trong sâu thẳm, còn các hàm B và A nằm ở vòng ngoài. Nếu bạn tiên liệu rằng hàm E sẽ tung ra biệt lệ, thì bạn phải tạo một khối **try/block** nằm sâu trong chương trình để chận hứng biệt lệ càng sát càng tốt nơi vấn đề có thể xảy

ra. Ngoài ra, đối với những biệt lệ "không mời mà đến" thì bạn chọn tạo những bộ thụ lý biệt lệ chung chung hơn nằm ở phía trên vòng ngoài.

12.2.3 Lệnh try-finally và try-catch-finally

Trong vài tình huống, việc tung ra một exception và cho trải ra call stack có thể gây ra vấn đề. Thí dụ, nếu bạn đã cho mở một tập tin hoặc nói cách khác bạn đã "điều động" nguồn lực (resource), bạn phải có cơ may đóng lại tập tin hoặc "tuôn đổ" (flush) vùng ký ức đệm. Thật ra, trong C#, điều này khó trở thành vấn đề như đối với các ngôn ngữ khác như C++ chẳng hạn, vì ở C# dịch vụ "thu gom rác" (garbage collection) sẽ ngăn ngừa việc biệt lệ có thể gây ra "rò rỉ ký ức" (memory leak).

Tuy nhiên, trong tình huống bạn phải hành động gì đó không cần biết liệu xem một biệt lệ được tung ra, chẳng hạn đóng lại một tập tin, bạn có hai phương án để mà chọn. Cách tiếp cận thứ nhất là cho bao gồm hành động nguy hiểm vào trong khối **try/catch** rồi sau đó cho đóng lại tập tin trong cả hai khối **try** và **catch**. Tuy nhiên, việc sao chép lên đôi đoạn mã xem ra không đẹp đẽ cho mấy và thường gây ra sai lầm. Do đó, C# cung cấp một cách "ngon lành" hơn thông qua khối **finally**.

Khối **finally** rất hữu ích khi ta phải làm việc "dọn dẹp" bất cứ nguồn lực nào bị chiếm dụng được cấp phát trong khối **try**. Quyền điều khiển bao giờ cũng trao qua cho khối **finally** cho dù thoát khỏi khối bằng cách nào đi nữa. Câu lệnh khối **try-finally** mang cú pháp như sau:

```
try try-block finally finally-block
```

theo đấy:

try-block: chứa đoạn mã mà ta chờ đợi biệt lệ được tung ra
finally-block: chứa bộ thụ lý biệt lệ và đoạn mã dọn dẹp

Một kiểu sử dụng thông dụng là câu lệnh **try-catch-finally,** theo đấy sử dụng các nguồn lực[45] (resource) nằm trong khối **try,** xử lý các tình huống đặc biệt trên khối **catch**, và giải phóng các nguồn lực trên khối **finally**.

Đoạn mã trong khối **finally** được bảo đảm là được thi hành bất chấp biệt lệ có được tung ra hay không. Hàm **TestFunc()** trên thí dụ 12-5 mô phỏng việc mở tập tin như là hành động đầu tiên. Sau đó hàm tiến hành vài tác vụ tính toán, rồi sau đó cho đóng lại tập tin. Có khả năng thỉnh thoảng giữa việc đóng mở tập tin, một biệt lệ được tung ra. Nếu việc này xảy ra, có khả năng tập tin tiếp tục đang ở tình trạng mở. Cho dù việc gì xảy ra,

[45] Chúng tôi không dịch là "tài nguyên" như nhiều tác giả khác.

ta cũng muốn là cuối hàm này, tập tin phải được đóng lại, do đó hàm **file close** được di chuyển về khối **finally**, để có thể thi hành bất chấp biệt lệ có tung ra hay không.

Thí dụ 12-5: Sử dụng khối finally

```
namespace Prog_CSharp
{  using System;

   public class Tester
   {  public static void Main()
      {  Tester t = new Tester();
         t.TestFunc();
      }  // end Main

      // Thử chia hai con số, và xử lý những biệt lệ có thể xảy ra
      public void TestFunc()
      {  try
         {  Console.WriteLine("Mở tập tin ở đây");
            double a = 5;
            double b = 0;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
            Console.WriteLine("Hàng này có thể in ra hoặc không");
         }

         // đầu tiên những biệt lệ xảy ra nhiều nhất
         catch (System.DivideByZeroException)
         {  Console.WriteLine("Đúng biệt lệ DivideByZeroException!");
         }

         catch (System.ArithmeticException)
         {  Console.WriteLine(Đúng biệt lệ
               System.ArithmeticException!");
         }

         // cuối cùng là exception generic
         catch
         {  Console.WriteLine("Chận bắt biệt lệ vô danh!");
         }

         finally
         {  Console.WriteLine("Đóng lại tập tin ở đây.");
         }
      }  // end TestFunc

      // Làm bài toán 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;
      }  // end DoDivide
```

```
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất

```
Mở tập tin ở đây
Đúng biệt lệ DivideByZeroException!
Đóng lại tập tin ở đây.
```

Kết xuất khi b=12

```
Mở tập tin ở đây
5 / 12 = 0.4166666666666669
Hàng này có thể in ra hoặc không
Đóng lại tập tin ở đây.
```

Trong thí dụ này, khối **finally** được thêm vào. Bất kể một biệt lệ được tung ra hay không, khối **finally** sẽ được thi hành, do đó trong cả hai lần kết xuất bạn thấy thông điệp **Đóng lại tập tin ở đây.**

Bạn để ý: Một khối **finally** có thể được tạo có hoặc không những khối **catch**, nhưng **finally** đòi hỏi phải cho thi hành một khối **try**. Sẽ gây ra sai lầm khi thoát một khối **finally** theo các lệnh **break, continue, return**, hoặc **goto**.

12.3 Lớp System.Exception

Lớp **System.Exception** là kiểu dữ liệu căn bản của tất cả các biệt lệ. Lớp này có một số thuộc tính khá nổi tiếng mà tất cả các biệt lệ phải chia sẻ sử dụng:

- **Message** là một thuộc tính read-only kiểu string chứa mô tả lý do biệt lệ mà ta có thể hiểu được.

- **InnerException** là một thuộc tính read-only property kiểu **Exception**. Nếu trị của nó non-null, nó chỉ về biệt lệ đã gây ra biệt lệ hiện hành. Bằng không, trị nó là null, cho biết biệt lệ này không được gây ra bởi một biệt lệ khác. (Số đối tượng biệt lệ bị nối lại với nhau theo cách này có thể là vô chừng.)

12.4 Các đối tượng *Exception*

Cho đến giờ, bạn sử dụng biệt lệ như là một chuông báo động - sự hiện diện của biệt lệ cho biết có sai lầm – nhưng bạn chưa hề rớ tới hoặc xem xét bản thân đối tượng Exception. Đối tượng System.Exception có một số hàm hành sự và thuộc tính hữu ích, như theo bảng 12-1 sau đây

Public Properties	
HelpLink	Đi lấy/Đặt để một kết nối về tập tin help được gắn liền với biệt lệ.
InnerException	Đi lấy đối tượng biệt lệ **Exception** gây ra biệt lệ hiện hành.
Message	Đi lấy một thông điệp mô tả biệt lệ hiện hành.
Source	Đi lấy/Đặt để tên ứng dụng hoặc đối tượng gây ra sai lầm.
StackTrace	Đi lấy một biểu diễn chuỗi trên call stack vào lúc biệt lệ hiện hành bị tung ra.
TargetSite	Đi lấy hàm hành sự tung ra biệt lệ hiện hành.
Public Methods	
Equals (inherited from Object)	*Overloaded*. Xác định liệu xem hai thể hiện Object instances có bằng nhau không.
GetBaseException	Khi bị phủ quyết (overridden) trên một lớp được dẫn xuất, trả về **Exception** gốc (root) đã gây ra một hoặc nhiều biệt lệ đi sau.
GetHashCode (inherited from Object)	Dùng làm một hàm băm (hash function) đối với một kiểu dữ liệu đặc biệt, thích ứng đem sử dụng vào các giải thuật băm và cấu trúc dữ liệu tương tự như một hash table.
GetObjectData	Khi bị phủ quyết (overridden) trên một lớp được dẫn xuất, hàm cho đặt để SerializationInfo với thông tin liên quan đến biệt lệ.
GetType (inherited from Object)	Đi lấy Type của thể hiện hiện hành.
ToString	*Overridden*. Tạo và trả về một biểu diễn chuỗi của biệt lệ hiện hành.

Thí dụ thuộc tính **Message** cung cấp những thông tin liên quan đến biệt lệ, chẳng hạn vì sao biệt lệ được tung ra. Thuộc tính **Message** thuộc loại read-only; đoạn mã tung biệt lệ ra có thể đặt để thuộc tính **Message** như là đối mục đối với hàm constructor của biệt lệ. Thuộc tính **HelpLink,** thuộc loại read/write, cho bạn một kết nối (link) về một tập tin help đi gắn liền với biệt lệ. Thuộc tính **StackTrace**, thuộc loại read-only, chỉ hiệu lực vào lúc chạy. Trong thí dụ 12-6, thuộc tính **Exception.HelpLink** sẽ được đặt để và được tìm thấy cung cấp thông tin cho người sử dụng liên quan đến **DivideByZero Exception**. Thuộc tính **StackTrace** được dùng để cung cấp một stack trace (ngăn chồng theo dõi) đối với lệnh sai lầm**. Stack Trace** sẽ cho hiển thị call stack: một loạt các triệu gọi hàm dẫn đến hàm đã tung ra biệt lệ.

Thí dụ 12-6: Làm việc với một đối tượng Exception

```
namespace Prog_CSharp
{  using System;

   public class Tester
   {  public static void Main()
      {  Tester t = new Tester();
         t.TestFunc();
      }  // end Main

      // Thử chia hai con số, và xử lý những biệt lệ có thể xảy ra
      public void TestFunc()
      {  try
         {  Console.WriteLine("Mở tập tin ở đây");
            double a = 12;
            double b = 0;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
            Console.WriteLine("Hàng này có thể in ra hoặc không ");
         }

         // đầu tiên những biệt lệ xảy ra nhiều nhất
         catch (System.DivideByZeroException e)
         {  Console.WriteLine("\nDivideByZeroException! Msg: {0}",
               e.Message);
            Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
            Console.WriteLine("\nĐây là một stack trace: {0}\n",
               e.StackTrace);
         }

         catch (System.ArithmeticException)
         {  Console.WriteLine(Đúng biệt lệ
               System.ArithmeticException !");
         }

         // cuối cùng là exception generic
         catch
         {  Console.WriteLine("Chận bắt biệt lệ vô danh!");
         }

         finally
         {  Console.WriteLine("Đóng lại tập tin ở đây.");
         }
      }  // end TestFunc

      // Làm bài tóan chia nếu hợp lệ
      public double DoDivide(double a, double b)
      {  if (b == 0)
         {  DivideByZeroException e = new DivideByZeroException();
            e.HelpLink = "http://www.libertyassociates.com";
            throw e;
         }
         if (a == 0)
```

```
            throw new System.ArithmeticException();
         return a/b;
      }  // end DoDivide
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Mở tập tin ở đây

DivideByZeroException! Msg: Attempted to divide by zero.

HelpLink: http://www.libertyassociates.com

Đây là một stack trace:
at Prog_CSharp.Tester.DoDivide(Double a, Double b)
    in c:\...exception06.cs:line 56
at Prog_CSharp.Testet.TestFunc()
    in.. exception06.cs:line 22

Đóng lại tập tin ở đây.

Trên phần kết xuất, stack trace sẽ liệt kê các hàm hành sự theo thứ tự ngược lại thứ tự được triệu gọi; nghĩa là cho thấy sai lầm xuất hiện ở **DoDivide**, được triệu gọi bởi **TestFunc()**. Khi các hàm nằm trong sâu, stack trace có thể giúp bạn hiểu thứ tự triệu gọi hàm.

Trong thí dụ này, thay vì đơn giản tung ra một **DivideByZeroException,** bạn tạo một thể hiện biệt lệ **e**:

```
DivideByZeroException e = new DivideByZeroException();
```

bạn không trao một thông điệp tự tạo, do đó thông điệp mặc nhiên được in ra:

```
DivideByZeroException! Msg: Attempted to divide by zero.
```

bạn có thể thay đổi dòng lệnh này để thay thế thông điệp mặc nhiên:

```
DivideByZeroException e = new DivideByZeroException("bạn cố chia cho
zero, thật là vô nghĩa!!!");
```

Trong trường hợp này, kết xuất sẽ phản ảnh thông điệp tự tạo của bạn.

```
DivideByZeroException! Msg: bạn cố chia cho zero, thật là vô nghĩa!!!
```

Trước khi tung biệt lệ, bạn cho đặt để thuộc tính **HelpLink**:

```
e.HelpLink = "http://www.libertyassociates.com";
```

Khi biệt lệ bị chận hứng, chương trình in ra thông điệp và **HelpLink**:

```
catch (System.DivideByZeroException e)
{  Console.WriteLine("\nDivideByZeroException! Msg: {0}",
            e.Message);
   Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
```

Điều này cho phép bạn cung cấp những thông tin hữu ích cho người sử dụng. Ngoài ra, chương trình in ra nôi dung của **StackTrace** bằng cách lấy thuộc tính **StackTrace** của đối tượng biệt lệ **e:**

```
Console.WriteLine("\nĐây là một stack trace: {0}\n", e.StackTrace);
```

Kết xuất của triệu gọi này phản ảnh toàn bộ nội dung của **StackTrace** dẫn đến lúc biệt lệ được tung ra:

```
Đây là một stack trace:
at Prog_CSharp.Tester.DoDivide(Double a, Double b)
   in c:\...exception06.cs:line 56
at Prog_CSharp.Testet.TestFunc()
   in.. exception06.cs:line 22
```

Chúng tôi đã cắt ngắn lối tìm về (path), nếu bạn in ra có thể là hơi khác một chút.

12.5 Những biệt lệ "cây nhà lá vườn"

Những kiểu biệt lệ "bẩm sinh" mà CLR cung cấp, cộng thêm những thông điệp tự tạo (custom message) mà ta thấy trong thí dụ đi trước cũng đủ chán để cung cấp thông tin cho khối **catch** khi một biệt lệ được tung ra. Tuy nhiên, cũng có lúc bạn muốn cung cấp thông tin phong phú hơn, hoặc cần đến những khả năng đặc biệt trong biệt lệ của bạn; hạn chế duy nhất là nó phải được dẫn xuất (trực tiếp hoặc gián tiếp) từ **System.ApplicationException**. Thí dụ 12-7 minh họa việc tạo một biệt lệ "cây nhà lá vườn" (custom exception).

Thí dụ 12-7: Tạo một biệt lệ "cây nhà lá vườn"

```
namespace Prog_CSharp
{  using System;

   public class MyCustomException: System.ApplicationException
   {  public MyCustomException(string message): base(message)
      {
```

```
      }
   }  // end MyCustomException

   public class Tester
   {  public static void Main()
      {  Tester t = new Tester();
         t.TestFunc();
      }  // end Main

      // Thử chia hai con số, và xử lý những biệt lệ có thể xảy ra
      public void TestFunc()
      {  try
         {  Console.WriteLine("Mở tập tin ở đây");
            double a = 0;
            double b = 5;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
            Console.WriteLine("Hàng này có thể in ra hoặc không.");
         }

         // đầu tiên những biệt lệ xảy ra nhiều nhất
         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);
         }

         // cuối cùng là exception generic
         catch
         {  Console.WriteLine("Chận bắt biệt lệ vô danh !");
         }

         finally
         {  Console.WriteLine("Đóng lại tập tin ở đây.");
         }
      }  // end TestFunc

      // Làm bài tóan chia nếu hợp lệ
      public double DoDivide(double a, double b)
      {  if (b == 0)
         {  DivideByZeroException e = new DivideByZeroException();
            e.HelpLink = "http://www.libertyassociates.com";
            throw e;
         }
         if (a == 0)
         {  MyCustomException e = new MyCustomException(
               "Can't have zero divisor" );
            e.HelpLink =
"http://www.libertyassociates.com/NoZeroDivisor.htm";
```

```
            throw e;
         }
         return a/b;
      }  // end DoDivide
   }  // end Tester
}  // end Prog_CSharp
```

MyCustomException được dẫn xuất từ **System.ApplicationException** và chỉ gồm độc nhất một hàm constructor lấy một chuỗi **message** được trao qua cho lớp cơ bản. Trong trường hợp này, lợi điểm khi tạo lớp biệt lệ tự tạo là nó phản ảnh trung thực thiết kế đặc biệt của lớp **Tester**, theo đấy là bất hợp lệ khi có một zero divisor. Sử dụng **ArithmeticException** thay vì một custom exception, cũng tốt thôi nhưng sẽ gây lúng túng đối với người sử dụng vì một zero divisor thường không thể cho là một sai lầm số học.

12.6 Tung lại biệt lệ

Có thể bạn muốn khối **catch** của bạn tiến hành một số hoạt động sửa sai rồi sau đó tung lại biệt lệ cho một khối **try** nằm ngoài (trong một hàm triệu gọi). Có thể khối này tung lại *cùng biệt lệ,* hoặc nó tung một biệt lệ khác đi. Nếu nó tung một biệt lệ khác đi, có thể nó muốn đặt lọt thỏm biệt lệ nguyên thủy trong lòng biệt lệ mới như vậy hàm triệu gọi có thể hiểu "lịch sử" biệt lệ. Thuộc tính **InnerException** của biệt lệ mới sẽ tìm thấy lại biệt lệ nguyên thủy.

Vì **InnerException** cũng là một biệt lệ, có thể nó cũng có một biệt lệ nằm trong sâu. Như vậy, một chuỗi biệt lệ có thể nằm lồng với nhau, biệt lệ này nằm trong biệt lệ kia, giống như những con búp bê Ukrain. Thí dụ 12-8 minh họa việc sử dụng biệt lệ nằm sâu và tung ra lại biệt lệ.

Thí dụ 12-8: Sử dụng biệt lệ nằm sâu và tung ra lại biệt lệ.

```
namespace Prog_CSharp
{  using System;

   public class MyCustomException: System.ApplicationException
   {  public MyCustomException(string message, Exception inner):
         base(message, inner) {}
   }  // end MyCustomException

   public class Tester
   {  public static void Main()
      {  Tester t = new Tester();
         t.TestFunc();
```

```
}  // end Main

// Thử chia hai con số, và xử lý những biệt lệ có thể xảy ra
public void TestFunc()
{  try
   {  DangerousFunc1();
   }

   // nếu chận bắt một custom exception,
   // thì in ra exception history
   catch (MyCustomException e)
   {  Console.WriteLine("\n{0}", e.Message);
      Console.WriteLine("Truy tìm lại lịch sử biệt lệ ");
      Exception inner = e.InnerException;
      while (inner != null)
      {  Console.WriteLine("{0}", inner.Message);
         inner = inner.InnerException;
      }
   }
}  // end TestFunc


public void DangerousFunc1()
{  try
   {  DangerousFunc2();
   }

   // nếu chận hứng bất cứ exception nào ở đây,
   // thì tung custom exception
   catch (System.Exception e)
   (  MyCustomException ex = new MyCustomException(
         "E3 - Tình hình Custom Exception!", e);
      throw ex;
   )
}  // end DangerousFunc1

public void DangerousFunc2()
{  try
   {  DangerousFunc3();
   }

   // nếu chận hứng một DivideByZeroException, thì cho sửa sai
   // rồi tung ra một exception tổng quát
   catch (System.DivideByZeroException e)
   (  Exception ex = new Exception("E2 - Func2 bị chận bắt
                                    chia cho zero", e);
      throw ex;
   )
}  // end DangerousFunc2

public void DangerousFunc3()
{  try
   {  DangerousFunc4();
```

```
            }

            catch (System.ArithmeticException)
            (   throw;
            )

            catch (System.Exception)
            {   Console.WriteLine("Biệt lệ được thụ lý ở đây.");
            }
        }   // end DangerousFunc3

        public void DangerousFunc4()
        {   throw new DivideByZeroException("E1 - DivideByZero Exception");
        }   // end DangerousFunc4
    }   // end Tester
}   // end Prog_CSharp
```

Kết xuất

```
E3 – Tình hình Custom Exception!
Truy tìm lại lịch sử biệt lệ …
E2 – Func2 bị chận bắt chia cho zero
E1 – DivideByZeroException
```

Vì đoạn mã này bị "lột bỏ tận xương" nên phần kết xuất có thể làm bạn gãi tai gãi đầu. Cách tốt nhất xem đoạn mã này hoạt động ra sao là dùng debugger đi lần từng bước một xuyên qua đoạn mã.

bạn bắt đầu triệu gọi **DangerousFunc1()** trên khối **try**:

```
try
{   DangerousFunc1();
}
```

DangerousFunc1() triệu gọi **DangerousFunc2()**, hàm này đến phiên lại gọi **DangerousFunc3()**, và hàm này đến phiên gọi **DangerousFunc4()**. Tất cả các hàm này đều có riêng những khối **try**. Ở cuối "đường hầm", **DangerousFunc4()** tung ra biệt lệ **DivideByZeroException.** Thông thường, **System.DivideByZeroException** có riêng cho mình thông điệp sai lầm, nhưng bạn có thể chuyển qua một thông điệp "cây nhà lá vườn". Để dễ nhận diện trình tự các biến cố, thông điệp tự tạo được trao qua là **E1 - DivideByZero Exception**.

Biệt lệ được tung ra ở **DangerousFunc4()** bị chận hứng ở **DangerousFunc3.** Lô gic trong **DangerousFunc3** là nếu bất cứ biệt lệ **ArithmeticException** bị chận hứng (chẳng hạn **DivideByZeroException**) nó không làm gì cả, đơn giản nó tung lộn lại exception:

```
catch(System.ArithmeticException)
(   throw;
}
```

Cú pháp để tung lại đúng biệt lệ (không thay đổi nó) là chỉ dùng vỏn vẹn từ chốt **throw**.

Như vậy, biệt lệ được tung cho **DangerousFunc2()**, bị chận hứng, thực hiện một số sửa sai, và tung ra một biệt lệ mới kiểu **Exception**. Trong hàm constructor đối với biệt lệ mới này, **DangerousFunc2** trao qua một thông điệp tự tạo (E2 – Func2 caught divide by zero) và *biệt lệ nguyên thủy*. Như vậy, biệt lệ nguyên thủy (E1) trở thành **InnerException** đối với biệt lệ mới (E2). **DangerousFunc2** sau đó tung biệt lệ mới E2 này cho **DangerousFunc1**.

DangerousFunc1 chận hứng biệt lệ, làm gì đó, rồi tạo một biệt lệ mới kiểu **MyCustomException**, chuyển cho hàm constructor một chuỗi mới (E3 – Custom Exception Situation!), và biệt lệ bị chận hứng ngay (E2). Bạn nhớ lại, biệt lệ vừa bị chận hứng là biệt lệ với một **DivideByZeroException** (E1) như là inner exception của nó. Tới đây, bạn có một biệt lệ kiểu **MyCustomException (E3)**, với một inner exception kiểu **Exception** (E2), đến phiên có một inner exception kiểu **DivideByZeroException** (E1). Sau đó tất cả đều tung về hàm **Tester** ở đây sẽ bị chận hứng.

Khi hàm **catch** chạy, nó in ra thông điệp:

```
E3 – Tình hình Custom Exception !
```

và rồi nó lần theo xuống các lớp sâu của những inner exception, in ra những thông điệp của chúng:

```
while (inner != null)
{   Console.WriteLine("{0}", inner.Message);
    inner = inner.InnerException;
}
```

Phần kết xuất phản ảnh chuỗi những biệt lệ được tung ra và bị chận hứng.

```
Truy tìm lại lich sử biệt lệ …
E2 – Func2 bị chận bắt chia cho zero
E1 – DivideByZeroException
```

Chương 13

Ủy thác và Tình huống

Khi một nguyên thủ quốc gia nào đó qua đời, vị chủ tịch nước vì một lý do gì đấy không thể đến dự lễ tang, nên thường ủy quyền (delegate) cho ai đó, đi thay mình. Thường là phó chủ tịch nước, nhưng thỉnh thoảng, vị phó chủ tịch nước này cũng bận, nên chủ tịch nước phải gởi một người khác, như bộ trưởng ngoại giao chẳng hạn. Chủ tịch nước không muốn "cột chặt cứng" sự ủy nhiệm vào một người duy nhất, mà ủy thác trách nhiệm cho ai đó có khả năng thi hành nghi thức quốc tế một cách hoàn hảo.

Chủ tịch nước định trước người nào sẽ được ủy quyền (đi dự lễ tang), những thông số nào sẽ được trao qua (lời chia buồn, lời khuyên thích ứng), và những gì có thể chờ đợi nhận được (sự thiện chí). Sau đó, chủ tịch nước sẽ chỉ định một người đặc biệt nhận trách nhiệm ủy quyền này vào "lúc chạy" (runtime) trong thời gian tại chức của chủ tịch.

Trong lập trình, bạn thường đứng trước tình huống bạn cần cho thi hành một điều đặc biệt gì đó, nhưng không biết trước sẽ dùng hàm hành sự nào, hoặc đối tượng nào mà bạn có thể triệu gọi để thi hành hành động này. Thí dụ, một nút ấn (button) nào đó có thể biết rằng nó phải báo cho một đối tượng *nào đó* khi nó bị ấn, nhưng có thể không biết đối tượng nào cần phải báo. Thay vì cột chặt nút vào một đối tượng đặc biệt, bạn sẽ cho gắn kết nút ấn này về một ***delegate***, rồi chuyển hàm hành sự đặc biệt cho delegate này khi chương trình chạy.

Trong những ngày đầu của thời kỳ "đồ đá" của ngành lập trình, chương trình thường thi hành một cách "thẳng tuột" từ đầu cho đến khi kết thúc. Nếu người sử dụng có can thiệp vào khi chương trình đang chạy, thì sự can thiệp này cũng rất thô thiển hạn chế, được kiểm soát nghiêm ngặt giới hạn vào việc điền dữ liệu vào một vài vùng mục tin.

Ngày nay, mô hình lập trình GUI (Graphical User Interface) sử dụng giao diện người sử dụng là chính, đòi hỏi một cách tiếp cận khác, được gọi dưới cái tên là *event-driven programming*, lập trình vận hành theo tình huống. Một chương trình "sành điệu" phải trình ra một "mặt tiền bắt mắt" gọi là user interface, giao diện người sử dụng, và chờ người sử dụng hành động. Người sử dụng có thể hành động theo nhiều cách, không tài nào biết trước được, chẳng hạn chọn mục trên trình đơn, ấn nút điều khiển, nhật tu các vùng dữ liệu văn bản, bấm tắt (click[46]) lên icon, v.v.. Mỗi hành động như thế sẽ gây nên (raise) một tình huống (event). Một số tình huống khác có thể được gây nên không cần

[46] Có người dịch là "nhắp chuột".

đến hành động trực tiếp của người sử dụng, chẳng hạn những tình huống gây ra bởi bộ đếm thời gian (timer), hoặc bởi email nhận được, hoặc việc sao chép tập tin hoàn tất.

Một tình huống là việc "gói ghém"một ý tưởng là "một việc gì đó sẽ xảy ra" mà chương trình phải đáp ứng. **Event** và **Delegate** là hai khái niệm quyện lấy nhau vì việc xử lý tình huống muốn uyển chuyển đòi hỏi việc đáp ứng trước tình huống phải được chuyển về (dispatched) hàm thụ lý tình huống[47] thích ứng. Trên C#, hàm thụ lý tình huống được thiết đặt như là một delegate.

Những delegate cũng được dùng như là callback (có nghĩa là "nhớ gọi lại giùm nhé!"), do đó một lớp có thể bảo một lớp khác "cậu làm giùm mình việc này nhé, xong rồi báo cho mình biết". Ngoài ra, delegate cũng có thể được dùng khai báo những hàm hành sự chỉ sẽ biết được vào giờ chót khi chương trình chạy, một đề mục mà chúng tôi sẽ triển khai trong các phần tới.

13.1 Ủy thác[48] (delegates)

Trên C#, delegate là những đối tượng loại "thượng đẳng", hoàn toàn được hỗ trợ bởi ngôn ngữ. Về mặt kỹ thuật, *delegate thuộc kiểu qui chiếu* cho phép lập trình viên gói ghém trong lòng một đối tượng delegate, một qui chiếu về một hàm hành sự với một dấu ấn (signature) riêng biệt và một kiểu dữ liệu được trả về. Sau đó, đối tượng delegate sẽ được trao qua cho đoạn mã theo đấy đoạn mã có thể triệu gọi hàm được qui chiếu, không cần biết đến vào lúc biên dịch hàm hành sự nào sẽ được triệu gọi. Bạn có thể hình dung delegate như là một hộp thư giao liên trong thời kỳ cách mạng. Bạn biết hộp thư ở đâu, nhưng bạn không biết là vào phút chót hộp thư sẽ chứa chỉ thị của ai và làm gì.

Bạn để ý: Trên C++ và trên các ngôn ngữ khác, bạn có thể thực hiện đòi hỏi trên bằng cách sử dụng con trỏ hàm (function pointer) và con trỏ chĩa vào các hàm thành viên (pointer to member function). Khác với function pointer, delegate hoàn toàn thiên đối tượng và an toàn về kiểu dữ liệu (type safe). Khác với pointer to member function, *delegate gói ghém cả thể hiện đối tượng lẫn hàm hành sự.*

Một khai báo delegate sẽ định nghĩa một lớp được dẫn xuất từ lớp **System.Delegate**. Một đối tượng delegate sẽ gói ghém một hoặc nhiều hàm hành sự, mỗi hàm được qui chiếu như là một thực thể khả dĩ triệu gọi được (callable entity). Đối với những hàm hành sự instance, một callable entity thường gồm cả hai một thể hiện và một hàm hành sự trên

[47] Event handler chúng tôi dịch là "thụ lý tình huống" giống như toà án thụ lý hồ sơ kiện. Chúng tôi không dùng từ "xử lý tình huống".

[48] Ủy quyền hay ủy thác hình như cũng tương tự. Có nhiều công ty kinh doanh giữ vai trò ủy thác trong nhiều vụ mua bán với ngoại quốc cho một số công ty khác. Do đó chúng tôi chọn từ ủy thác hơn là ủy quyền.

thể hiện này. Còn đối với những hàm hành sự static, một callable entity thường chỉ gồm một hàm hành sự cần được triệu gọi mà thôi. Chỉ cần một đối tượng delegate với một lô thích ứng những đối mục, ta có thể triệu gọi tất cả các hàm hành sự instance của delegate với lô đối mục này.

Một đặc tính khá lý thú và hữu ích của một thể hiện delegate là nó cóc cần biết hoặc quan tâm đến liên quan đến những lớp của những hàm hành sự mà nó gói ghém; điều duy nhất nó quan tâm là những hàm hành sự này có tương thích với kiểu dữ liệu của delegate hay không. Như vậy delegate rất ngon lành khi cần giao những triệu gọi "nặc danh".

Bạn có thể tạo một delegate thông qua từ chốt **delegate**, như theo cú pháp sau đây:

```
[attributes] [modifiers] delegate result-type identifier
([formal-parameters]);
```

theo đấy:

attributes (tùy chọn) sẽ được đề cập ở chương 4, tập 2 bộ sách này;
modifiers (tùy chọn) chỉ chấp nhận các từ chốt: **new, public, private, internal** và **protected**.
result-type: kiểu dữ liệu trả về, khớp với kiểu dữ liệu của hàm hành sự
identifier: tên delegate.
formal-parameters (tùy chọn): danh sách các thông số, nghĩa là dấu ấn của những hàm hành sự mà bạn có thể ủy quyền cho delegate..

Thí dụ:

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

Lệnh trên khai báo một delegate mang tên **WhichIsFirst** cho gói ghém bất cứ hàm hành sự nào nhận hai thông số kiểu **object**, và trả về một **int**.

Một khi delegate đã được khai báo xong, bạn có thể cho gói ghém một hàm hành sự thành viên vào với delegate này bằng cách cho hiển lộ delegate, trao một hàm hành sự khớp với dấu ấn và kiểu dữ liệu trả về.

13.1.1 Dùng delegate để khai báo những hàm hành sự vào lúc chạy

Bạn dùng delegate để khai báo những loại hàm hành sự nào đó mà bạn muốn dùng để thụ lý tình huống và thiết đặt những callback trong ứng dụng của bạn. Các delegate cũng

có thể được dùng để khai báo những hàm hành sự static hoặc những hàm hành sự thể hiện mà chúng không biết được cho tới khi chương trình chạy.

Callback là gì?

Windows API thường xuyên dùng các con trỏ hàm (function pointer) để tạo ra những thực thể được gọi là "callback function" gọi tắt là "callback". Sử dụng callback, lập trình viên có khả năng cấu hình một hàm lo báo cáo trở lại (do đó có chữ callback) cho một hàm khác trên ứng dụng. Với C++, callback đơn giản chỉ là một vị chỉ ký ức, thiếu sót những thông tin an toàn như số lượng (và kiểu dữ liệu) thông số, kiểu trị trả về và qui ước triệu gọi. Do đó, với C#, con trỏ hàm được thay thế bởi delegate, nên kỹ thuật callback sẽ an toàn hơn và mang tính lập trình thiên đối tượng hơn.

Thí dụ, giả sử bạn muốn tạo một lớp "túi chứa" (container) đơn giản mang tên **Pair** có thể đựng và sắp xếp bất cứ những đối tượng nào được bỏ vào "túi". Bạn không biết trước loại đối tượng nào mà **Pair** có thể chứa, nhưng bằng cách tạo những hàm hành sự trong lòng những đối tượng này, theo đấy việc sắp xếp (sorting) có thể được ủy thác; bạn có thể giao trách nhiệm sắp xếp cho ngay những đối tượng.

Mỗi loại đối tượng có cách sắp xếp riêng biệt; thí dụ, một **Pair** gồm toàn đối tượng **counter** có thể sẽ sắp xếp theo thứ tự số, trong khi một **Pair** gồm đối tượng **Buttons** thì có thể cho sắp xếp theo tên, theo thứ tự ABC. Như vậy, muốn thực hiện điều này, bạn buộc các đối tượng được trữ trên **Pair** phải có một hàm hành sự cho biết cách sắp xếp các đối tượng được chứa.

Bạn định nghĩa hàm hành sự bạn cần, bằng cách tạo ra một delegate lo định nghĩa dấu ấn và kiểu trị trả về của hàm mà đối tượng (nghĩa là **Button**) phải cung cấp cho phép **Pair** xác định đối tượng nào phải đi đầu, đối tượng nào đi sau.

Lớp **Pair** định nghĩa một delegate mang tên **WhichIsFirst** như sau:

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

Ở đây, bạn thấy kiểu trả về là **comparison,** một kiểu enumeration được định nghĩa như sau:

```
public                          enum                          comparison
{   theFirstComesFirst                          =                        1,
    theSecondComesFirst                          =                       2
}
```

Hàm **Sort**, lo việc sắp xếp, sẽ nhận một thông số là một thể hiện của **WhichIsFirst**. Khi **Pair** cần biết làm cách nào sắp xếp thứ tự các đối tượng của mình, nó sẽ triệu gọi

delegate chuyển giao cho nó hai đối tượng như là thông số. Trách nhiệm quyết định xem trong hai đối tượng, đối tượng nào đi đầu sẽ được ủy thác cho hàm hành sự được gói ghém bởi delegate.

Để trắc nghiệm delegate, bạn sẽ tạo hai lớp: một lớp **Dog** (chó) và một lớp **Student** (sinh viên). **Dog** và **Student** không có đặc tính chi chung nhau, ngoại trừ việc cả hai thiết đặt những hàm hành sự mà **WhichIsFirst** có thể gói ghém, và do đó cả đối tượng **Dog** lẫn đối tượng **Student** đều là "ứng viên" được trữ trong lòng các đối tượng **Pair**.

Trong chương trình trắc nghiệm, bạn sẽ tạo một cặp **Student** và một cặp **Dog,** và cho trữ mỗi cặp trong **Pair**. Sau đó, bạn sẽ tạo ra những đối tượng delegate để gói ghém những hàm hành sự tương ứng khớp với dấu ấn và kiểu trị trả về, và bạn sẽ yêu cầu các đối tượng **Pair** sắp xếp các đối tượng **Dog** và **Student**. Ta thử làm theo từng bước một.

Bạn bắt đầu bằng cách tạo ra một hàm constructor **Pair** chịu nhận hai đối tượng và giấu chúng trên một bản dãy private:

```
public class Pair
{
    // hai đối tượng, được thêm theo thứ tự được nhận vào
    public Pair (object firstObject, object secondObject)
    {
        thePair[0] = firstObject;
        thePair[1] = secondObject;
    }
    // khai báo bản dãy dành trữ cả hai đối tượng
    private object[] thePair = new object[2];// bản dãy chứa 2 object
}
```

Tiếp theo, bạn cho nạp chồng (overriden) hàm **ToString** để có thể nhận trị kiểu chuỗi của hai đối tượng:

```
public override string ToString()
{
    return thePair[0].ToString() + ", " + thePair[1].ToString();
}
```

Bây giờ, bạn đã có hai đối tượng trong **Pair**, và bạn có thể in ra trị của chúng. Bạn đã sẵn sàng cho sắp xếp chúng và in ra kết quả sắp xếp. Bạn không thể biết trước loại đối tượng nào bạn sẽ có, do đó bạn ủy thác ngay cho các đối tượng trách nhiệm quyết định xem đối tượng nào vào trước trong **Pair** được sắp xếp. Do đó, bạn đòi hỏi mỗi đối tượng được trữ trong **Pair** thiết đặt một hàm cho biết một trong hai vào trước. Hàm sẽ lấy hai đối tượng (kiểu dữ liệu nào cũng được) và trả về một trị được "đánh số" (enumerated): **theFirstComesFirst** nếu đối tượng đầu tiên vào trước, và **theSecondComesFirst** nếu đối tượng thứ hai vào trước.

Những hàm được yêu cầu sẽ được gói ghém bởi delegate **WhichIsFirst** mà bạn đã định nghĩa trong lòng lớp **Pair**:

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

Trị trả về thuộc kiểu **comparison**, là một enum:

```
public enum comparison
{
   theFirstComesFirst = 1,
   theSecondComesFirst = 2
}
```

Bất cứ hàm static nào nhận hai đối tượng và trả về một **comparison** có thể được gói ghém bởi delegate này vào lúc chương trình chạy.

Bây giờ, bạn có thể định nghĩa hàm **Sort** cho lớp **Pair**:

```
public void Sort(WhichIsFirst theDelegateFunc)
{
   if (theDelegateFunc(thePair[0], thePair[1]) ==
        comparison.theSecondComesFirst)
   {
      object temp = thePair[0];  // ta tiến hành swap (tráo nhau)
      thePair[0] = thePair[1];
      thePair[0] = temp;
   }
}  // end Sort
```

Hàm **Sort** này nhận một thông số: một đối tượng delegate kiểu **WhichIsFirst** mang tên **theDelegateFunc.** Hàm **Sort** ủy thác trách nhiệm quyết định xem đối tượng nào vào trước trong hai đối tượng trong **Pair**, giao cho hàm được gói ghém bởi delegate lo. Trong thân hàm **Sort** nó triệu gọi hàm delegate, **theDelegateFunc** rồi quan sát trị trả về là một trong hai trị liệt kê kiểu **comparison**. Nếu trị trả về là **theSecondComesFirst**, thì các đối tượng trong **thePair** sẽ bị hoán vị (swap), bằng không sẽ không làm gì cả.

Bạn để ý **theDelegateFunc** là tên thông số tượng trưng cho hàm bị gói ghém bởi delegate. Bạn có thể gán bất cứ hàm hành sự nào (với trị trả về và dấu ấn thích ứng) cho thông số này. Đây giống như bạn có một hàm hành sự nhận một **int** làm thông số:

```
int SomeMethod(int myParam) {//…}
```

Tên thông số là **myParam**, nhưng bạn có thể trao bất cứ trị **int** hoặc biến nào . Cũng tương tự như thế, tên thông số trong thí dụ là **theDelegateFunc**, nhưng bạn có thể trao

qua bất cứ hàm hành sự nào đáp ứng trị trả về và dấu ấn được định nghĩa bởi delegate **WhichIsFirst**.

Bạn thử tưởng tượng bạn đang sắp xếp sinh viên theo thứ tự tên họ. Bạn viết một hàm hành sự trả về **theFirstComesFirst** nếu sinh viên đầu tiên vào trước, và **theSecondComesFirst** nếu là tên sinh viên thứ hai vào trước. Nếu bạn trao "Amy, Beth" thì hàm sẽ trả về **theFirstComesFirst**, còn nếu bạn trao "Beth, Amy" thì hàm sẽ trả về **theSecondComesFirst**. Nếu bạn trở lại **theSecondComesFirst,** hàm **Sort** sẽ đảo ngược các mục tin trong bản dãy, cho Amy về đầu và Beth về thứ hai.

Bây giờ bạn thêm một hàm nữa, **ReverseSort**, cho các mục tin trong bản dãy theo thứ tự ngược lại:

```
public void ReverseSort(WhichIsFirst theDelegateFunc)
{
    if (theDelegateFunc(thePair[0], thePair[1]) ==
                        comparison.theFirstComesFirst)
    {
        object temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
    }
}   // end ReverseSort
```

Chắc bạn hiểu logic của **ReverseSort**. Cả hai, **Sort** và **ReverseSort** đều dùng cùng **theDelegateFunc**, khỏi phải ép đối tượng hỗ trợ thêm một hàm trả về trị sắp xếp ngược lại.

Bây giờ, điều bạn cần là vài đối tượng để sắp xếp. Bạn tạo ra hai lớp đơn giản hơi vô duyên: **Student** và **Dog**. Vào lúc tạo bạn gán một cái tên cho những đối tượng:

```
public class Student
{
    public Student(string name)    // hàm constructor
    {
        this.name = name;
    }
}
```

Lớp **Student** đòi hỏi hai hàm hành sự, một phủ quyết (override) **ToString()** và một hàm khác được gói ghém như là hàm được ủy thác (delegated method).

Student phải phủ quyết **ToString**, như vậy hàm **ToString** trong **Pair**, đến phiên được triệu gọi đối với các đối tượng bị chứa sẽ làm việc đúng đắn hơn: việc thiết đặt đơn giản trả về tên sinh viên (là một đối tượng string):

```
public override string ToString()
```

```
{
   return name;
}
```

Ngoài ra, lớp **Student** cũng phải thiết đặt một hàm hành sự theo đấy **Pair.Sort** có thể ủy thác trách nhiệm xác định đối tượng nào vào trước:

```
return (String.Compare(s1.name, s2.name) < 0 ?
   comparison.theFirstComesFirst:
   comparison.theSecondComesFirst);
```

String.Compare là một hàm của .NET Framework đối với lớp **String**, cho phép so sánh hai chuỗi và trả về nhỏ thua zero nếu chuỗi đầu nhỏ thua chuỗi thứ hai, lớn hơn zero nếu chuỗi thứ hai lớn hơn, và zero nếu hai chuỗi bằng nhau. Bạn để ý, lôgic ở đây là trả về **theFirstComesFirst** chỉ khi nào chuỗi thứ nhất nhỏ thua, còn nếu bằng hoặc lớn hơn thì trả về **theSecondComesFirst.**

Bạn để ý hàm **WhichStudentComesFirst()** sau đây cũng nhận hai thông số là đối tượng và trả về một **comparison:**

```
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);
}
```

Điều này làm cho hàm xứng đáng là một hàm ủy thác **Pair.WhichIsFirst**, với dấu ấn và trị trả về khớp.

Lớp thứ hai phải tạo là **Dog**. Đối với lớp này, ta cho sắp xếp theo trọng lượng, con chó nào nhẹ cân đứng trước, nặng cân đúng sau. Sau đây là toàn bộ khai báo lớp **Dog**:

```
public class Dog
{
   public Dog(int weight)
   {
      this.weight = weight;
   }

   // chó được sắp xếp theo trọng lượng
   public static comparison WhichDogComesFirst(object o1, object o2)
   {   Dog d1 = (Dog) o1;
      Dog d2 = (Dog) o2;
      return d1.weight > d2.weight ?
         theSecondComesFirst: theFirstComesFirst;
   }
```

```
      public override string ToString()
      {   return weight.ToString();
      }
      private int weight;
   }
```

Bạn để ý là các hàm ủy thác của **Dog** và **Student** không mang cùng tên. Việc cho mang khác tên (WhichDogComesFirst và WhichStudentComesFirst) làm cho đoạn mã dễ hiểu hơn, và dễ bảo trì.

Thí dụ 13-1 sau đây là trọn vẹn chương trình minh hoạ việc sử dụng delegate

Thí dụ 13-1: Làm việc với delegate

```
namespace Prog_CSharp
{  using System;

   public enum comparison
   {  theFirstComesFirst = 1,
      theSecondComesFirst = 2
   }

   // một collection đơn giản chứa hai item
   public class Pair
   {  // khai báo delegate
      public delegate comparison WhichIsFirst(object obj1, object obj2);

      // hai đối tượng, được thêm vào hàm constructor
      // theo thứ tự được nhận vào
      public Pair (object firstObject, object secondObject)
      {  thePair[0] = firstObject;
         thePair[1] = secondObject;
      }

      // hàm lo sắp xếp hai đối tượng với bất cứ
      // tiêu chuẩn sắp xếp tùy thích
      public void Sort(WhichIsFirst theDelegateFunc)
      {  if (theDelegateFunc(thePair[0], thePair[1]) ==
            comparison.theSecondComesFirst)
         {  object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }  // end Sort

      // hàm lo sắp xếp hai đối tượng theo thứ tự ngược lại
      public void ReverseSort(WhichIsFirst theDelegateFunc)
      {  if (theDelegateFunc(thePair[0], thePair[1]) ==
            comparison.theFirstComesFirst)
         {  object temp = thePair[0];
```

```
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }  // end ReverseSort

      // yêu cầu hai đối tượng cho ra trị chuỗi
      public override string ToString()
      {  return thePair[0].ToString() + ", " + thePair[1].ToString();
      }

      // cho trữ cả hai đối tượng lên một bản dãy
      private object[] thePair = new object[2];
   }

   public class Dog
   {  public Dog(int weight)
      {  this.weight = weight;
      }

      // chó được sắp xếp theo trọng lượng
      public static comparison WhichDogComesFirst(Object o1, Object o2)
      {  Dog d1 = (Dog) o1;
         Dog d2 = (Dog) o2;
         return d1.weight > d2.weight ?
            theSecondComesFirst: theFirstComesFirst;
      }

      public override string ToString()
      {  return weight.ToString();
      }
      private int weight;
   }

   public class Student
   {  public Student(string name)
      {  this.name = name;
      }

      // sinh viên được sắp xếp theo ABC
      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;
      }
      private string name;
   }
```

```
   public class Tester
   {  static void Main()
      {  // tạo hai sinh viên và hai con chó, rồi thêm vào Pair,
         // rồi in ra
         Student Thien = new Student("Thien");
         Student Vinh = new Student("Vinh");
         Dog Milo = new Dog(65);
         Dog Fred = new Dog(12);
         Pair studentPair = new Pair(Thien, Vinh);
         Pair dogPair = new Pair(Milo, Fred);

         Console.WriteLine("studentPair\t\t\t: {0}",
                           studentPair.ToString());
         Console.WriteLine("dogPair\t\t\t: {0}", dogPair.ToString());

         // cho thể hiện những delegate
         Pair.WhichIsFirst theStudentDelegate = new Pair.WhichIsFirst(
            Student.WhichStudentComesFirst);
         Pair.WhichIsFirst theDogDelegate = new Pair.WhichIsFirst(
            Dog.WhichDogComesFirst);

         // sắp xếp sử dụng delegate
         studentPair.Sort(theStudentDelegate);
         Console.WriteLine("Sau hàm Sort studentPair\t\t: {0}",
                             studentPair.ToString());
         studentPair.ReverseSort(theStudentDelegate);
         Console.WriteLine("Sau hàm ReverseSort studentPair\t\t: {0}",
            studentPair.ToString());

         dogPair.Sort(theDogDelegate);
         Console.WriteLine("Sau hàm Sort dogPair\t\t: {0}",
                           dogPair.ToString());
         dogPair.ReverseSort(theDogDelegate);
         Console.WriteLine("Sau hàm ReverseSort dogPair\t: {0}",
            dogPair.ToString());
      }  // end Main
   }  // end Tester
}  // end Prog_CSharp*
```

Kết xuất

studentPair : Thien, Vinh
dogPair : 65, 12
Sau hàm Sort studentPair : Thien, Vinh
Sau hàm ReverseSort studentPair : Vinh, Thien
Sau hàm Sort dogPair : 12, 65
Sau hàm ReverseSort dogPair : 65, 12

Chương trình **Tester** tạo hai đối tượng **Student** và hai đối tượng **Dog** và sau đó bỏ

vào thùng chứa **Pair**. Hàm constructor **Student** lấy một chuỗi đối với tên sinh viên, còn hàm constructor **Dog** thì lấy một số nguyên đối với trọng lượng con chó:

```
Student Thien = new Student("Thien");
Student Vinh = new Student("Vinh");
Dog Milo = new Dog(65);
Dog Fred = new Dog(12);
Pair studentPair = new Pair(Thien, Vinh);
Pair dogPair = new Pair(Milo, Fred);
Console.WriteLine("studentPair\t\t\t: {0}",
                  studentPair.ToString());
Console.WriteLine("dogPair\t\t\t: {0}", dogPair.ToString());
```

Sau đó cho in ra nội dung của hai thùng chứa **Pair** để xem thứ tự vào của các đối tượng. Kết xuất như theo chờ đợi:

```
studentPair             : Thien, Vinh
dogPair                 : 65, 12
```

Tiếp theo, ta cho thể hiện hai đối tượng delegate:

```
Pair.WhichIsFirst theStudentDelegate = new Pair.WhichIsFirst(
       Student.WhichStudentComesFirst);
Pair.WhichIsFirst theDogDelegate = new Pair.WhichIsFirst(
       Dog.WhichDogComesFirst);
```

Delegate đầu tiên, **theStudentDelegate**, được tạo ra bằng cách trao qua hàm static thích ứng từ lớp **Student.** Còn delegate thứ hai, **theDogDelegate**, được trao qua một hàm static từ lớp **Dog**.

Giờ đây, những delegate là những đối tượng mà ta có thể trao qua cho các hàm. Bạn trao trước tiên cho hàm **Sort** rồi sau đó cho hàm **ReverseSort**, rồi cho in ra kết quả:

```
Sau hàm Sort studentPair        : Thien, Vinh
Sau hàm ReverseSort studentPair : Vinh, Thien
Sau hàm Sort dogPair            : 12, 65
Sau hàm ReverseSort dogPair     : 65, 12
```

13.1.2 Static Delegates

Điểm bất lợi trên thí dụ 13-1 là nó ép lớp triệu gọi, ở đây là **Tester**, cho thể hiện những delegate nó cần đến để sắp xếp những đối tượng trong **Pair**. Tốt hơn là cho thể hiện delegate từ lớp **Student** hoặc từ lớp **Dog**. Bạn có thể làm điều này bằng cách cấp cho mỗi lớp một hàm static delegate riêng. Do đó, bạn có thể thay đổi **Student** để thêm như sau:

```
public static readonly Pair.WhichIsFirt OrderStudents = new
                Pair.WhichIsFirt(Student.WhichStudentComesFirst);
```

Lệnh trên tạo ra một delegate static, read-only mang tên **OrderStudents**. Việc cho **OrderStudents** mang thuộc tính read-only làm cho một khi vùng mục tin static được tạo ra sẽ không bị thay đổi. Bạn cũng có thể tạo một delegate tương tự như với lớp **Dog**:

```
public static readonly Pair.WhichIsFirt OrderDogs = new
                          Pair.WhichIsFirt(Dog.WhichDogComesFirst);
```

Bây giờ mỗi lớp có một vùng mục tin static tương ứng. Mỗi vùng mục tin bị "cột cứng" vào hàm hành sự thích ứng trong lòng lớp. Bạn có thể triệu gọi delegate không cần khai báo một thể hiện delegate cục bộ (local delegate instance). Bạn chỉ cần trao qua mục delegate static của lớp:

```
studentPair.Sort(Student.OrderStudents);
Console.WriteLine("Sau hàm Sort studentPair\t\t: {0}",
                  studentPair.ToString());
studentPair.ReverseSort(Student.OrderStudents);
Console.WriteLine("Sau hàm ReverseSort studentPair\t\t: {0}",
                  studentPair.ToString());

dogPair.Sort(Dog.OrderDogs);
Console.WriteLine("Sau hàm Sort dogPair\t\t: {0}",
                 dogPair.ToString());
dogPair.ReverseSort(Dog.OrderDogs);
Console.WriteLine("Sau hàm ReverseSort dogPair\t\t: {0}",
   dogPair.ToString());
```

Kết xuất do việc thay đổi vừa kể trên cũng tương tự như thí dụ đi trước.

13.1.3 Delegate hoạt động như là thuộc tính

Vấn đề đối với static delegate, là nó phải được thể hiện, cho dù có sử dụng hay không, giống như **Student** và **Dog** trong thí dụ đi trước. Bạn có thể cải thiện các lớp này bằng cách biến các vùng mục tin static thành thuộc tính.

Đối với **Student** bạn cho thay thế khai báo

```
public static readonly Pair.WhichIsFirt OrderStudents = new
       Pair.WhichIsFirt(Student.WhichStudentComesFirst);
```

bởi:

```
public static Pair.WhichIsFirst OrderStudents
{   get
    {
        return new Pair.WhichIsFirst(WhichStudentComesFirst);
    }
}
```

Với lớp **Dog**, ta cũng làm như thế:

```
public static Pair.WhichIsFirst OrderDogs
{   get
    {   return new Pair.WhichIsFirst(WhichDogComesFirst);
    }
}
```

Khi thuộc tính **OrderStudent** được truy xuất, delegate được tạo ra:

```
return new Pair.WhichIsFirst(WhichStudentComesFirst);
```

Lợi điểm chủ yếu ở đây là delegate chỉ được tạo ra khi được yêu cầu. Như vậy cho phép lớp **Tester** xác định khi nào nó cần một delegate, nhưng vẫn cho phép những chi tiết tạo delegate thuộc trách nhiệm của lớp **Student** (hoặc **Dog**).

13.1.4 Cho đặt để thứ tự thi hành thông qua bản dãy delegate

Delegate có thể giúp bạn tạo một hệ thống theo đấy người sử dụng có thể quyết định thứ tự các tác vụ một cách linh động. Giả sử bạn có một hệ thống xử lý ảnh theo đấy một hình ảnh có thể được thao tác theo một số cách đã được định sẵn trước, như làm nhòe (blurring), làm cho sắc bén (sharpening), cho quay (rotating), cho lọc bỏ (filtering), v.v.. Bạn giả sử, thứ tự các effect này được áp dụng lên hình ảnh rất quan trọng. Người sử dụng muốn chọn từ một trình đơn các effect, cho tiến hành trên tất cả những gì họ thích, rồi ra lệnh cho image processor (bộ phận xử lý hình ảnh) cho chạy những effect này, effect này sau effect kia theo thứ tự họ đã định sẵn.

Bạn có thể tạo delegate cho mỗi tác vụ, rồi đưa chúng vào một collection được sắp xếp, như một bản dãy chẳng hạn, theo thứ tự mà bạn muốn cho thi hành. Một khi các

delegate được tạo ra, thêm vào collection, bạn chỉ cần đơn giản cho rảo qua bản dãy, mỗi lần triệu gọi hàm được ủy thác.

Bạn bắt đầu tạo một lớp **Image** tượng trưng cho hình ảnh mà bạn sẽ xử lý thông qua **ImageProcessor**.

```
public class Image
{   public Image()      // hàm constructor
    {   Console.WriteLine("Một hình ảnh được tạo ra");
    }
}
```

Bạn có thể tưởng tượng hình ảnh là một tập tin .gif hoặc .jpeg hoặc một hình ảnh khác. Bạn để ý là chúng tôi thường dùng một lệnh Console.WriteLine() xem như là chui vào hàm và đã thi hành xong những gì hàm phải thi hành.

Sau đó, **ImageProcessor** sẽ khai báo một delegate. Lẽ dĩ nhiên, bạn có thể định nghĩa delegate của bạn trả về bất cứ kiểu dữ liệu nào và nhận bất cứ thông số nào bạn thích. Đối với thí dụ này, bạn sẽ khai báo delegate gói ghém bất cứ hàm hành sự nào trả về **void** và không nhận thông số:

```
public delegate void DoEffect();     // delegate mang tên DoEffect
```

ImageProcessor sau đó khai báo một số hàm hành sự, mỗi hàm xử lý một **image** và mỗi hàm sẽ khớp trị trả về và dấu ấn của delegate.

```
public static void Blur()
{   Console.WriteLine("Blurring hình ảnh");
}

public static void Filter()
{   Console.WriteLine("Filtering hình ảnh");
}

public static void Sharpen()
{   Console.WriteLine("Sharpening hình ảnh");
}

public static void Rotate()
{   Console.WriteLine("Rotating hình ảnh");
}
```

Trong thực tế cuộc sống, các hàm này rất phức tạp, nhưng cuối cùng cũng thực hiện các động tác blurring, filtering, sharpening và rotating kể trên.

Lớp **ImageProcessor** cần đến một bản dãy để chứa những delegate mà người sử dụng đã chọn, một biến trong bản dãy tổng cộng bao nhiêu effect và lẽ dĩ nhiên một biến cho bản thân **image**.

```
DoEffect[] arrayOfEffects;
Image image;
int numEffectsRegistered = 0;
```

Ngoài ra, **ImageProcessor** cũng cần một hàm hành sự để đưa các delegate vào bản dãy:

```
public void AddToEffects(DoEffect theEffect)
{   if (numEffectsRegistered >= 10)
    {   throw new Exception("Có quá nhiều phần tử trên bản dãy");
    }
    arrayOfEffects[numEffectsRegistered++] = theEffect;
}
```

ImageProcessor lại cần đến một hàm hành sự khác lo luân phiên triệu gọi từng hàm một:

```
public void ProcessImages()
{   for (int i = 0; i < numEffectsRegistered; i++)
    {   arrayOfEffects[i] ();
    }
}
```

Cuối cùng, bạn chỉ cần khai báo các delegate static mà khách hàng có thể triệu gọi, gắn chúng vào những hàm xử lý**:**

```
public DoEffect BlurEffect = new DoEffect(Blur);
public DoEffect SharpenEffect = new DoEffect(Sharpen);
public DoEffect FilterEffect = new DoEffect(Filter);
public DoEffect RotateEffect = new DoEffect(Rotate);
```

Bạn để ý: Trong một môi trường sản xuất, theo đấy có thể bạn có hằng tá effects, có thể bạn nên chọn cho những effect này thành thuộc tính. Như vậy, ít rắc rối.

Đoạn mã khách hàng thường kèm theo một bộ phận GUI, nhưng chúng tôi cho mô phỏng bằng cách chọn những effect, đưa chúng vào bản dãy rồi cho gọi **ImageProcessor** như theo thí dụ 13-2 sau đây:

Thí dụ 13-2: Sử dụng một bản dãy delegate

```
namespace Prog_CSharp
{   using System;
```

```
// hình ảnh mà ta sẽ thao tác
public class Image
{   public Image()      // hàm constructor
    {   Console.WriteLine("Một hình ảnh được tạo ra");
    }
}

public class ImageProcessor
{   // khai báo delegate
    public delegate void DoEffect();

    // tạo những static delegate gắn với các hàm thành viên
    public DoEffect BlurEffect = new DoEffect(Blur);
    public DoEffect SharpenEffect = new DoEffect(Sharpen);
    public DoEffect FilterEffect = new DoEffect(Filter);
    public DoEffect RotateEffect = new DoEffect(Rotate);

    // hàm constructor khởi gán hình ảnh và bản dãy
    public ImageProcessor(Image image)
    {   this.image = image;
        arrayOfEffects = new DoEffect[10];
    }

    // trong môi trường sản xuất, ta dùng một collection uyển chuyển
    // hơn
    public void AddToEffects(DoEffect theEffect)
    {   if (numEffectsRegistered >= 10)
        {   throw new Exception("Có quá nhiều phần tử trên bản dãy");
        }
        arrayOfEffects[numEffectsRegistered++] = theEffect;
    }

    // các hàm xử lý hình ảnh
    public static void Blur()
    {   Console.WriteLine("Blurring hình ảnh");
    }

    public static void Filter()
    {   Console.WriteLine("Filtering hình ảnh");
    }

    public static void Sharpen()
    {   Console.WriteLine("Sharpening hình ảnh");
    }
    public static void Rotate()
    {   Console.WriteLine("Rotating hình ảnh");
    }
    public void ProcessImages()
    {   for (int i = 0; i < numEffectsRegistered; i++)
        {   arrayOfEffects[i] ();
        }
    }
```

```
        // các biến thành viên private
        DoEffect[] arrayOfEffects;
        Image image;
        int numEffectsRegistered = 0;
    }
    // Test driver
    public class Tester
    {   static void Main()
        {   Image theImage = new Image();
            ImageProcessor theProc = new ImageProcessor(theImage);
            theProc.AddToEffects(theProc.BlurEffect);
            theProc.AddToEffects(theProc.FilterEffect);
            theProc.AddToEffects(theProc.RotateEffect);
            theProc.AddToEffects(theProc.SharpenEffect);
            theProc.ProcessImages();
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

Một hình ảnh được tạo ra
Blurring hình ảnh
Filtering hình ảnh
Rotating hình ảnh
Sharpening hình ảnh

Trên lớp Tester của thí dụ 13-2, **ImageProcessor** được thể hiện và các effect được thêm vào. Nếu người sử dụng chọn cho nhoè nhoẹt (blurring) hình ảnh trước việc thanh lọc (filtering) hình ảnh, thì điều đơn giản là thêm delegate vào bản dãy theo thứ tự thích ứng. Cũng tương tự như thế, một tác vụ nào đó có thể được lặp lại thường xuyên theo ý muốn người sử dụng, đơn giản bằng cách thêm nhiều delegate vào collection.

Bạn có thể tưởng tượng cho hiển thị thứ tự những tác vụ trên một ô liệt kê (listbox) cho phép người sử dụng sắp xếp lại thứ tự các hàm hành sự, cho lên xuống trên ô liệt kê theo ý muốn. Khi các tác vụ bị thay đổi thứ tự, bạn chỉ cần thay đổi thứ tự sắp xếp trên collection. Bạn cũng có thể quyết định cho thu lượm (capture) thứ tự những tác vụ ghi lên một căn cứ dữ liệu, rồi sau đó nạp xuống một cách động, cho thể hiện các delegate như theo yêu cầu của các mẫu tin mà bạn đã trữ trên căn cứ dữ liệu.

Delegate cung cấp sự uyển chuyển trong việc xác định một cách linh động những hàm hành sự nào sẽ được triệu gọi, theo thứ tự nào và thường xuyên thế nào.

13.1.5 Multicasting

Tới đây, điều ta mong mỏi là *multicast*: nghĩa là triệu gọi hai hàm thiết đặt thông qua một delegate đơn lẻ. Điều này đặc biệt quan trọng khi ta thụ lý các tình huống (event), mà chúng tôi sẽ đề cập trong giây lát.

Mục đích là có một delegate đơn độc cho phép triệu gọi nhiều hàm hành sự hơn chỉ một. Điểm này khác với việc có một collection gồm delegate, mỗi delegate triệu gọi một hàm hành sự đơn lẻ. Trong thí dụ đi trước, collection dùng để sắp xếp thứ tự những delegate khác nhau. Ta có khả năng thêm một delegate đơn lẻ vào collection theo nhiều lần và sử dụng collection để sắp xếp lại các delegate để điều khiển thứ tự triệu gọi.

Với multicasting, bạn tạo một delegate đơn lẻ có thể triệu gọi nhiều hàm hành sự được gói ghém. Thí dụ khi bạn ấn một button, bạn muốn tiến hành nhiều tác vụ hơn là chỉ một. Bạn có thể thiết đặt điều này bằng cách cho button một collection gồm nhiều delegate, nhưng xem ra tạo một multicasting delegate đơn lẻ là dễ hơn và "sạch sẽ" hơn.

Bất cứ delegate nào trả về **void** là một multicast delegate, mặc dù bạn có thể coi nó như là một single-cast delegate. Hai multicast delegate có thể phối hợp nhau thông qua tác tử cộng (+). Kết quả là một multicast delegate mới triệu gọi các hàm hành sự nguyên thủy của cả hai gộp lại. Thí dụ, giả sử **Writer** và **Logger** là hai delegate trả về **void**. Dòng lệnh sau đây phối hợp chúng lại, cho ra một multicast delegate mới mang tên **myMulticastDelegate**:

```
myMulticastDelegate = Writer + Logger;
```

Bạn có thể thêm delegate vào multicast delegate bằng cách dùng tác tử +=. Tác tử này thêm delegate phía tay phải của tác tử vào multicast delegate phía tay trái. Thí dụ, giả sử **Transmitter** và **myMulticastDelegate** là những delegate, dòng lệnh sau đây thêm **Transmitter** vào **myMulticastDelegate**:

```
myMulticastDelegate += Transmitter;
```

Muốn biết có bao nhiêu multicast delegate được tạo ra và được sử dụng, ta thử rảo qua một thí dụ hoàn chỉnh. Trong thí dụ 13-3, bạn tạo một lớp mang tên **MyClass WithDelegate**, được định nghĩa như là một delegate trả về **void** và nhận một thông số kiểu string:

```
public delegate void StringDelegate(string s);
```

Sau đó, bạn định nghĩa một lớp mang tên **MyImplementingClass**, gồm 3 hàm hành sự, mỗi hàm trả về **void** và nhận một thông số kiểu string: **WriteString** (viết chuỗi lên một standard output)**, LogString** (viết lên một tập tin theo dõi) và **Transmitting** (mô

phỏng việc chuyển chuỗi lên mạng Internet). Bạn thể hiện những delegate triệu gọi những hàm hành sự thích ứng:

```
Writer("Chuỗi được chuyển qua Writer\n");
Logger("Chuỗi được chuyển qua Logger\n");
Transmitter("Chuỗi được chuyển qua Transmitter\n");
```

Muốn biết làm thế nào phối hợp những delegate, bạn tạo một thể hiện **Delegate** khác:

```
MyClassWithDelegate.StringDelegate myMulticastDelegate;
```

và gán cho nó kết quả việc "cộng" lại hai delegate hiện hữu:

```
myMulticastDelegate = Writer + Logger;
```

và thêm vào delegate này một delegate bổ sung, sử dụng tác tử +=:

```
myMulticastDelegate += Transmitter;
```

Cuối cùng, bạn có thể chọn gỡ bỏ delegate dùng đến tác tử -=:

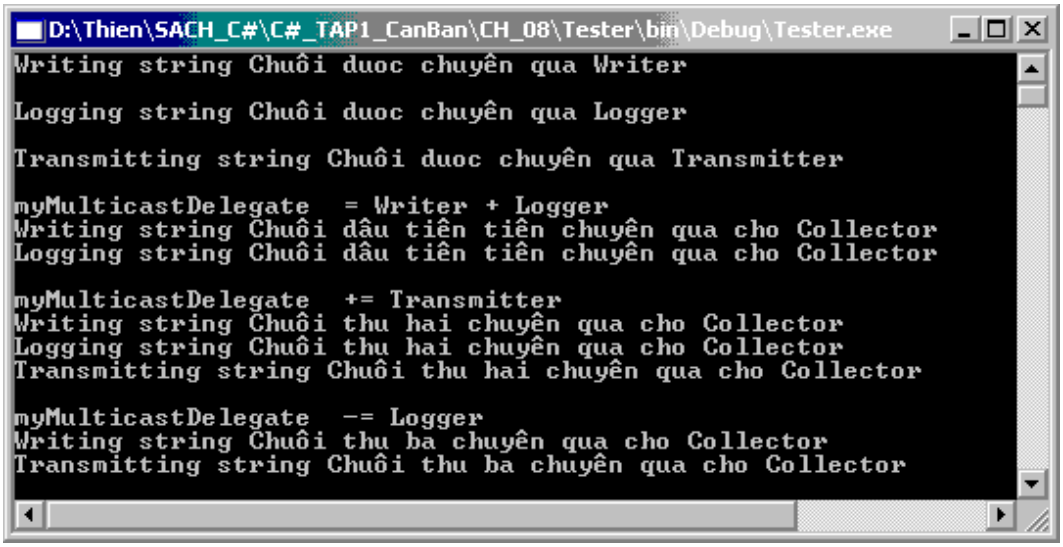
```
DelegateCollector -= Logger;
```

Thí dụ 13-3: Phối hợp các delegate

```
namespace Prog_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 Tester
```

```
   {  static void Main()
      {  // định nghĩa 3 đối tượng StringDelegate
         MyClassWithDelegate.StringDelegate Writer, Logger, Transmitter;

         // định nghĩa một đối tượng StringDelegate khác hoạt động như
         // một multicast delegate
         MyClassWithDelegate.StringDelegate myMultiCastDelegate;

         // thể hiện 3 delegate đầu tiên,
         // chuyển các hàm để được gói ghém
         Writer = new MyClassWithDelegate.StringDelegate(
                  MyImplementingClass.WriteString);
         Logger = new MyClassWithDelegate.StringDelegate(
                  MyImplementingClass.LogString);
         Transmitter = new MyClassWithDelegate.StringDelegate(
                  MyImplementingClass.TransmitString);

         // triệu gọi các hàm delegate
         Writer("Chuỗi được chuyển qua Writer\n");
         Logger("Chuỗi được chuyển qua Logger\n");
         Transmitter("Chuỗi được chuyển qua Transmitter\n");

         // báo cho người sử dụng biết bạn đang phối
         // hợp làm thành multicast delegate
         Console.WriteLine("myMulticastDelegate = Writer + Logger");
         myMulticastDelegate = Writer + Logger;

         // triệu gọi các hàm delegate, hai hàm sẽ được triệu gọi
         myMulticastDelegate("Chuỗi đầu tiên chuyển qua cho Collector");

         // báo cho người sử dụng bạn sắp chuyển delegate thứ 3 vào
         // multicast delegate
         Console.WriteLine( "\nmyMulticastDelegate += Transmitter");
         myMulticastDelegate += Transmitter;

         // triệu gọi 3 hàm delegate
         myMulticastDelegate("Chuỗi thứ hai chuyển qua cho Collector");

         // báo cho người sử dụng bạn sắp gỡ bỏ logger delegate
         Console.WriteLine( "\nmyMulticastDelegate -= Logger");
         myMulticastDelegate -= Logger;

         // triệu gọi hai hàm delegate còn lại
         myMulticastDelegate("Chuỗi thứ ba chuyển qua cho Collector");
      }  // end Main
   }  // end Tester
}  // end Prog_CSharp
```

Kết xuất



Trong phần Tester của thí dụ 13-3, các thể hiện delegate được định nghĩa và 3 đầu tiên (**Writer, Logger**, và **Transmitter**) được triệu gọi. Delegate thứ tư, **myMulticastDelegate**, sau đó được gán với phối hợp của hai delegate đầu tiên, được triệu gọi làm cho cả hai hàm delegate được triệu gọi. Delegate thứ ba được thêm vào, và khi **myMulticastDelegate** được triệu gọi thì cả 3 hàm delegate được triệu gọi . Cuối cùng, **Logger** bị gỡ bỏ và khi **myMulticastDelegate** được triệu gọi thì chỉ 2 hàm delegate còn lại được triệu gọi

Sức mạnh của multicast delegate sẽ được hiểu sâu khi đề cập đến tình huống (event). Khi một tình huống, như một button bị ấn xuống chẳng hạn, được gắn liền với multicast delegate, ta có thể triệu gọi một loạt những hàm thụ lý tình huống (event handler) đáp ứng đòi hỏi của tình huống.

13.2 Các tình huống (Events)

GUI (Graphical User Interface), Windows và Web Browsers đòi hỏi chương trình phải đáp ứng (response) trước những *tình huống*[49] (*event*). Một tình huống có thể là khi một button bị ấn xuống, một mục trình đơn được chọn, việc sao chép một tập tin được hoàn tất, v.v.. Nói một cách ngắn gọn, một điều gì đó xảy ra, bạn phải phản ứng lại trước biến cố này. Bạn không thể tiên liệu trước thứ tự mà các tình huống này sẽ nổi lên (arise).

[49] Chúng tôi không dịch là "biến cố".

Hệ thống bất động cho tới khi tình huống xảy ra, rồi nó bật lên hành động để thụ lý tình huống.

Trong một môi trường GUI, bất cứ số "lục lăng lục chốt" (widget[50]) nào cũng có thể gây ra một tình huống. Thí dụ, khi bạn bấm tắt (click) lên một nút ấn, có thể việc này sẽ gây nên tình huống **Click**. Khi bạn thêm một mục tin (item) vào một ô liệt kê kéo xuống (drop-down list), việc này có thể gây ra một tình huống **ListChanged**.

Các lớp khác sẽ được chú ý đến đáp ứng những tình huống này. Việc đáp ứng thế nào thì phía các lớp gây ra tình huống sẽ không quan tâm đến. Nút bảo "Ta bị click" và lớp đáp ứng sẽ phản ứng một cách thích hợp.

13.2.1 Bố cáo và Đăng ký thụ lý

Trên C#, bất cứ đối tượng nào cũng có thể "bố cáo" (*publish*) một loạt những tình huống mà các lớp khác có thể "đăng ký thụ lý" (*subscribe*). Khi lớp bố cáo gây nên một tình huống, thì tất cả các lớp đã đăng ký thụ lý sẽ được thông báo (notify). Với cơ chế này, đối tượng của bạn có thể bảo "Sau đây là những điều tôi có thể thông báo cho bạn biết", và các lớp khác có thể "ký giao ước" bảo rằng "Vâng, cho tớ biết khi nào xảy ra". Thí dụ, một button được gọi là ***publisher*** vì button bố cáo cho biết là tình huống **Click** sẽ xảy ra khi nó bị ấn xuống, và các lớp khác là những ***subscriber*** vì các lớp này đăng ký chịu thụ lý tình huống **Click**.

13.2.2 Tình huống và Uỷ thác

Trên C#, các tình huống được thiết đặt thông qua delegate. Lớp **publisher** định nghĩa một delegate, còn lớp **subscriber** thì phải lo thiết đặt delegate này. Khi một tình huống nổi lên, các hàm hành sự của lớp **subcriber** sẽ được triệu gọi thông qua delegate.

Một hàm hành sự thụ lý một tình huống được gọi là một *event handler* (hàm thụ lý tình huống). Bạn cũng có thể khai báo bất cứ delegate khác là những event handler của bạn.

Theo qui ước, trên .NET Framework, hàm thụ lý tình huống trả về **void** và nhận hai thông số: thông số thứ nhất là "nguồn" (source) gây ra tình huống; nghĩa là đối tượng publisher, còn thông số thứ hai là một đối tượng được dẫn xuất từ **EventArgs** (tắt Event Argument). Người ta khuyên hàm thụ lý tình huống của bạn nên theo cách thiết kế mẫu dáng (design pattern) này.

---

[50] Để chỉ những ô control trên biểu mẫu. Widget viết tắt chữ Window gadget.

EventArgs là lớp cơ bản đối với tất cả các dữ liệu tình huống (event data). Ngoài hàm constructor của nó, lớp **EventArgs** thừa kế tất cả các hàm hành sự của nó từ **Object**, mặc dù nó có thêm một vùng mục tin static mang tên **empty** tượng trưng cho một tình huống không trạng thái (state). Những lớp được dẫn xuất từ **EventArgs** thường chứa thông tin liên quan đến tình huống.

Các tình huống là những thuộc tính của lớp bố cáo tình huống. Từ chốt **event** điều khiển cách thuộc tính tình huống được truy xuất thế nào bởi những lớp bố cáo. Từ chốt **event** được thiết kế để duy trì khái niệm publish/subscribe.

Giả sử bạn muốn tạo một lớp **Clock** sử dụng tình huống để thông báo cho những ai đăng ký biết khi thời gian tại chỗ thay đổi trị, mỗi lần một giây. Ta gọi tình huống này là **OnSecondChange**. Bạn khai báo tình huống và kiểu event handler delegate như sau:

[*attribute*] [*modifiers*] **event** *type member-name*

Thí dụ:

```
public event SecondChangeHandler OnSecondChange
```

Thí dụ này không có *attribute* (chúng tôi sẽ đề cập sau attribute). *Modifier* có thể là **abstract, new, override, static, virtual**, hoặc một trong 4 access modifier (**public, private, protected** và **internal**), trong trường hợp này là **public**. Theo sau modifier là từ chốt **event**. *Type* là delegate mà bạn sẽ gán tình huống, trong trường hợp này là **SecondChangeHandler**. *Member-name* là tên tình huống, và trong trường hợp này là **OnSecondChange**. Theo tập quán lập trình ta bắt đầu đặt tên cho tình huống bởi từ **On**. Nói tóm lại, khai báo trên cho biết **OnSecondChange** là một tình huống được thiết đặt bởi một delegate kiểu **SecondChangeHandler**.

Khai báo đối với **SecondChangeHandler** delegate sẽ như sau:

```
public delegate void SecondChangeHandler(object clock,
                                TimeInfoEventArgs timeInformation);
```

Lệnh trên khai báo một delegate. Như đã nói trên, theo qui ước, một hàm thụ lý tình huống phải trả về **void** và phải có hai thông số: nguồn tình huống (ở đây là **clock**), và một đối tượng được dẫn xuất từ **EventArgs**, trong trường hợp này là **TimeInfo EventArgs. 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 chứa thông tin liên quan đến giờ phút giây hiện hành. Nó định nghĩa một hàm constructor, và 3 biến số nguyên public và read-only.

Ngoài một delegate và một event, một đối tượng **Clock** có 3 biến thành viên: **hour, minute** và **second**. Và chỉ có duy nhất một hàm hành sự, **Run()**:

```
public void Run()
{   for(;;)
    {   // ngủ đi 10 ms
        Thread.Sleep(10);
    // đi lấy thời gian hiện hành
        System.DateTime dt = System.DateTime.Now;

        // nếu giây thay đổi báo cho subscribers
        if (dt.Second != second)
        {   // tạo đối tượng TimeInfoEventArgs để chuyển cho subscriber
            TimeInfoEventArgs timeInformation = new TimeInfoEventArgs(
        dt.Hour, dt.Minute, dt.Second);

            // nếu có ai đã đăng ký, thông báo cho họ biết
            if (OnSecondChange != null)
            {   OnSecondChange(this, timeInformation);
            }
        }
        // nhật tu trạng thái
        this.second = dt.Second;
        this.minute = dt.Minute;
        this.hour = dt.Hour;
    }   // end for
}   // end Run
```

Run() tạo một vòng lặp vô tận, cho kiểm tra theo định kỳ thời gian hệ thống. Nếu có thay đổi từ thời gian hiện hành của đối tượng **Clock**, nó sẽ loan báo cho tất cả các subscriber, rồi cho nhật tu trạng thái riêng của mình. Bước đầu tiên là cho ngủ 10 ms:

```
Thread.Sleep(10);
```

Lệnh trên sử dụng một hàm static thuộc lớp **Thread** từ namespace **System.Threading**. Việc triệu gọi **Sleep()** ngăn không cho vòng lặp chạy quá gắt đến nỗi hệ thống không làm ăn gì được. Sau khi ngủ xong 10 ms, hàm **Run()** kiểm tra thời gian hiện hành:

```
System.DateTime dt = System.DateTime.Now;
```

Cứ mỗi 100 lần hàm kiểm tra, **second** sẽ tăng, và hàm thông báo sự thay đổi này cho subscriber. Muốn thế, hàm trước tiên sẽ tạo một đối tượng **TimeInfoEventArgs** mới:

```
if (dt.Second != second)
   {  // tạo đối tượng TimeInfoEventArgs để chuyển cho subscriber
      TimeInfoEventArgs timeInformation = new TimeInfoEventArgs(
dt.Hour, dt.Minute, dt.Second);
```

rồi thông báo thay đổi cho subscriber bằng cách phát đi (firing) tình huống **OnSecond Change**:

```
// nếu có ai đã đăng ký, thông báo cho họ biết
if (OnSecondChange != null)
{  OnSecondChange(this, timeInformation);
}
```

Nếu một tình huống không có ai đăng ký, thì nó sẽ được định trị bằng null. Trắc nghiệm trên kiểm tra xem trị này có null hay không, bảo đảm là có subscriber trước khi triệu gọi **OnSecondChange()**. Bạn nhớ cho là **OnSecondChange()** chấp nhận 2 đối mục: nguồn và đối tượng được dẫn xuất từ **EventArgs**. Trong đoạn mã con (snippet) bạn thấy là qui chiếu **this** của **clock** được trao qua, vì clock là nguồn của tình huống. Thông số thứ hai là đối tượng **timeInformation** kiểu **TimeInfoEventArgs** được tạo trên dòng lệnh nằm trên.

Việc gây nên tình huống sẽ triệu gọi bất cứ hàm hành sự nào được đăng ký với lớp **Clock** thông qua delegate. Ta sẽ xét đến trong chốc lát. Một khi tình huống được "nổi lên", bạn nhật tu tình trạng của lớp **Clock**.

```
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
```

Việc còn lại là tạo những lớp có thể đăng ký thụ lý tình huống. Bạn sẽ tạo ra hai lớp: lớp thứ nhất sẽ là lớp **DisplayClock**, với nhiệm vụ là không phải theo dõi thời gian, mà là hiển thị thời gian hiện hành lên màn hình. Thí dụ này đơn giản hoá chỉ còn hai hàm hành sự. Hàm thứ nhất mang tên **Subscribe()** là một hàm hỗ trợ (helper), lo đăng ký tình huống **OnSecondChange** của **Clock**. Còn hàm thứ hai là hàm thụ lý tình huống **TimeHasChanged()**:

```
public class DisplayClock
{  public void Subscribe(Clock theClock)
   {  theClock.OnSecondChange += new Clock.SecondChangedHandler(
         TimeHasChanged);
   {

   public void TimeHasChanged(object theClock, TimeInfoEventArgs ti)
   {  Console.WriteLine("Thời gian hiện hành: {0}:{1}: {2}",
         ti.hour.ToString(),
         ti.minute.ToString(),
```

```
            ti.second.ToString());
    }
}
```

Khi hàm đầu tiên **Subscribe()** được triệu gọi, nó sẽ tạo một delegate mới, **Second ChangedHandler** chuyển cho hàm thụ lý tình huống này hàm hành sự **TimeHasChanged**. Rồi nó cho đăng ký delegate với tình huống **OnSecondChange** của **Clock.**

Bạn sẽ tạo một lớp thứ hai cũng sẽ đáp ứng tình huống này, **LogCurrentTime**. Lớp này thông thường theo dõi tình huống ghi lên một tập tin, nhưng trong trường hợp của chúng ta, nó ghi theo dõi lên màn hình:

```
public class LogCurrentTime
{   public void Subscribe(Clock theClock)
    {   theClock.OnSecondChange += new Clock.SecondChangedHandler(
            WriteLogEntry);
    {

    // hàm này phải viết lên một tập tin,
    // nhưng chúng tôi cho viết lên console
    public void WriteLogEntry(object theClock, TimeInfoEventArgs ti)
    {   Console.WriteLine("Ghi theo dõi lên file: {0}:{1}: {2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}
```

Mặc dù trong thí dụ này, hai lớp **DisplayClock** và **LogCurrentTime** hầu như giống nhau, nhưng trong một môi trường sản xuất phần mềm bất cứ lớp tạp nham nào cũng có thể đăng ký vào một tình huống.

Bạn để ý các tình huống được thêm vào sử dụng tác tử +=, cho phép các tình huống mới được thêm vào **OnSecondChange** mà không phá hủy những tình huống đã được đăng ký trước đó. Khi **LogCurrentTime** đăng ký vào tình huống **OnSecondChange**, bạn không muốn tình huống đánh mất sự theo dõi sự kiện **DisplayClock** đã đăng ký rồi.

Việc còn lại là tạo một lớp **Clock**, tạo lớp **DisplayClock** và bảo nó đăng ký vào tình huống. Sau đó, bạn tạo một lớp **LogCurrentTime** và cũng bảo nó đăng ký vào tình huống. Cuối cùng, bạn bảo **Clock** chạy, như theo thí dụ 13-4 sau đây:

Thí dụ 13-4: Làm việc với events

```
namespace Prog_CSharp
{   using System;
    using System.Threading;

    // một lớp dùng trữ thông tin liên quan đên tình huống . Trong trường
    // hợp này chỉ trữ thông tin có sẵn trong lớp Clock, nhưng có thể trữ
    // thông tin tình trạng
    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 của chúng ta – đây là lớp mà các lớp khác sẽ quan sát.
    // Lớp này bố cáo một tình huống: OnSecondChange. Các lớp quan sát
    //  sẽ đăng ký tình huống này.
    public class Clock
    {   // delegate mà các subscriber phải thiết đặt
        public delegate void SecondChangeHandler(object clock,
                                TimeInfoEventArgs timeInformation);

        // tình huống mà ta sẽ bố cáo
        public event SecondChangeHander OnSecondChange;

        // cho chạy đối tượng Clock. Nó sẽ gây ra tình huống mỗi lần tăng
        // một giây
        public void Run()
        {   for(;;)
            {   // ngủ đi 10 ms
                Thread.Sleep(10);
            // đi lấy thời gian hiện hành
                System.DateTime dt = System.DateTime.Now;

                // nếu giây thay đổi báo cho subscribers
                if (dt.Second != second)
                {   // tạo đối tượng timeInfomation để chuyển cho subscriber
                    TimeInfoEventArgs timeInformation = new
                        TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);

                    // nếu có ai đã đăng ký, thông báo cho họ biết
                    if (OnSecondChange != null)
                    {   OnSecondChange(this, timeInformation);
                    }
                }
                // nhật tu trạng thái
                this.second = dt.Second;
                this.minute = dt.Minute;
```

```
            this.hour = dt.Hour;
        }   // end for
    }   // end Run
    private int hour;
    private int minute;
    private int second:
}   // end Clock

// một lớp quan sát. DisplayClock đăng ký với tình huống của Clock.
// DisplayClock lo hiển thị thời gian hiện hành
public class DisplayClock
{   // với một đối tượng Clock nào đó, cho đăng ký với
    // bộ xử lý tình huống SecondChangeHandler
    public void Subscribe(Clock theClock)
    {   theClock.OnSecondChange += new Clock.SecondChangedHandler(
            TimeHasChanged);
    }

    // hàm thiết đặt chức năng delegate
    public void TimeHasChanged(object theClock, TimeInfoEventArgs ti)
    {   Console.WriteLine("Thời gian hiện hành: {0}:{1}: {2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}   // end DisplayClock

// một subscriber thứ hai, lo viết lên tập tin
public class LogCurrentTime
{   public void Subscribe(Clock theClock)
    {   theClock.OnSecondChange += new Clock.SecondChangedHandler(
            WriteLogEntry);
    }

    // hàm này phải viết lên một tập tin, nhưng chúng tôi
    // cho viết lên console
    public void WriteLogEntry(object theClock, TimeInfoEventArgs ti)
    {   Console.WriteLine("Ghi theo dõi lên file: {0}:{1}: {2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}   // end LogCurrentTime

public class Tester
{   static void Main()
    {   // tạo một đối tượng Clock mới
        Clock theClock = new Clock();

        // tạo bộ phận hiển thị và bảo đăng ký với theClock vừa mới tạo
        DisplayClock dc = new DisplayClock();
        dc.Subscribe(theClock);

        // tạo một đối tượng LocCurrentTime và bảo đăng ký với theClock
```

```
            // vừa mới tạo
            LocCurrentTime lct = new LocCurrentTime();
            lct.Subscribe(theClock);
        }  // end Main
    }  // end Tester
}  // end Prog_CSharp
```

Kết xuất

```
Thời gian hiện hành: 14:53:56
Ghi theo dõi lên file: 14:53:56
Thời gian hiện hành: 14:53:57
Ghi theo dõi lên file: 14:53:57
    Thời gian hiện hành: 14:53:58
Ghi theo dõi lên file: 14:53:58
    Thời gian hiện hành: 14:53:59
Ghi theo dõi lên file: 14:53:59
    Thời gian hiện hành: 14:54:00
Ghi theo dõi lên file: 14:54:00
```

Tác dụng rõ ràng của đoạn mã này là tạo hai lớp **DisplayClock** và **LogCurrent Time**, cả hai đăng ký với một lớp tình huống thứ ba, **Clock.OnSecondChange**.

13.2.3 Gỡ bỏ mối liên hệ giữa Publisher và Subscriber

Lớp **Clock** có thể đơn giản in ra thời gian thay vì cho nổi lên một tình huống, do đó sao phải bận tâm sử dụng delegate một cách gián tiếp? Lợi điểm của publish/subscribe là bất cứ số lượng lớp nào cũng có thể được thông báo khi một tình huống xảy ra. Lớp subscriber không cần biết **Clock** hoạt động thế nào, và **Clock** cũng không cần biết là mình sẽ làm gì đáp ứng tình huống. Cũng tương tự như thế, một button có thể bố cáo một tình huống **OnClick**, và bất cứ đối tượng nào cũng có thể đăng ký xử lý tình huống này và sẽ nhận thông báo khi button bị ấn xuống.

Publisher và Subscriber được "tách rời" (decoupled) bởi delegate. Điều này rất cần thiết làm cho đoạn mã uyển chuyển và mạnh hơn. Lớp **Clock** có thể thay đổi cách phát hiện thay đổi thời gian mà không phá rối bất cứ lớp subscriber nào. Còn các lớp subscriber cũng có thể thay đổi làm thế nào đáp ứng thay đổi thời gian mà không phải gây rắc rối cho lớp **Clock**. Cả hai loại lớp hoạt động độc lập với nhau; do đó dễ bảo trì đoạn mã.

Chương 14

Lập trình trên môi trường .NET

Tới đây, bạn đã biết qua những gì là cơ bản về ngôn ngữ C#. Bây giờ chúng ta bước vào giai đoạn thực hành, nghĩa là sử dụng C# để viết những ứng dụng bao quát khá nhiều lĩnh vực. Tuy nhiên, trước khi thực hiện được điều này, ta cần biết làm thế nào sử dụng các công cụ những chức năng mà Visual Studio .NET cung cấp, thông qua **Integrated Development Environmet** (IDE), *môi trường tích hợp triển khai phần mềm*.

Trong chương này, chúng ta sẽ xem trong thực tế, việc lập trình trong môi trường .NET, IDE là thế nào, chúng tôi sẽ đề cập đến một vài công cụ có sẵn cho phép bạn viết và gỡ rối chương trình cũng như hướng dẫn bạn trong việc viết tốt các chương trình.

14.1 Visual Studio .NET

Visual Studio .NET là một môi trường tích hợp triển khai (Integrated Development Environmet, IDE) phần mềm, cho phép bạn viết các đoạn mã, gỡ rối và biên dịch thành một assembly một cách dễ dàng thoải mái.

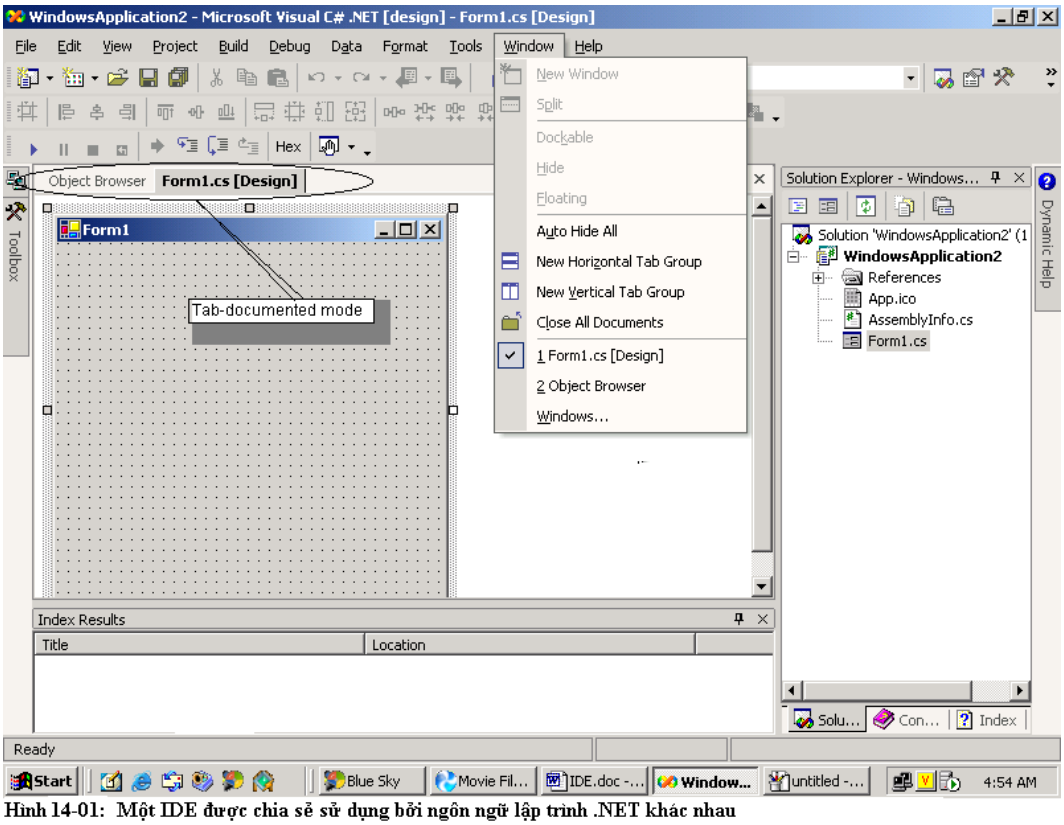
Integrated Development Environmet (IDE) có những chức năng sau đây:

- IDE chia sẻ sử dụng bởi tất cả các ngôn ngữ lập trình. Xem mục 14.1.1
- Hai chế độ giao diện (interface mode). Xem mục 14.1.2
- Hỗ trợ những cửa sổ khác nhau. Xem mục 14.1.3
- Biên dịch trong lòng IDE. Xem mục 14.1.4
- Chức năng Web Browser được cài sẵn. Xem mục 14.1.5
- Cửa sổ Command Window. Xem mục 14.1.6
- Object Browser có sẵn. Xem mục 14.1.7
- Integrated debugger. Xem mục 14.1.8
- Integrated profiler. Xem mục 14.1.9
- Integrated Help system. Xem mục 14.1.10
- Macros. Xem mục 14.1.11
- Các công cụ triển khai được nâng cấp. Xem mục 14.1.12

- Text editor Xem mục 14.1.13
- IDE và các công cụ thay đổi . Xem mục 14.1.14
- Server Explorer. Xem mục 14.1.15
- Các công cụ thao tác căn cứ dữ liệu. Xem mục 14.1.16
- Toolbox. Xem mục 14.1.17
- Task List. Xem mục 14.1.18.

14.1.1 Tất cả các ngôn ngữ đều chia sẻ sử dụng cùng một IDE

Visual Studio .NET cung cấp một IDE dành cho tất cả các ngôn ngữ .NET (Visual C#, VB .NET, Visual C++) chia sẻ sử dụng (Hình 14.1), cho phép lập trình viên sử dụng cũng cùng những công cụ giống nhau qua các ngôn ngữ lập trình khác nhau. Khi bạn mở một dự án C# hoặc VB .NET, thì IDE của Visual Studio .NET hiện lên giống như nhau.



Hình 14-01: Một IDE được chia sẻ sử dụng bởi ngôn ngữ lập trình .NET khác nhau

Bạn sẽ có sẵn một **design view editor** (bộ hiệu đính màn hình thiết kế), cho phép bạn

sử dụng kỹ thuật lôi thả đặt lên biểu mẫu (form) những ô control tạo giao diện người sử dụng (user inteface) cũng như những ô control lo việc truy xuất dữ liệu (data-access control) thuộc dự án . Khi bạn làm như thế, thì Visual Studio .NET tự động kết sinh những đoạn mã C# cần thiết đưa vào các tập tin nguồn để cho hiển lộ (instantiate) những ô control này trong dự án của bạn. Bạn nhớ cho tất cả các ô control trên .NET đều là những thể hiện của những lớp cơ bản đặc biệt.

14.1.2 Hai chế độ giao diện

IDE hỗ trợ hai chế độ giao diện (interface mode), trong việc sắp xếp các cửa sổ và khung cửa sổ (pane):

- ***Multiple document interface (MDI) mode***: Theo kiểu giao diện này, cửa sổ cha-mẹ MDI sẽ "bảo bọc" một số cửa sổ MDI con-cái trong lòng mình. Bạn có thể thấy 2 cửa sổ được mở: Start Page và Form1.cs[Design] chẳng hạn nằm chồng lên nhau như là những cửa sổ MDI con-cái.

- ***Tabbed document mode***: Đây là chế độ mặc nhiên, rất tiện lợi, vì các cửa sổ được thể hiện bởi những tấm thẻ (tab) ở trên đầu (xem hình 14.1). Bạn chỉ cần click lên tab cửa sổ nào bạn muốn cho hiển thị.

Bạn có thể chuyển bật qua lại giữa hai interface mode bằng cách sử dụng khung đối thoại **Options**. Bạn ra lệnh **Tools | Options | Environment | General**, rồi chọn một trong hai radio button (nút đài): `Tabbed Document` hoặc `MDI environment`.

14.1.3 Hỗ trợ những cửa sổ khác nhau

IDE hỗ trợ những cửa sổ cho phép bạn nhìn xem và thay đổi những khía cạnh khác nhau trên dự án của bạn. Thí dụ, bạn có sẵn những cửa sổ cho bạn thấy những lớp hình thành đoạn mã nguồn cũng như những thuộc tính trên các lớp biểu mẫu Windows hoặc biểu mẫu Web. Bạn cũng có thể sử dụng những cửa sổ này để khai báo những mục chọn biên dịch khác nhau, chẳng hạn những assembly nào (đặc biệt những assembly chứa những lớp cơ bản) đoạn mã nguồn sẽ qui chiếu về.

14.1.4 Biên dịch trong lòng IDE

Thay vì cho chạy trình biên dịch C# từ command line, bạn có thể click mục chọn `Build` trên trình đơn để cho biên dịch dự án và Visual Studio .NET sẽ triệu gọi trình biên dịch giùm bạn. Nó cũng có thể cho chạy một chương trình khả thi đã được biên dịch, như vậy bạn có thể biết chương trình chạy tốt hay không, và bạn có thể chọn giữa hai cấu

hình xây dựng chương trình khác nhau: *debug build* (phiên bản gỡ rối) hoặc *release build* (phiên bản phân phối).

14.1.5 Chức năng Web Browser có sẵn

IDE có cài sẵn một Web browser (built-in browser), do đó bạn khỏi phải cài đặt một browser khác. Tuy nhiên, người ta khuyên nên cài đặt một cái thứ hai dùng trắc nghiệm. Web browser rất hữu ích khi bạn cần nhìn xem những trang Web mà khỏi biên dịch và chạy toàn bộ dự án.

Bạn có thể xem một trang HTML thuộc thành phần dự án mở trong built-in browser, nhưng bạn cũng có thể nhìn xem một trang HTML khác, của MSDN chẳng hạn, cho dù bạn đang online hoặc không. Muốn rảo xem một URL cụ thể nào đó, bạn phải bảo đảm là Web toolbar hiện diện trên IDE. Bạn có thể cho nó hiện lên bằng cách ra lệnh **View | Toolbars | Web.** Trên ô text box, bạn có thể khỏ vào bất cứ URL (hoặc tên một tập tin) để rảo xem một Web site (hoặc mở một tập tin). Web browser được cài sẵn hoạt động giống như một browser bình thường, mà đúng như vậy.

14.1.6 Cửa sổ Command Window

Cửa sổ **Command Window** hoạt động theo hai chế độ: `Command Mode` & `Immediate Mode:`

14.1.6.1 Command mode

Bạn có thể gọi vào cửa sổ này bằng cách ấn tổ hợp phím <Ctrl+Alt+A>, hoặc ra lệnh **View | Other Windows…| Command Window**. Cửa sổ này được dùng để thi hành trực tiếp những lệnh hoặc alias, bỏ qua hệ thống trình đơn hoặc để thi hành những lệnh không có trên bất cứ trình đơn nào. Khi cửa sổ này hiện lên, bạn thấy ký hiệu (>) xuất hiện trên cửa sổ như là dấu nhắc khỏ vào một lệnh. Sau đây là các công tác có thể thực hiện trên cửa sổ Command Window:

Công tác	Giải pháp	Thí dụ
Định trị các biểu thức trên Command mode.	Cho đặt dấu hỏi (?) trước biểu thức .	`?myvar`
Chuyển bật qua Immediate mode từ Command mode.	Khỏ vào immed không có dấu (>)	`Immed`
Chuyển bật trở lại Command mode từ Immediate mode.	Khỏ vào cmd lên cửa sổ	`>cmd`

14.1.6.1.1 Parameters, Switches, và Values

Một vài lệnh của Visual Studio .NET cần có những đối mục, switches và trị. Một vài qui tắc được áp dụng khi làm việc với những lệnh loại này. Sau đây là thí dụ của một rich command để làm sáng tỏ từ ngữ.

```
Edit.ReplaceInFiles /case /pattern:regex var[1-3]+ oldpar
```

Trong thí dụ này,

- **`Edit.ReplaceInFiles`** là lệnh
- **`/case`** và **`/pattern:regex`** là những switches (đi đầu là ký tự slash [/])
- **`regex`** là trị của switch `/pattern`; switch **`/case`** không có trị
- **`var[1-3]+`** và **`oldpar`** là những thông số.

Trên dòng lệnh switches và parameters có thể tự do thay đổi vị trí với nhau, ngoại trừ lệnh **Shell**, đòi hỏi switches và parameters nằm theo một thứ tự nhất định.

14.1.6.1.2 Escape Characters

Ký tự caret (^) trên command line có nghĩa là ký tự đi liền sau được suy diễn là một ký tự chứ không phải là một ký tự điều khiển. Ta có thể dùng (^) để đặt lọt thỏm dấu nháy ("), spaces, leading slashes, carets, hoặc bất cứ ký tự trực kiện (literal) trong một thông số hoặc trị switch value, ngoại trừ tên switch. Thí dụ,

```
>Edit.Find ^^t /regex
```

14.1.6.1.3 Mark Mode

Mark mode trên cửa sổ Command window cho phép bạn tuyển, sao và dán văn bản lên cửa sổ hoặc cho thi hành lại những lệnh đi trước. Bạn chọn Mark mode bằng cách click lên hàng đi trước trện cửa sổ Command window (hoặc ở Command mode hoặc ở Immediate mode), hoặc bằng cách chọn Mark Mode từ trình đơn shorcut hoặc bằng cách ấn tổ phím <CTRL + SHIFT + M>. Một khi đã vào Mark mode, bạn có thể hoặc dùng các lệnh trên trình đơn shorcut hoặc phím tương ứng để cut, copy, paste, hoặc clear văn bản trên cửa sổ.

14.1.6.2 Immediate mode

Bạn có thể cho hiện lên cửa sổ Immediate Window, bằng cách ấn tổ hợp phím <Ctrl+Alt|I> hoặc ra lệnh **Debug | Windows | Immediate**. Khi cửa sổ hiện lên bạn thấy tựa đề: `Command Window - Immediate`.

Immediate mode của cửa sổ Command window dùng vào việc gỡ rối như định trị các biểu thức, thi hành các lệnh, in ra trị các biến, v.v.. Nó cho phép bạn khỏ vào những biểu thức cần được định trị hoặc được thi hành. Trong vài trường hợp, bạn có thể thay đổi trị của các biến.

Bạn cũng có thể tạm thời ra những lệnh Visual Studio .NET trên Immediate mode. Điều này có thể hữu ích khi bạn đang gỡ rối một ứng dụng và sử dụng Immediate mode để nhìn xem và thay đổi trị của các biến nhưng vẫn tiếp tục tương tác với IDE sử dụng các lệnh.

Muốn phát một lệnh đơn Visual Studio .NET khi đang ở Immediate mode, phải khỏ trước dấu (>).

14.1.7 Object Browser được cài sẵn

Một khía cạnh quan trọng khi lập trình trong môi trường .NET là có khả năng tìm ra những hàm hành sự nào có sẵn trên các lớp cơ bản cũng như bất cứ thư viện khác mà bạn qui chiếu từ assembly của bạn. Chức năng này có sẵn trong **Object Browser**,[51] bộ rảo xem các đối tượng.

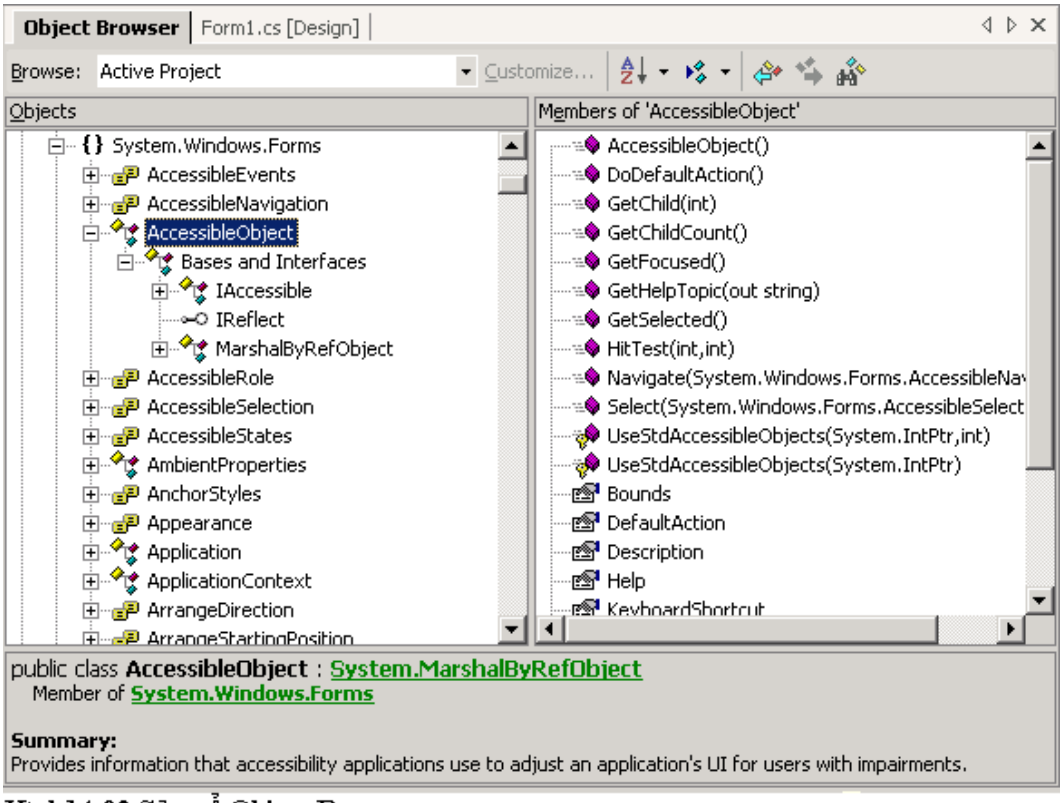
Bạn có thể cho hiện lên cửa sổ **Object Browser** (hình 14-02), bằng cách ấn tổ hợp phím <Ctrl+Alt+J>, hoặc click lên **Tab Object Browser** trên màn hình code editor, hoặc ra lệnh **View | Other Windows | Object Browser**. Cửa sổ này cho phép bạn quan sát và khám phá những đối tượng (namespace, lớp, hàm hành sự, tình huống, biến, hằng, mục tin enum, v.v..) từ những cấu kiện khác nhau; các cấu kiện này có thể là những dự án trong giải pháp hiện hành, những cấu kiện được qui chiếu trong lòng những dự án này, cũng như những cấu kiện nằm ngoài.

Cửa sổ **Object Browser** cho hiển thị một tree view (cách nhìn theo kiểu cây) cho thấy cấu trúc lớp của ứng dụng, cho phép bạn thanh sát các thành viên lớp. Các thành viên này được hiển thị trên khung cửa phía tay phải. Ngoài việc cho thấy namespace và lớp của dự án, cửa sổ này còn cho phép những namespace và lớp trên tất cả các assembly được qui chiếu bởi dự án.

Một điểm duy nhất bạn phải chú ý là trước tiên cửa sổ **Object Browser** cho gộp lớp theo thư viện mà các lớp này được trữ, sau đó mới đến namespace. Rất tiếc, vì namespace đối với lớp cơ bản trải dài trên nhiều thư viện, do đó bạn sẽ nhọc công dò tìm một lớp đặc biệt nào đó trừ khi bạn biết nó nằm ở thư viện nào.

---

[51] Chúng tôi không dám dịch là "bộ duyệt các đối tượng". Từ browse là rảo xem chứ chả có gì mà "duyệt" cả.



Hình14-02:Cửa sổ Object Browser

Bạn có thể xác định những cấu kiện nào sẽ được hiện lên vào một lúc nào đó bằng cách chọn và "cắt xén" (customizing) phạm vi rảo xem (browsing scope). Những cấu kiện trong phạm vi rảo xem bao gồm cấu kiện COM và cấu kiện .NET Framework. Hình 14.2 cho thấy **Object Browser**. Sau đây là những chi tiết về ý nghĩa của các thành phần trên cửa sổ này.

Browse

Ô liệt kê này cho phép bạn chọn một trong hai phạm vi rảo xem **Active Project** hoặc **Selected Components. Active Project** browsing scope là nội dung của dự án hiện dịch và những cấu kiện được qui chiếu của nó. Khi dự án hiện dịch thay đổi thì **Object Browser** sẽ nhật tu những thay đổi này.

Customize

Nút này chỉ hiệu lực khi bạn chọn **Selected Components** như là phạm vi rảo xem. Ấn nút này sẽ cho hiển thị khung đối thoại **Selected Components** cho phép bạn chỉ định những cấu kiện bạn muốn rảo xem — "projects and their referenced components" và "external components and libraries"

Object Browser Sort Objects

Nút **Sort Objects** cho phép bạn chọn một cách nhìn nào đó đối với các đối tượng. Bạn có thể sắp xếp các đối tượng độc lập khỏi các thành viên. Bạn có thể chọn **Sort Alphabetically** (sắp xếp theo thứ tự ABC), **Sort By Type** (sắp xếp theo kiểu dữ liệu, chẳng hạn bases, theo sau là interfaces, theo sau là hàm hành sự v.v..), **Sort By Access** (sắp xếp theo kiểu truy xuất, chẳng hạn public hoặc private), **Group ByType** (sắp xếp theo nhóm kiểu cấu kiện theo dạng cây).

Object Browser Sort Members

Nút **Sort Members** cho phép bạn chọn một cái nhìn đặc biệt các thành viên. Bạn có những lựa chọn: **Sort Alphabetically** (sắp xếp theo thứ tự ABC), **Sort By Type** (sắp xếp theo kiểu dữ liệu, chẳng hạn bases, theo sau là interfaces, theo sau là hàm hành sự v.v..), **Sort By Access** (sắp xếp theo kiểu truy xuất, chẳng hạn public hoặc private).

Back và **Forward**

Nút **Back** này cho lội ngược về những mục được chọn trước đó, còn nút **Forward** thì theo chiều tiến tới. Cả hai nút **Back** và **Forward** duy trì một danh sách lịch sử những mục mà bạn đã rảo qua trước đó.

Find Symbol

Cho hiển thị khung đối thoại **Find Symbol** cho phép bạn truy tìm trên phạm vi rảo xem hiện hành đối với một ký hiệu đặc biệt

Objects Pane

Khung cửa sổ nằm phía tay trái **Object Browser** cho hiển thị các container objects trên những cấu kiện trên phạm vi rảo xem hiện hành, chẳng hạn namespaces, classes, structures, interfaces, types, và enums.

Members Pane

Khung cửa sổ nằm phía tay phải **Object Browser** cho hiển thị các thành viên của đối tượng được chọn ở khung cửa sổ **Objects**. Đây có thể bao gồm properties, methods, events, variables, constants, và enum items.

Description Pane

Khung cửa sổ nằm cuối cửa sổ **Object Browser** cho hiển thị thông tin chi tiết liên quan đến đối tượng hoặc thành viên hiện được chọn.

14.1.8 Integrated Debugger

Integrated Debugger giờ đây hỗ trợ việc gỡ rối xuyên ngôn ngữ (cross-language debugging) trong khuôn viên IDE. Ngoài ra, bạn có thể gỡ rối trong một lúc nhiều

chương trình. Bạn có thể hiệu đính đoạn mã ngay trong text editor Visual Studio .NET để sửa chữa bug, rồi cho biên dịch lại và cho chạy lại chương trình đã được sửa chữa ngay tại chỗ bỏ dở vì lỗi.

Chương 3, "Sử dụng Debugger thế nào?", tập I bộ sách này, đã điểm qua Debugger rồi. Đề nghị bạn xem lại.

14.1.9 Integrated Profiler

Một khi chương trình của bạn đã chạy "ngon lành" và được biên dịch, Visual Studio .NET có thể "móc nối" (hook up) về một profiler[52]; bộ phận này sẽ đo lường xem chương trình của bạn mất bao nhiêu thời gian trên mỗi hàm hành sự. Nó còn cung cấp cho bạn một thông tin khác cho bạn biết một hàm hành sự nào đó có được thi hành hay không, như vậy bạn có cơ may cải tiến hiệu năng của chương trình. Nếu bạn muốn chương trình chạy nhanh hơn, dữ kiện mà profiler cung cấp cho phép bạn biết hàm hành sự nào ngốn hết thời gian của chương trình, nghĩa là bạn sẽ phải để ý đến những hàm nào phải tối ưu hoá. Visual Studio .NET cung cấp cho bạn một **Visual Studio Analyzer**, một công cụ cho phép bạn phân tích hiệu năng của từng cấu kiện trên một dự án, để từ đó tiến hành tối ưu hoá cấu kiện nào kém hiệu năng. Ở đây chúng tôi không thể đề cập trọn vẹn việc sử dụng công cụ này. Bạn có thể tham khảo trên MSDN.

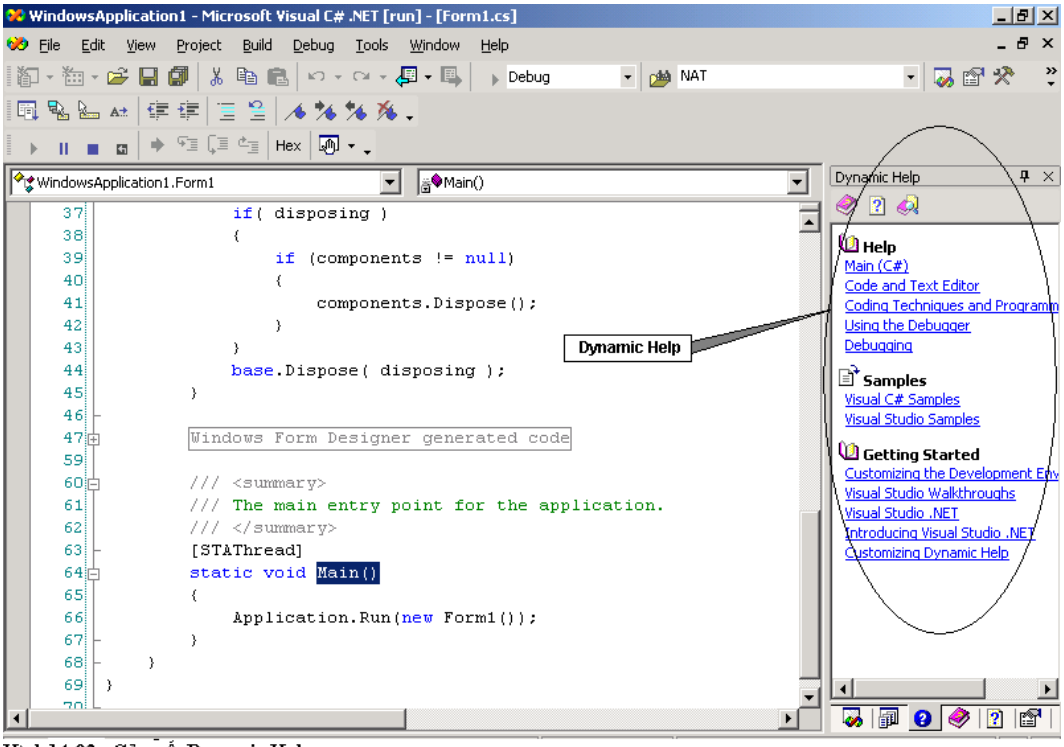
14.1.10 Integrated Help System

Chắc bạn đã biết Help là gì rồi. Thí dụ, bạn đang ở trên text editor, và bạn không chắc ý nghĩa của một từ chốt nào đó, bạn chỉ việc cho ngời sáng từ chốt này rồi ấn phím <F1>, thì Visual Studio .NET sẽ yêu cầu MSDN cho hiện lên đề mục liên quan đến từ chốt này. Cũng tương tự như thế, nếu bạn không biết chắc ý nghĩa của một vài sai lầm khi biên dịch, bạn có thể gọi Help để cho hiện lên sưu liệu về sai lầm này. Nói tóm lại, bạn nên làm quen hệ thống Help của Visual Studio .NET, vì nó khá tiến bộ, và gồm các phần sau đây:

- ***Dynamic Help***. Cửa sổ **Dynamic Help** (hiện lên khi bạn ấn nút <Ctl+F1>) thay đổi nội dung một cách linh động dựa trên item nào bạn hiện đang chọn (cửa sổ, trình đơn, từ chốt mã nguồn, v.v..). Thí dụ, nếu bạn chĩa con trỏ vào hàm **Main()** trên cửa sổ mã nguồn, thì cửa sổ Dynamic Help sẽ hiển thị những gì như bạn thấy trên hình 14.3. Trên cửa sổ này, mỗi hàng có gạch dưới là một kết nối (link) về đề mục mà bạn quan tâm. Bạn chỉ cần click vào đề mục bạn quan tâm, thì cửa sổ Help sẽ hiện lên với nội dung đề mục bạn nhắm tới. Phần văn bản hiện lên hoặc trong ứng dụng MSDN Library hoặc trong lòng IDE như là document, tùy theo

[52] Tạm thời chúng tôi chưa dịch được. Profile là dáng dấp.

việc đặt để Help như là external hoặc internal display (khi bạn ra lệnh **Tools | Options | Environment | Help** rồi chọn radio button **Internal Help/External Help**). Cửa sổ Dynamic Help chỉ hoạt động khi bạn đang ở trên cửa sổ mã nguồn, cũng như khi bạn đang hiệu đính đoạn mã HTML, hoặc sử dụng XML Designer, thiết kế báo cáo, hiệu đính CSS (Cascading Style Sheet), v.v..



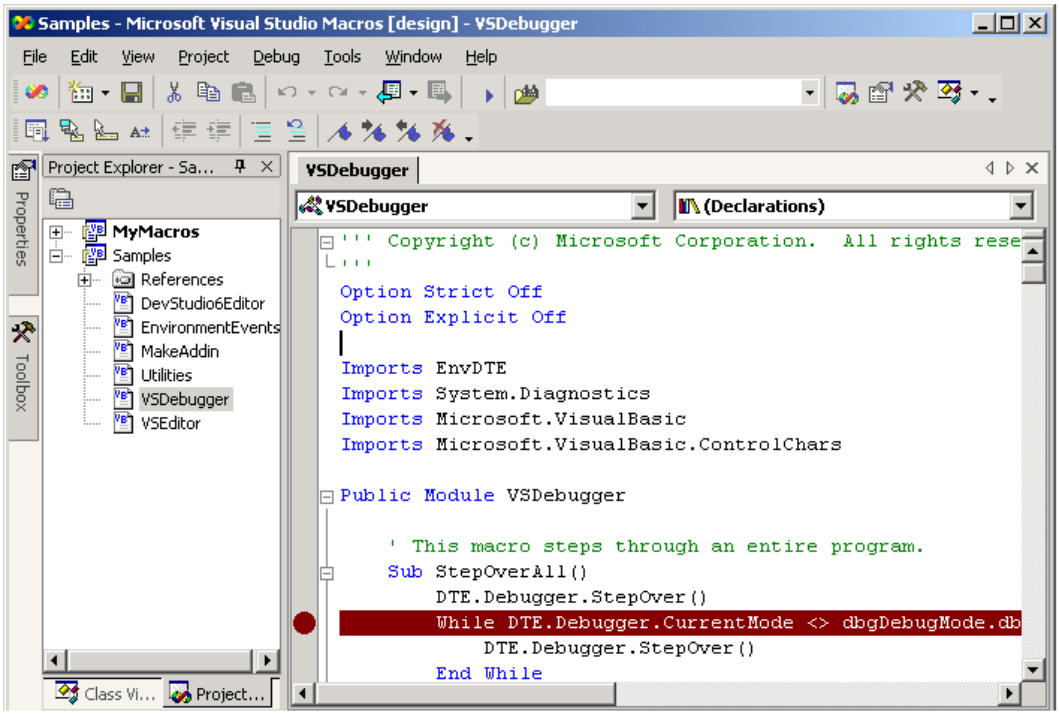
Hình 14-03: Cửa sổ Dynamic Help

- ***Microsoft Visual Studio .NETdocumentation***: Hệ thống Help trọn vẹn đối với Visual Studio .NET được bao gồm vào như là sưu liệu Platform SDK và .NET Framework. Help browser được cải tiến so với những phiên bản đi trước. Nó bây giờ là một ứng dụng kiểu browser với những link nối về thông tin thích ứng trên những Microsoft Web site khác nhau.

14.1.11 Macros

Khi triển khai ứng dụng, có thể bạn đã biết bạn phải làm một số công tác, và những công tác này mang tính lặp đi lặp lại liên tục. Visual Studio .NET cho bạn cơ hội để tự động hóa những công tác mang tính lặp đi lặp lại này, thông qua việc sử dụng những macro. Macro có thể được tạo và/hoặc hiệu đính sử dụng hoặc **Recorder** hoặc **Macros**

IDE. Hình 14.4 cho thấy Macro IDE, mà bạn có thể cho hiện lên bằng cách ấn nút <Alt+F11>, hoặc ra **lệnh Tools | Macros | Macro IDE**. Trên hình 14.4, bạn có thể thấy một trong những thí dụ macro kèm theo, macro VSDebugger, được hiển thị trên Macro IDE. Mã nguồn viết theo VB .NET và chúng tôi có đặt một chốt ngừng (nơi có dấu tròn đậm) cho thấy Macros IDE hoạt động giống Visual Studio .NET IDE.



Hình 14-04: Visual Studio .NET Macro IDE

14.1.12 Các công cụ triển khai được nâng cấp

Các công cụ triển khai ứng dụng trên Visual Studio .NET được nâng cấp rất nhiều so với những phiên bản đi trước. Trên Visual Studio .NET, bạn có thể thực hiện những công tác sau đây:

- Triển khai những ứng dụng tiers trên những test server khác nhau cho phép gỡ rồi từ xa.
- Triển khai những ứng dụng Web.
- Phân phối những ứng dụng sử dụng phần mềm Microsoft Windows Installer.

14.1.13 Trình soạn thảo văn bản

Một trình soạn thảo văn bản (text editor) cho phép bạn viết đoạn mã C# (cũng như VB .NET hoặc C++). Trình soạn thảo văn bản này khá tinh vi, rất rành cú pháp C#. Nghĩa là, khi bạn khỏ các câu lệnh vào, nó có thể tự động bố trí đoạn mã, thí dụ bằng cách thụt canh cột các dòng lệnh, cho khớp cặp dấu {}, và tô màu những từ chốt. Ngoài ra, nó còn kiểm tra một vài cú pháp khi bạn đang khỏ lệnh, và có thể gạch dưới báo động là có thể có sai lầm khi biên dịch. Thêm lại, nó có một chức năng khá đặc biệt gọi là **Intellisense**, theo đấy khi bạn bắt đầu khỏ tên lớp, vùng mục tin hoặc hàm hành sự chẳng hạn, thì nó tự động hiển thị những ô liệt kê nho nhỏ cho thấy những mục có sẵn mà bạn có thể chọn, khỏi nhọc công khỏ tiếp để hoàn tất câu lệnh.

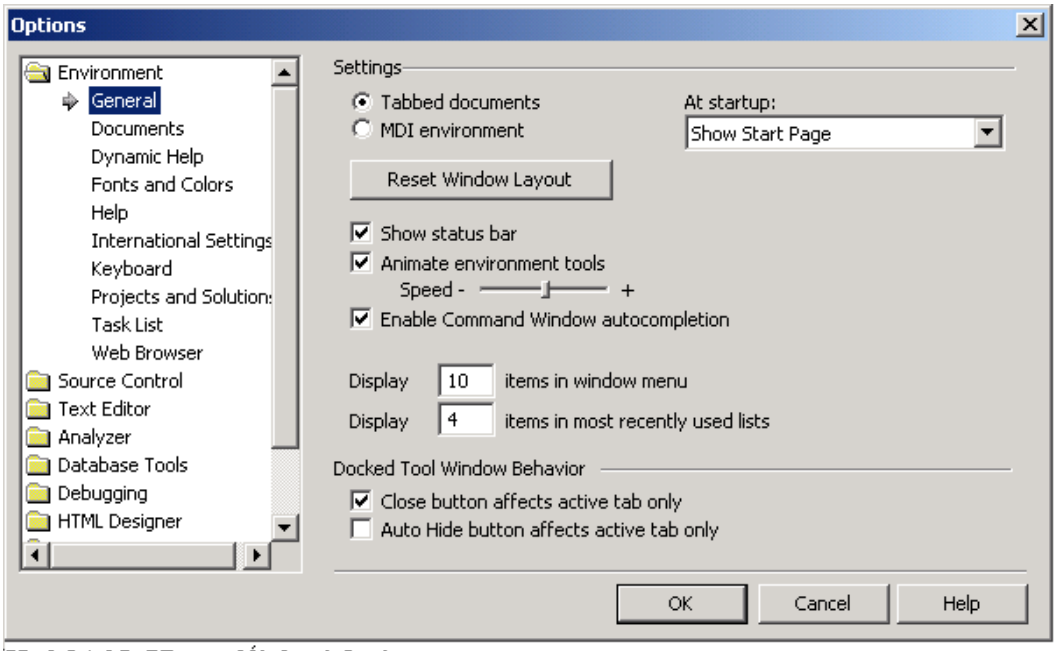
Tuy nhiên, có điểm mới trên text editor của IDE là đưa vào việc đánh số thứ tự hàng được hiển thị ở phía tay trái cửa sổ mã nguồn (xem hình 14.8). Theo mặc nhiên thì không cho hiện lên số thứ tự hàng này. Tuy nhiên, bạn có thể cho hiện lên đối với một ngôn ngữ hoặc đối với tất cả các ngôn ngữ sử dụng **Text Editor Category** trên khung đối thoại **Options** (ra lệnh **Tools | Options | Text Editor | C# | Line numbers**).

14.1.14 IDE và những công cụ thay đổi

Theo mặc nhiên, IDE và các công cụ trên IDE đã mang sẵn dáng dấp nào đó làm bạn thoải mái trong công việc. Tuy nhiên, bạn có khả năng thay đổi phần lớn những chức năng từ khung đối thoại **Option**, mà bạn có thể cho hiện lên bằng cách ra lệnh **Tools | Options…** Xem hình 14.5.

Trên khung đối thoại **Options**, bạn có thể thay đổi cách hành xử và dáng dấp mặc nhiên của IDE cũng như các công cụ liên đới. Phía tay trái, bạn thấy có một cấu trúc cây với tất cả các loại mục chọn mà bạn có thể thay đổi. Trên hình 14.5, loại **Environment** được chọn. Bạn thấy những radio button hoặc những check box hoặc drop down list box v.v.. mà bạn có thể chọn thay đổi. Thí dụ, bạn chọn sử dụng theo **Tabbed documents** (tài liệu sắp xếp theo dạng trang) hoặc theo **MDI environment (**cửa sổ sắp xếp theo trật tự cha-mẹ/con cái).

Các ngôn ngữ lập trình khác nhau cũng sẽ có những đặt để (setting) hơi khác nhau, do đó nếu bạn cần những đặt để nào, chẳng hạn kích thước canh cột (tab size) và phông chữ giống nhau cho tất cả các ngôn ngữ bạn dùng đến, thì bạn kê rõ ra ở trên khung đối thoại **Options** này.



Hình 14-05: Khung đối thoại Options

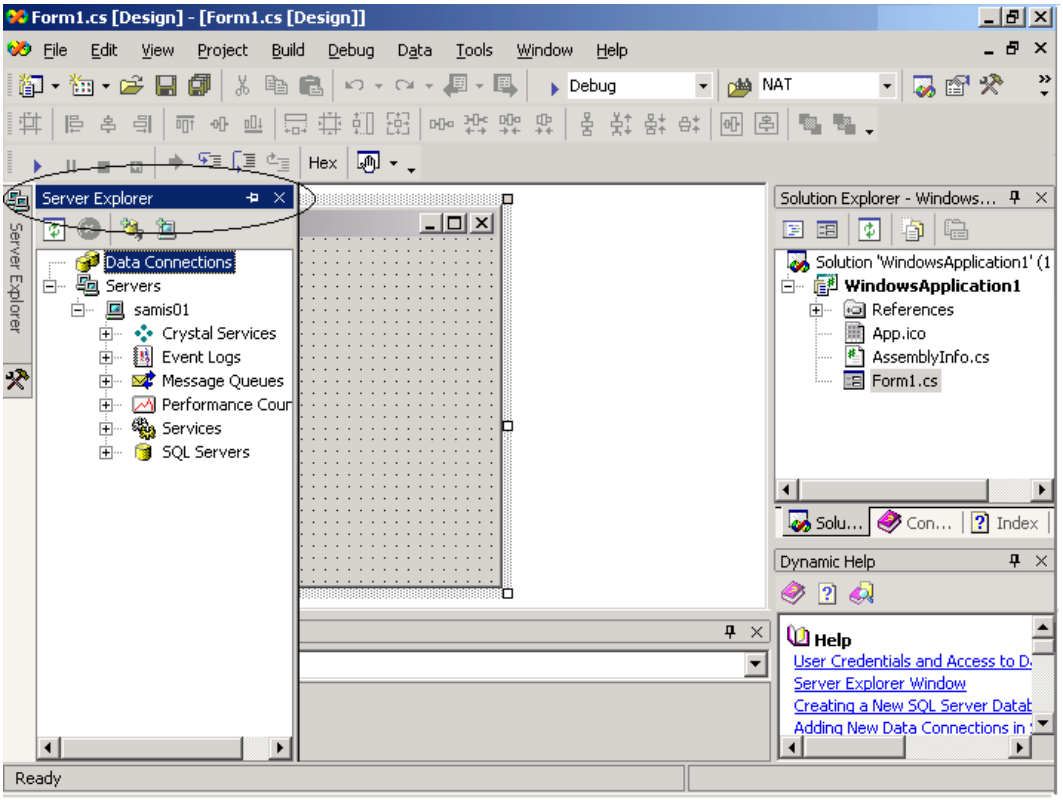
14.1.15 Server Explorer

Bạn có thể dùng cửa sổ **Server Explorer**, nằm ở thanh công cụ đứng phía tay trái theo mặc nhiên, để thao tác trên các nguồn lực[53] (resource) trên bất cứ server nào mà bạn có thể thâm nhập. Những nguồn lực mà bạn có thể thao tác bao gồm:

- Các đối tượng căn cứ dữ liệu (database object)
- Cái đếm hiệu năng (Performance counter)*
- Hàng nối đuôi các thông điệp (message queue)*
- Các dịch vụ (Services)*
- Bảng theo dõi các tình huống (event logs)*

Cửa sổ này được xem như là trung tâm điều khiển của một ứng dụng được phân phối mà bạn đang xây dựng. Sử dụng **Server Explorer**, bạn có khả năng kết nối về căn cứ dữ liệu cục bộ hoặc nằm từ xa và thao tác lên căn cứ dữ liệu này (và nhìn xem bất cứ đối tượng trên căn cứ dữ liệu), chui vào một hàng nối đuôi các thông điệp cũng như có được thông tin liên quan đến máy (các dịch vụ và bảng theo dõi tình huống).

[53] Chúng tôi không dịch là "tài nguyên" .



Hình 14-06: Cửa sổ Server Explorer

Phần lớn những nguồn lực này đều mới, đánh dấu bởi dấu hoa thị (*), và bạn có thể lôi các nguồn lực này lên biểu mẫu Web hoặc Windows rồi thao tác lên chúng trong lòng mã nguồn của bạn. Xem hình 14.6.Việc truy xuất các nguồn lực từ Server Explorer tùy thuộc vào phiên bản Visual Studio .NET nào bạn tậu được.

Cửa sổ **Server Explorer** thường kết nối với cửa sổ **Properties**, do đó khi bạn mở mắt gút (node) **Services** và click một service nào đó, thì những thuộc tính của dịch vụ này sẽ hiện lên trên cửa sổ **Properties** này. Bạn xem cửa sổ **Properties** ở mục 14.3.2.2 vào cuối chương này. Bạn sẽ làm quen nhiều với Server Explorer, khi nói đến ADO.NET trong tập IV bộ sách này.

14.1.16 Các công cụ thao tác căn cứ dữ liệu

Hỗ trợ Integated Database cũng là thành phần của Visual Studio IDE. Thông qua cửa sổ **Server Explorer**, bạn có thể đặt để những kết nối (connection) với một căn cứ dữ liệu

và thao tác lên các đối tượng căn cứ dữ liệu theo nhiều cách khác nhau. Bạn có thể tìm thấy chi tiết trong phần nói về ADO .NET, tập IV của bộ sách này.

14.1.17 Hộp đồ nghề

Hộp đồ nghề (toolbox), nằm ở thanh công cụ đứng phía tay trái của IDE, theo mặc nhiên, chứa tất cả các công cụ bẩm sinh mà bạn có thể dùng trong khi thiết kế và hiệu đính, chẳng hạn **HTML design, Windows Form design, Web Forms design, Code Editor** và **UML Diagram Design.** Khi bạn mở một cửa sổ thiết kế nào đó, thì nội dung của Toolbox cũng thay đổi theo thích ứng với loại thiết kế hiện bạn đang làm việc (Xem hình 14.7). Muốn cho Toolbox hiện lên bạn ra lệnh **View | Toolbox.**

Nguyên tắc toolbox được áp dụng trên tất cả các môi trường triển khai trên Visual Studio 6, nhưng với .NET số cấu kiện (component) có sẵn trên Toolbox tăng một cách đáng kể. Loại cấu kiện có sẵn cho bạn dùng tùy thuộc trong chừng mực nào đó vào loại dự án bạn đang thiết kế. Ta có thể kể những loại cấu kiện được phân loại như sau:

- **Data Acces Components:** Đây là những lớp cho phép bạn kết nối về các nguồn dữ liệu (data source).

- **Windows Forms Components**: Đây là những lớp tượng trưng cho những ô control nhìn thấy được, chẳng hạn text box, list box hoặc tree view, v.v..

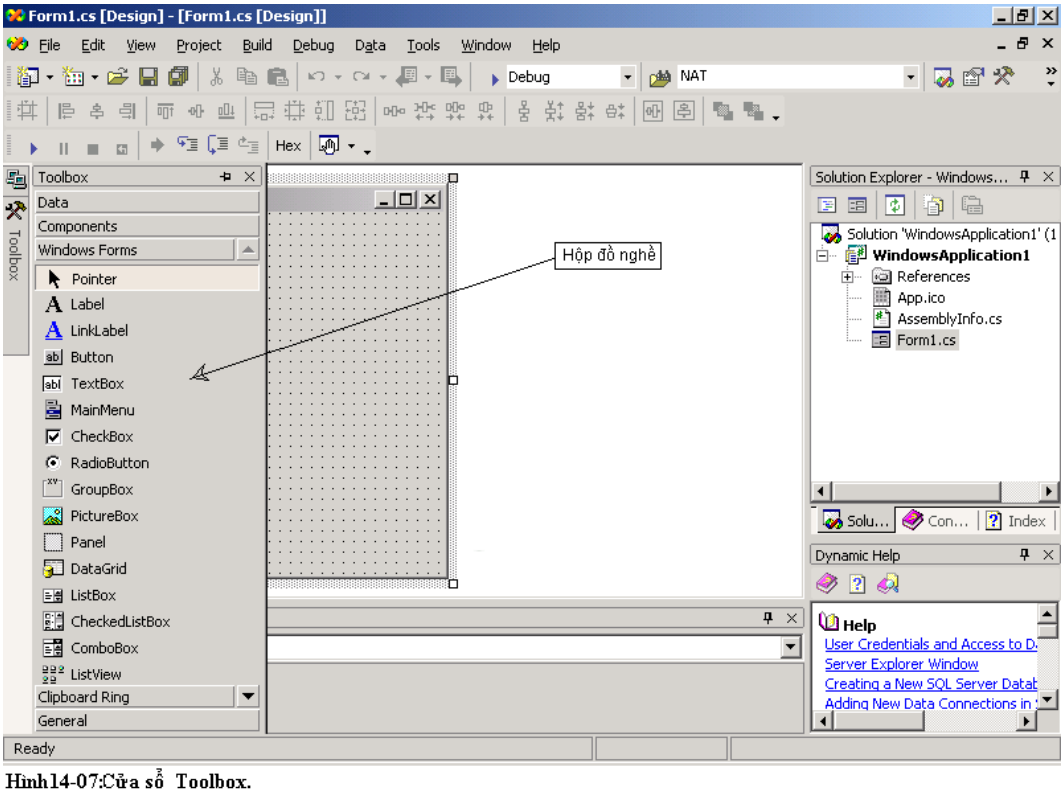
- **Web Forms Components:** Đây là những lớp cơ bản thực hiện những gì các ô control trên windows đã thực hiện nhưng ở đây sẽ làm việc trong môi trường Web Browser, bằng cách chuyển đi những kết xuất HTML để mô phỏng những ô control trên browser.

- **Components.** Đây là những lớp .NET "lục lăng lục chốt" thi hành những công tác hữu ích khác nhau trên máy của bạn, chẳng hạn kết nối về dịch vụ thư mục hoặc theo dõi các tình huống (event log).

14.1.18 Cửa sổ Task List

Cửa sổ **Task List** (hình 14.8), đậu ở cuối IDE theo mặc nhiên, giúp bạn tổ chức và quản lý những công tác khác nhau mà bạn cần hoàn tất để có thể xây dựng giải pháp của bạn. Nếu cửa sổ này không hiện lên, bạn có thể ấn tổ hợp phím <Ctrl+Alt+K>, hoặc ra lệnh **View | Other Windows | Task List.** Nhà triển khai phần mềm và CLR cả hai sử dụng cửa sổ **Task List** này. Nghĩa là, lập trình viên có thể thêm vào bằng tay những công tác, nhưng CLR cũng thêm vào một công tác nếu có sai lầm xảy ra khi biên dịch. Khi một công tác được thêm vào danh sách, nó chứa một tên tập tin và số thứ tự hàng lệnh. Như

vậy, bạn chỉ cần double-click tên công tác và đi thẳng vào vấn đề hoặc công tác chưa hoàn tất.



Hình14-07:Cửa sổ Toolbox.

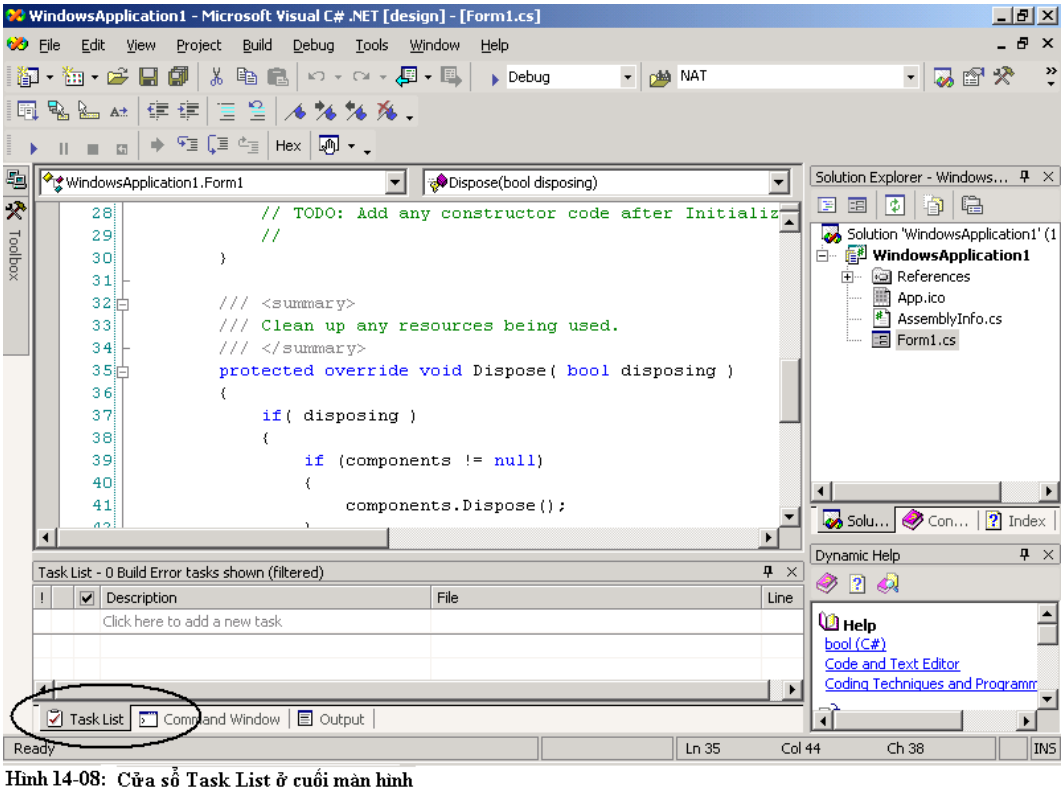
Một khi bạn hoàn tất một công tác, bạn có thể chọn ô check box thích ứng để cho biết công tác nào đã xong.

14.2 Trình đơn và các thanh công cụ

Các trình đơn (menu) cho phép bạn truy cập nhiều lệnh (command) cũng như những khả năng của Visual Studio .NET. Những lệnh nào thường xuyên được dùng đến sẽ được sao đôi lên các nút trên các thanh công cụ (toolbar) để thao tác cho nhanh hơn.

Các trình đơn cũng như thanh công cụ đều thuộc loại cảnh ứng (context-sensitive), nghĩa là sự lựa chọn dành sẵn tùy thuộc vào việc phần nào IDE hiện được chọn ra và hoạt động nào được chờ đợi hoặc được phép. Thí dụ, nếu cửa sổ hiện dịch hiện hành là một cửa sổ hiệu đính đoạn mã, thì trình đơn chính chỉ gồm: **File, Edit, View, Project, Build,**

Debug, Tools, Window, và **Help**. Còn nếu bạn đang ở một ứng dụng Web và đang ở chế độ design, thì ngoài các mục chọn trình đơn kể trên, bạn còn có những mục chọn trình đơn: **Data, Format, Table, Insert,** và **Frames**.



Hình 14-08: Cửa sổ Task List ở cuối màn hình

Phần sau đây mô tả các mục chọn trình đơn (menu item) và trình đơn con (submenu), tập trung vào những khía cạnh quan trọng và khác nhau so với các lệnh Windows thông thường.

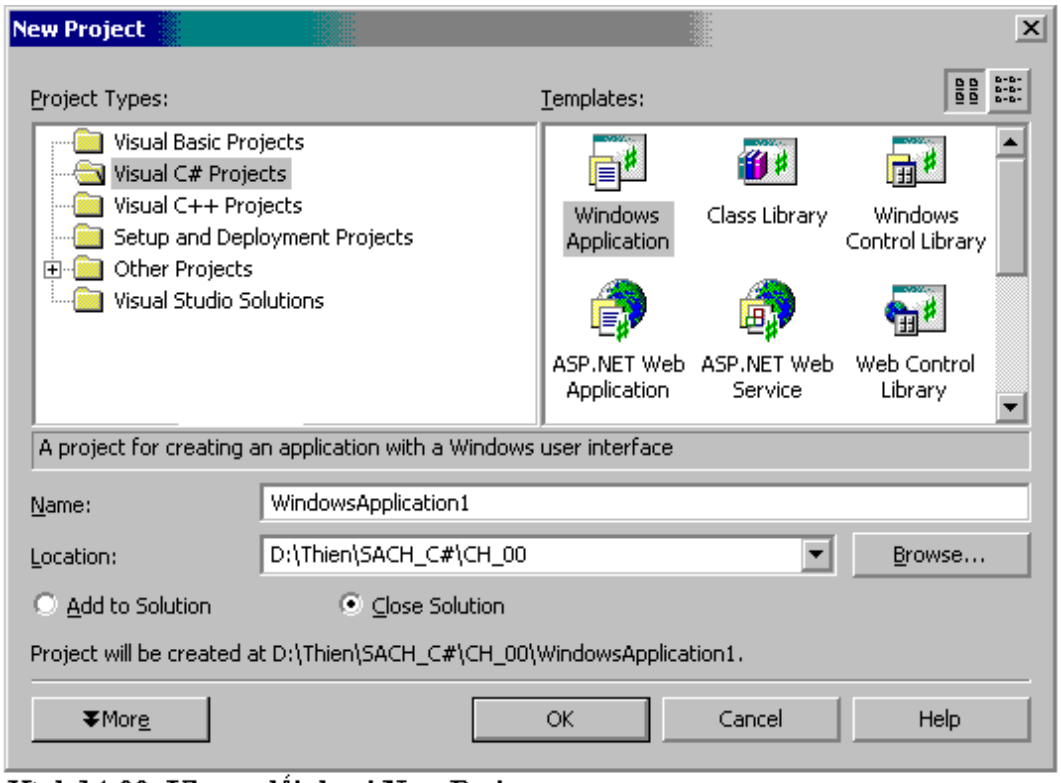
14.2.1 File Menu

File menu cho phép bạn truy cập một số tập tin, dự án và những lệnh có liên hệ với giải pháp (solution-related command). Phần lớn các lệnh này là nhạy cảm đối với nội dung (content sensitive). Sau đây là những mô tả đối với những lệnh không tự giải thích được:

14.2.1.1 New

Giống như phần lớn các ứng dụng Windows, mục chọn trình đơn **New** sẽ tạo những items mới dùng hoạt động với ứng dụng. Trên Visual Studio .NET, New có 3 submenu item, cho phép thụ lý những khả năng khác nhau. Đó là:

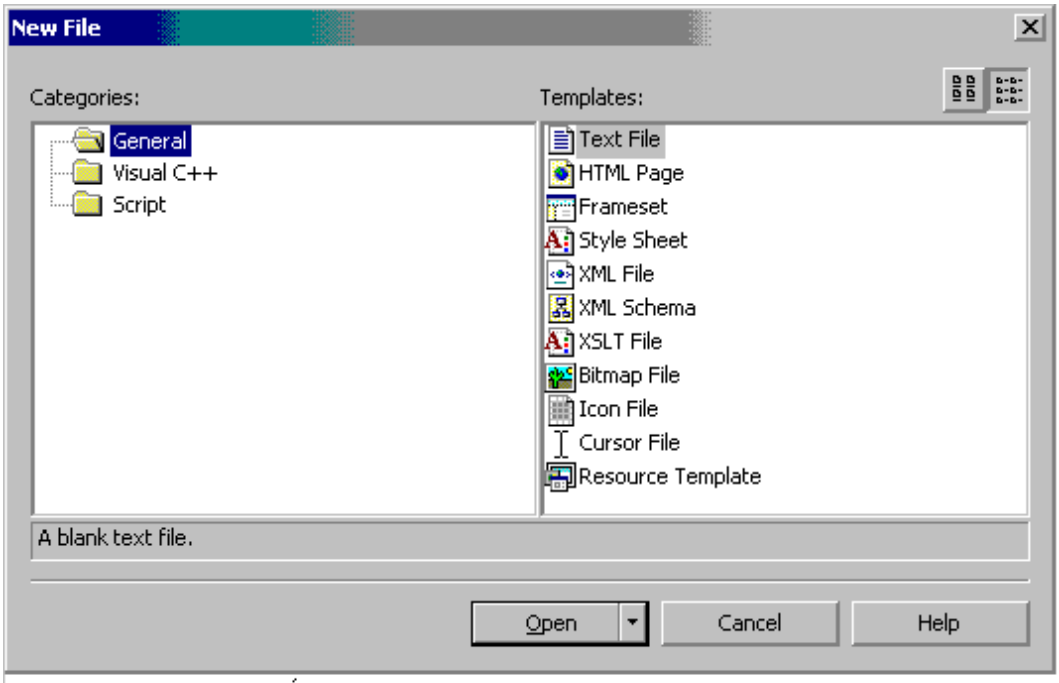
- ***Project…(Ctrl+Shift+N)***: Lệnh **Project** cho hiện lên khung đối thoại **New Project**, thuộc loại cảnh ứng. Nếu không có dự án nào hiện được mở, bạn sẽ thấy khung đối thoại như theo hình 14-9, nhưng không có hai nút đài **Add to Solution** và **Close Solution**. Còn nếu đã có một dự án hiện được mở, thì sự hiện diện hai nút đài kể trên cho phép bạn chọn hoặc thêm một dự án mới vào solution (cho On nút đài **Add to Solution**) hoặc cho đóng lại dự án hiện được mở (cho On nút đài **Close Solution**) và tạo một dự án mới. Bạn để ý có hai nút hình ảnh trên góc top-right khung đối thoại cho phép bạn hiển thị template theo icon lớn hoặc nhỏ.



Hình 14-09: Khung đối thoại New Project

- ***File***.. (Ctrl+N): Lệnh **File** sẽ cho hiện lên khung đối thoại **New File** như theo hình 14-10. Nó cung cấp 3 category tập tin khác nhau và nhiều kiểu tập tin (template) trong mỗi category. Các tập tin được tạo theo cách này sẽ được trữ theo mặc nhiên trên cùng thư mục dự án (mặc dù bạn có thể cho trữ về một thư mục khác). Các tập tin này sẽ được hiển thị trên cửa sổ Solution Explorer nếu nút Show All Files bị ấn xuống, nhưng hiện chúng chưa thuộc thành phần của

solution trừ khi bạn thêm vào một cách tường minh (explicit), bằng cách sử dụng một trong những Add menu item sẽ được mô tả về sau trong chương này. Nói cách khác, đây là những tập tin linh tinh mà chúng tôi sẽ mô tả về sau.



Hình 14-10: Khung đối thoại New File

- ***Blank Solution…*** Lệnh **Blank Solution** cũng cho hiện lên một khung đối thoại **New Project** tương tự như hình 14.09, với nút radio **Add To Solution** về Off, và Project Type mặc nhiên được cho về **Visual Studio Solutions**, và Template cho về **Blank Solution**. Khi một giải pháp trống rỗng (blank solution) được tạo ra, nó không chứa item nào cả. Các items sẽ được thêm về sau, sử dụng **Add Menu Items**, sẽ được mô tả về sau.

Lệnh **New** có một nút tương đương trên thanh công cụ **Standard**, cho thấy các lệnh **New Project** và **Blank Solution**.

14.2.1.2 Open

Open menu item được dùng cho mở những item hiện hữu trước đó, và gồm 4 submenu item:

- ***Project…(Ctrl+Shift+O)*:** Mục chọn này cho mở một dự án hiện hữu trước đây. Solution hiện được mở sẽ bị đóng lại trước khi dự án mới được mở.

- ***Project From Web…***: Lệnh này cho hiện lên một khung đối thoại **Open Project From Web**, chấp nhận một URL chỉ về dự án cần phải mở. Solution hiện được mở sẽ bị đóng lại trước khi dự án mới được mở.

- ***File…(Ctrl+O)***: Lệnh này cho hiện lên một khung đối thoại chuẩn **Open File**, cho phép bạn rảo tìm mở bất cứ tập tin có thể truy cập trên máy hoặc trên mạng của bạn. Các tập tin được mở sẽ hiện lên cửa sổ và có thể hiệu đính được trên Visual Studio .NET, *nhưng không phải là thành phần của dự án.* Muốn cho tập tin thuộc thành phần dự án, bạn phải sử dụng một trong những lệnh **Add** menu sẽ được mô tả về sau. Lệnh **Open File** có một nút tương đương trên thanh công cụ **Standard**.

- ***File From Web…***: Lệnh này cho hiện lên một khung đối thoại **Open File From Web**, chấp nhận một URL chỉ về tập tin cần phải mở. Giống như với **Open | File** Các tập tin được mở sẽ hiện lên cửa sổ và có thể hiệu đính được trên Visual Studio .NET, *nhưng không phải là thành phần của dự án.*

14.2.1.3 Add New Item…(Ctrl+Shift+A)

Lệnh **Add New Item** này, chỉ hiện lên trên dự án Web, cho phép bạn thêm một item mới vào dự án hiện hành, và cho hiện lên một khung đối thoại **Add New Item** (hình 14-11).

Bằng cách cho bung các mắt gút trên khung cửa **Categories** phía tay trái sẽ làm giới hạn danh sách các **Templates** trên khung cửa phía tay phải. Đây là menu item dùng thêm những tập tin mới vào dự án, bao gồm các tập tin đoạn mã nguồn mới. Đối với mã nguồn, điển hình bạn thường thêm một tập tin Class mới, tự động mang phần đuôi đặc hữu của ngôn ngữ lập trình.

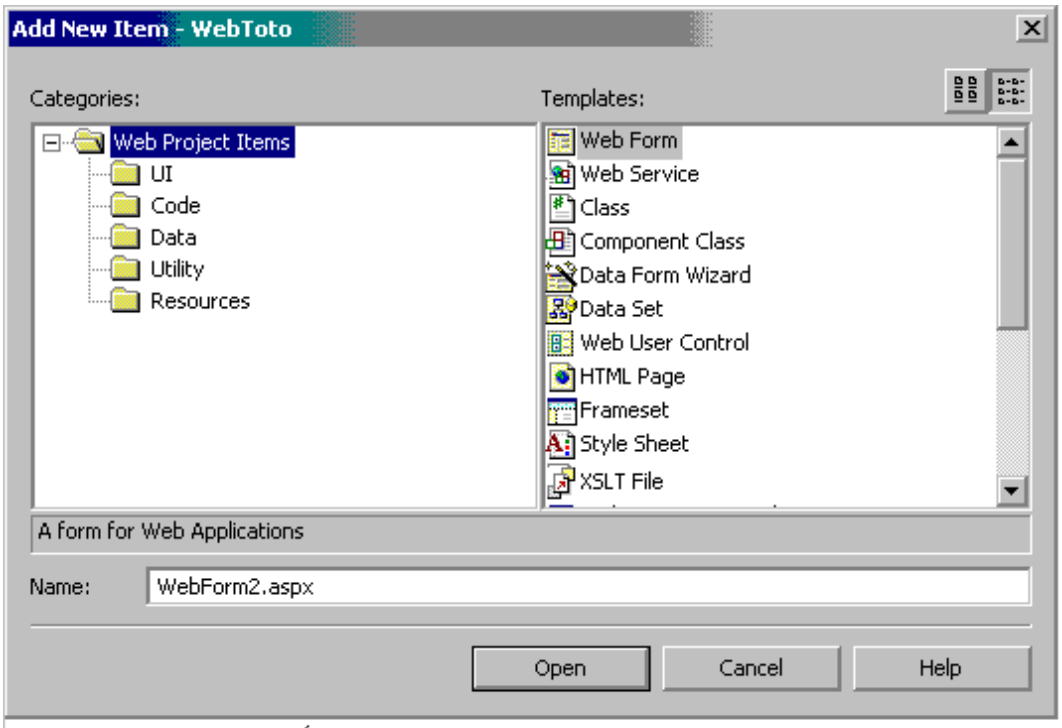
Trên thanh công cụ Standard, lệnh **Add New Item** cũng có một nút tương đương.

Trên Solution Explorer, bạn cũng có thể gọi lệnh này, bằng cách right click lên tên dự án để cho trình đơn shortcut hiện lên, rồi ra lệnh **Add | Add New Item…**thì khung đối thoại 14-11 hiện lên.

14.2.1.4 Add Existing Item…(Shift+Alt+A)

Lệnh **Add Existing Item** cũng tương tự như **Add New Item** đi trước, ngoại trừ việc thêm item hiện hữu vào dự án hiện hành. Một khung đối thoại kiểu Open File sẽ hiện lên để bạn chọn tập tin. Nếu item được thêm vào nằm ngoài thư mục dự án, thì một bản sao sẽ được thực hiện và được đưa vào thư mục dự án. Lệnh này cũng có thể truy cập từ **Solution Explorer** như với **Add New Item**. Trên **Solution Explorer**, bạn right click lên

tên dự án để cho trình đơn shortcut hiện lên, rồi ra lệnh **Add | Add Existing Item…**thì khung đối thoại 14-11 hiện lên.



Hình 14-11: Khung đối thoại Add New Item

14.2.1.5 Add Project

Add Project có 3 submenu: hai mục đầu **New Project** và **Existing Project** cho phép bạn thêm hoặc một dự án mới hoặc một dự án hiện hữu. Mục thứ ba **Existing Project From Web** sẽ cho hiện lên một khung đối thoại chấp nhận URL của dự án Web hiện hữu cần được thêm vào.

14.2.1.6 Open Solution

Khi bạn click lên **Open Solution** thì khung đối thoại **Open Solution** hiện lên cho phép bạn rảo tìm mở một solution. Solution hiện được mở sẽ bị đóng lại trước khi đưa solution mới vào.

14.2.1.7 Close Solution

Menu item này chỉ có sẵn khi một solution hiện được mở. Nếu menu item này bị click thì solution hiện được mở sẽ bị đóng lại.

14.2.1.8 Advanced Save Options…

Advanced Save Options là một submenu cảnh ứng chỉ hiện lên khi bạn đang ở chế độ hiệu đính đoạn mã. Nó sẽ cho hiện lên khung đối thoại **Advanced Save Options** cho phép bạn đặt để việc mã hóa các ký tự cũng như ký tự kết thúc các hàng (line ending) đối với tập tin. Nếu bạn sử dụng tiếng Việt trong đoạn mã thì khi khung đối thoại này hiện lên bạn chọn "Unicode(UTF-8 with signature) Code Page 65001".

14.2.1.9 Source Control

Source Control cho phép bạn tương tác với chương trình Visual Source Safe 6.0. Từ ngữ *source control* mô tả một hệ thống theo đấy một phần mềm trên server sẽ cất trữ các phiên bản liên quan đến các tập tin và theo dõi kiểm soát việc truy cập vào các tập tin kể trên.

14.2.2 Edit Menu

Trình đơn **Edit** gồm các lệnh hiệu đính văn bản và truy tìm mà bạn có thể chờ đợi, nhưng cũng cho bao gồm thêm các lệnh hữu ích trong khi hiệu đính đoạn mã. Các lệnh được sử dụng nhiều nhất là:

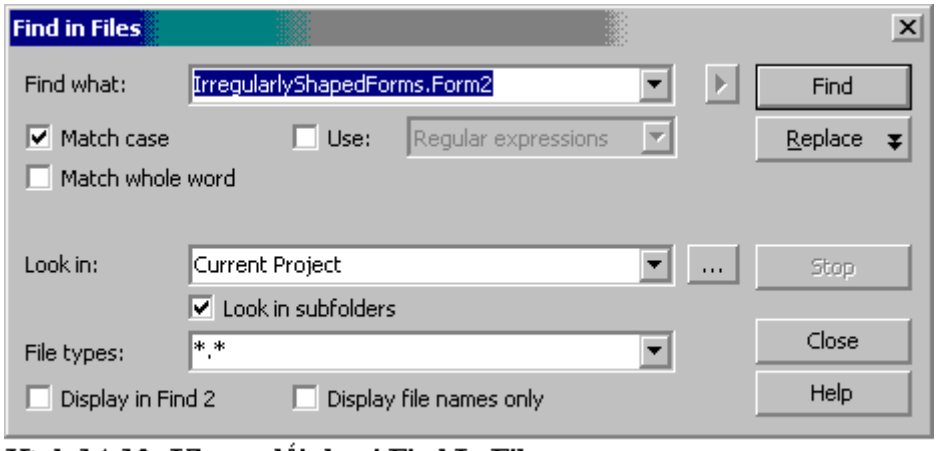
14.2.2.1 Cycle Clipboard Ring (Ctrl+Shift+V)

Clipboard Ring cũng giống như copy-and-past trên steroid. Sao chép một số chọn lựa khác nhau lên Windows clipboard, bằng cách sử dụng lệnh **Edit | Cut** (Ctrl+X) hoặc **Edit | Copy** (Ctrl +C). Rồi sau đó sử dụng **Ctrl+Shift+V** cho xoay vòng qua tất cả các chọn lựa cho phép bạn dán đúng chọn lựa khi đến phiên nó. Bạn cũng có thể thấy toàn bộ clipboard ring trên Toolbox (là một trong những khung cửa hiện lên khi bạn hiệu đính một tập tin văn bản).

Submenu này là cảnh ứng và chỉ hiện hình khi hiệu đính cửa sổ đoạn mã.

14.2.2.2 Find & Replace/Find in Files (Ctrl+Shift+F)

Find in Files là một tiện ích truy tìm cực mạnh lo tìm ra những chuỗi văn bản bất cứ nơi nào trên một thư mục hoặc thư mục con (subfolder). Nó cho hiện lên một khung đối thoại như theo hình 14-12.



Hình 14-12: Khung đối thoại Find In Files

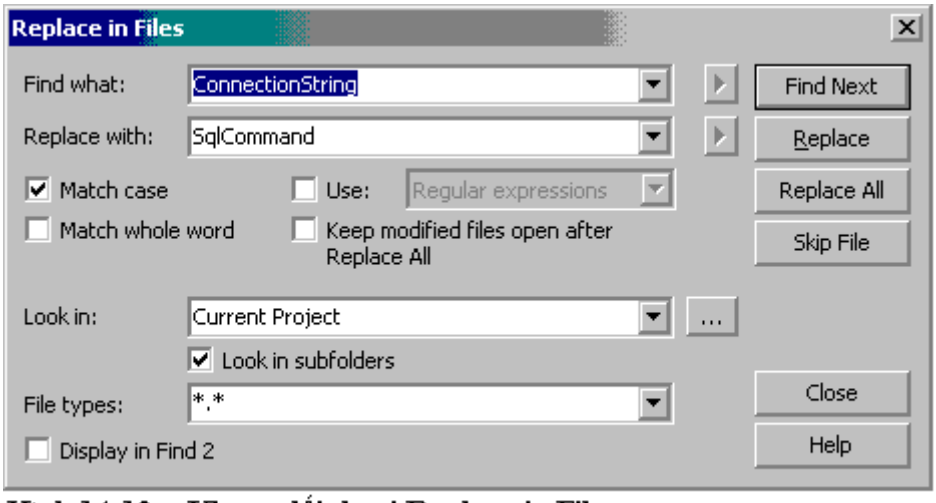
Các ô checkbox cho trình bày một số chọn lựa khá rõ ràng khỏi phải giải thích, kể cả khả năng truy tìm sử dụng hoặc wildcards hoặc regular expressions.

Nếu bạn click nút <Replace> trên khung đối thoại **Find in Files**, bạn sẽ nhận khung đối thoại **Replace in Files** như theo hình 14-13, được mô tả kế tiếp.

14.2.2.3 Find & Replace/Replace in Files (Ctrl+Shift+H)

Khung đối thoại **Replace in Files** cũng tương tự như **Find in Files**, được mô tả ở trên ngoại trừ nó cho phép bạn thay thế chuỗi văn bản mục tiêu bởi chuỗi văn bản thay thế.

Lệnh này rất hữu ích trong việc đổi tên các biểu mẫu, các lớp, các namespace, các dự án, v.v.. Việc đổi tên các đối tượng là chuyện xảy ra như cơm bửa trong lập trình, vì thường xuyên bạn không ưa những tên mặc nhiên cung cấp bởi Visual Studio .NET.



Hình 14-13: Khung đối thoại Replace in Files

Việc đổi tên không khó, nhưng đôi khi cũng có thể gặp phải khó khăn. Tên các đối tượng nằm rải rác trong dự án, thường xuyên nằm ẩn trong những chỗ tối tăm chẳng hạn trong các tập tin solution, tập tin dự án và trong suốt các tập tin mã nguồn. Mặc dù các tập tin này thuộc loại văn bản nên có thể truy tìm và thay thế, nhưng việc làm này chán ngắt và rất dễ sai lầm. Do đó lệnh **Replace in Files** làm cho công việc dễ dàng hơn và khá an toàn.

14.2.2.4 Find & Replace/Find Symbol (Alt+F12)

Lệnh **Find Symbol** sẽ cho hiện lên khung đối thoại **Find Symbol** như theo hình 14-14, cho phép bạn truy tìm các ký hiệu (chẳng hạn namespace, lớp và interfaces) cũng như các thành viên (chẳng hạn thuộc tính, hàm hành sự, tình huống, và biến). Kết quả truy tìm sẽ được hiển thị trên một cửa sổ mang tên **Find Symbol Results**. Từ đây, bạn có thể nhảy về mỗi vị trí của symbol trong đoạn mã bằng cách double click lên mỗi hàng kết quả.

14.2.2.5 Go To…

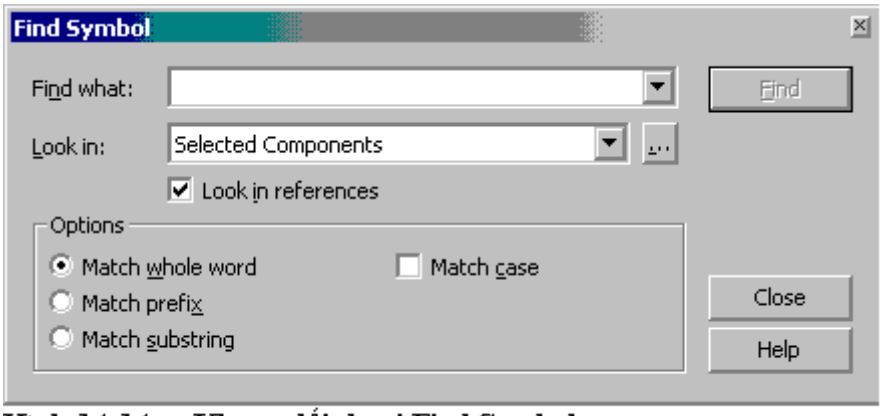
Lệnh này cho hiện lên khung đối thoại **Go To Line**, cho phép bạn khỏ vào một con số để nhảy ngay về hàng này. Lệnh này thuộc loại cảnh ứng và chỉ hiện hình khi bạn đang ở cửa sổ hiệu đính đoạn mã.

14.2.2.6 Insert File As Text

Lệnh này cho phép bạn chèn nội dung của bất cứ tập tin nào vào đoạn mã nguồn của bạn, xem như là bạn khỏ vào. Lệnh này thuộc loại cảnh ứng và chỉ hiện hình khi bạn

đang ở cửa sổ hiệu đính đoạn mã.

Khi bạn click lệnh này, thì một khung đối thoại **Insert File** (một loại Open File chuẩn) hiện lên giúp bạn rảo mở một tập tin bạn nhắm tới. Phần đuôi mặc nhiên của tập tin sẽ tương ứng với ngôn ngữ lập trình của dự án, nhưng bạn có thể truy tìm tập tin dựa trên bất cứ extension nào.



Hình 14-14: Khung đối thoại Find Symbol

14.2.2.7 Advanced

Lệnh **Advanced** này thuộc loại cảnh ứng và chỉ hiện hình khi bạn đang ở cửa sổ hiệu đính đoạn mã. Lệnh này có nhiều submenu items, bao gồm những lệnh:

- Tạo hoặc gỡ bỏ những tab trong một lựa chọn (chuyển đổi tab thành space hoặc ngược lại): **Tabify Selection** & **UnTabify Selection.**
- Ép đoạn văn bản trở thành văn bản viết theo chữ hoa hoặc chữ thường: **Make Uppercase & Make Lowercase.**
- Xoá sạch horizontal white space.
- Cho hiện lên white space (tab và ký tự space): **View White Space**.
- Chuyển bật qua lai word wrap.
- Comment & Uncomment một đoạn văn bản: **Comment Selection & Uncom ment Selection.**
- Tăng hoặc giảm độ canh thụt lùi: **Increase Line Indent & Decrease Line Indent**
- Truy tìm tăng lần: **Incremental Search** (xem mục kế tiếp)

14.2.2.8 Advanced | Incremental search (Ctrl+I)

Việc incremental search cho phép bạn truy tìm một cửa sổ hiệu đính bằng cách khỏ vào chuỗi truy tìm từng ký tự một. Khi mỗi ký tự được khỏ vào, con nháy sẽ di chuyển về chỗ xuất hiện đầu tiên của văn bản khớp với gì bạn khỏ vào.

Muốn sử dụng incremental search trên một cửa sổ, bạn ra lệnh **Edit | Advanced | Incremental Search** (hoặc ấn <Ctrl + I> hoặc <Ctrl + Shift + I>), thì lúc này con nháy biến thành một ống nhòm (binocular) với một mũi tên cho biết chiều truy tìm bắt đầu (mũi tên chĩa xuống khi bạn ấn phím <Ctrl+I> hoặc chĩa lên khi bạn ấn <Ctrl+Shift+I>). Bạn bắt đầu khỏ chuỗi văn bản cần truy tìm. Mỗi lần bạn khỏ một ký tự thì con nháy nhảy về từ nào có chuỗi khớp với những gì bạn khỏ vào và cho ngời sáng.

Tính case sensitivity của incremental search sẽ tùy thuộc vào các lệnh **Find, Replace, Find in Files,** hoặc **Replace in File** mà bạn dùng truy tìm trước đó.

Việc truy tìm bắt đầu từ trên xuống và từ trái qua phải từ vị trí hiện hành. Muốn truy tìm ngược lại thí ấn phím <Ctrl+Shift+I>.

Khi truy tìm kiểu incremental, bạn có thể dùng các tổ hợp phím sau đây:

Esc	Cho ngưng truy tìm
Backspace	Gỡ bỏ một ký tự khỏi chuỗi truy tìm
Ctrl+Shift+I	Cho đổi hướng truy tìm
Ctrl+I	Di chuyển về xảy ra kế tiếp trong tập tin đối với chuỗi văn bản truy tìm hiện hành

14.2.2.9 Bookmarks

Bookmark và Shortcut rất hữu ích để đánh dấu những điểm bạn muốn nhảy về trong đoạn mã của bạn. Bạn có thể cho đặt để những bookmark và shortcut rồi dùng shortcut key để rảo qua các điểm nhảy về này. Bookmark là tạm thời, chỉ giữ lại khi cửa sổ hiệu đính bị đóng rồi lại được mở lại, nhưng không bị giữ lại sau khi thoát khỏi Visual Studio .NET (nghĩa là khi bạn đóng lại tập tin thì bookmark sẽ biến mất). Còn shortcut thì cũng tương tự như bookmark nhưng chỉ giữ lại giữa các chầu hiệu đính. Khi bạn tạo một shortcut trên đoạn mã, thì một mục vào (entry) sẽ được ghi nhận trên một cửa sổ Task List cho phép di chuyển nhanh về đoạn mã được đánh dấu. Bạn có thể nhảy từ shortcut này qua shortcut kia giống như với bookmark.

Mỗi dòng lệnh trong đoạn mã chỉ có thể chứa hoặc một bookmark hoặc một shortcut, không thể cả hai trên cùng một hàng lệnh.

Có 4 Bookmark submenu và một **Add Task List Shortcut** submenu được liệt kê như sau, bảng 14-01:

Bảng 14-01: Các lệnh Bookmark

Các lệnh	Tổ hợp phím	Mô tả
Toggle Bookmark	Ctrl+K, Ctrl+K	Cho đặt hoặc gỡ bỏ một bookmark lên hàng hiện hành. Khi một bookmark được đặt để, một icon hình chữ nhật sẽ hiện lên ở cạnh trái của cửa sổ hiệu đính.
Next Bookmark	Ctrl+K, Ctrl+N	Nhảy về bookmark kế tiếp
Previous Bookmark	Ctrl+K, Ctrl+P	Nhảy về bookmark đi trước
Clear Bookmark	Ctrl+K, Ctrl+L	Gỡ bỏ tất cả các bookmark.
Add Tasl List Shortcut	Ctrl+K, Ctrl+H	Thêm một mục vào trên Task List (sẽ được mô tả về sau ở View Menu) đối với hàng lệnh hiện hành. Khi một shortcut được đặt để thì một icon kiểu mũi tên cong sẽ xuất hiện ở cạnh trái của cửa sổ hiệu đính.

Menu item này chỉ xuất hiện khi cửa sổ đoạn mã là cửa sổ hiện hành.

14.2.2.10 Outlining

Visual Studio .NET cho phép bạn *outline*, hoặc teo và bung ra, những phân đoạn của đoạn mã làm cho dễ nhìn thấy cấu trúc toàn diện. Khi một phân đoạn bị teo lại, nó xuất hiện với một dấu cộng (+) trên một ô nằm ở cạnh trái của cửa sổ đoạn mã. Chỉ việc click lên dấu cộng thì phân đoạn sẽ được bung ra lại.

Bạn có thể cho lồng nhau các vùng bị outline, như vậy một phân đoạn có thể chứa một hoặc nhiều phân đoạn khác bi teo lại. Có nhiều lệnh cho phép outlining dễ dàng hơn như theo bảng 14-02 sau đây:

Bảng 14-02: Các lệnh Outline

Các lệnh	Tổ hợp phím	Mô tả
Hide Selection	Ctrl+M, Ctrl+H	Cho teo đoạn văn bản hiện được tuyển chọn. Chỉ trên C#, lệnh này chỉ hiện lên khi outlining tự động bị off hoặc lệnh **Stop Outlining** được chọn ra.
Toggle Outlining Expansion	Ctrl+M, Ctrl+M	Cho đảo tình trạng outlining hiện hành của phân đoạn nằm tận cùng phía trong nơi con nháy đang nằm.
Toggle All Outlining	Ctrl+M, Ctrl+L	Cho đặt để tất cả các phân đoạn về cùng

		tình trạng outlining. Nếu một vài phân đoạn bị bung ra và một vài lại bị teo lại, thì tất cả sẽ bị teo lại.
Stop Outlining	Ctrl+M, Ctrl+P	Cho bung ra tất cả các phân đoạn. Cho gỡ bỏ ký hiệu outlining khỏi view.
Stop Hiding Current	Ctrl+M, Ctrl+U	Cho gỡ bỏ thông tin outlining đối với phân đoạn hiện được tuyển chọn. Chỉ trên C#, lệnh này chỉ hiện lên khi outlining tự động bị off hoặc lệnh **Stop Outlining** được chọn ra.
Collpase to Definitions	Ctrl+M, Ctrl+O	Tự động tạo những phân đoạn đối với mỗi procedure trong code window và cho teo lại tất cả các phân đoạn này.
Start Automatic Outlining		Khởi động lại automatic outlining sau khi ngưng.
Collapse Block		Trên C++ mà thôi. Giống như **Collpase to Definitions** ngoại trừ việc chỉ áp dụng đối với vùng đoạn mã chứa con nháy.
Collapse All In		Trên C++ mà thôi. Giống như Collapse Block ngoại trừ việc cho teo một cách đệ quy tất cả các cấu trúc lô gic trong một hàm trong một bước duy nhất.

Hành vi mặc nhiên của outlining có thể được đặt để bằng cách ra lệnh **Tools | Options | Text Editors | C# | Formatting.** Rồi cho các checkbox On.

14.2.2.11 IntelliSense

Công nghệ **IntelliSense** của Microsoft làm cho cuộc sống dân lập trình dễ thở hơn một chút. Nó có sẵn một help thời gian thực, cảnh ứng và xuất hiện ngay phía tay phải con nháy. Cấu trúc lệnh hiện lên sẽ giúp bạn giảm thiểu việc khó các phần tử cú pháp. Những ô liệt kê kéo xuống cung cấp cho bạn những hàm hành sự, thuộc tính của phạm trù hiện hành cho phép bạn dùng phím Tab hoặc click con chuột để đưa vào khỏi phải khỏ tránh sai lầm.

Các tính năng mặc nhiên của IntelliSense có thể được cấu hình hoá bằng cách ra lệnh **Tools | Options | Text Editor | C#** chẳng hạn.

Ngoài ra, menu item **Edit | IntelliSense** còn cung cấp các lệnh như theo bảng 14-03:

Bảng 14-03: Các lệnh IntelliSense

Các lệnh	Tổ hợp phím	Mô tả
List Members	Ctrl+J	Cho hiển thị một danh sách tất cả các biến thành viên đối với context hiện hành. Các phím gõ ra truy tìm kiểu incremental trên danh sách này. Bạn ấn bất cứ phím nào để đưa mục ngời sáng trên danh sách lên đoạn mã của bạn bạn dùng phím Tab để chọn mục trên danh sách mà khỏi khỏ thêm gì nữa.
Parameter Info	Ctrl+Shift+Space	Cho hiển thị một danh sách các số, tên và kiểu dữ liệu thông số đòi hỏi bởi hàm hành sự hoặc thuộc tính.
Quick Info	Ctrl+K, Ctrl+I	Cho hiển thị toàn bộ khai báo đối với bất cứ identifier nào (nghĩa là tên biến hoặc tên lớp) trên đoạn mã của bạn. Danh sách này cũng sẽ hiện lên khi con chuột rề rề lên bất cứ identifier nào.
Complete Word	Alt+Right Arrow hoặc Ctrl+Space	Tự động hoàn tất việc khỏ vào bất cứ identifier nào một khi bạn khỏ vào đủ nhận diện duy nhất identifier này. Việc này chỉ hoạt động nếu identifier được khỏ vào trong một vị trí hợp lệ trên đoạn mã.

Danh sách các thành viên sẽ hiện lên khi bạn khỏ vào dấu chấm theo sau tên bất cứ lớp nào hoặc tên thành viên.

Mỗi thành viên lớp sẽ được liệt kê, và mỗi kiểu dữ liệu thành viên sẽ được chỉ dẫn bởi một icon. Có những icon đối với hàm hành sự, vùng mục tin, thuộc tính, tình huống, v.v.. Ngoài ra, mỗi icon có thể có icon thứ hai nằm chồng cho biết việc truy cập của thành viên: public, private, protected, v.v.. Nếu không có icon truy cập, thì coi như là public.

Các icon object bao gồm các đối tượng như sau: Class, Constant, Delegate, Enum, Enum Item, Event, Exception, Global, Interface, Intrinsic, Macro, Map, Map Item, Method (hoặc Function), Module, Namespace, Operator, Property, Structure, Template, Typedef, Union, Unknown (hoặc Error), Variable (hoặc Field).

Còn các accessibility icon bao gồm: Shortcut, Friend, Internal, Private và Protected. Như đã nói Public không có icon.

14.2.3 View Menu

View menu thuộc loại cảnh ứng cho phép truy vấn vào vô số cửa sổ có sẵn trên Visual Studio .NET. Có thể một vài cửa sổ sẽ được mở thường xuyên bất cứ lúc nào, một số ít khi dùng đến, nếu không nói là không bao giờ.

View menu thuộc loại cảnh ứng nên nếu biểu mẫu bạn không có ô control nào, thì submenu **Tab Order** sẽ mang màu xám xịt.

Khi ứng dụng chạy, một số cửa sổ khác trở thành visible hoặc có sẵn. Những cửa sổ này có thể truy cập thông qua menu item **Debug | Windows**, chứ không phải từ menu item **View**.

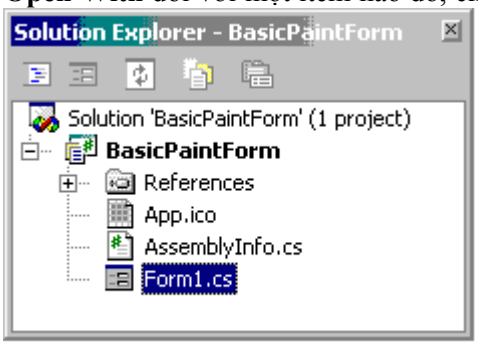
Visual Studio .NET có thể trữ nhiều cách bố trí cửa sổ khác nhau. Đặc biệt, nó biết nhớ một lô cửa sổ được mở hoàn toàn khác nhau trong những chầu gỡ rối, hơn là trong một phiên hiệu đính bình thường. Những layout này được trữ theo từng người sử dụng (per-user) chứ không phải theo từng dự án hoặc solution.

Trong mục này, chúng tôi chỉ giải thích những mục chọn không tự giải thích được.

14.2.3.1 Open & Open With

Các lệnh này cho phép bạn mở item hiện hành (nghĩa là item hiện được chọn ra trên Solution Explorer), trong chương trình mà bạn chọn. Lệnh **Open** sử dụng editor mặc nhiên, còn **Open With** thì lại cho phép bạn chọn từ một danh sách các chương trình. Bạn có thể thêm các chương trình khác lên danh sách này.

Ngoài ra, lệnh **Open With** cũng cho phép bạn mở một item với một editor mới bạn chọn trong Visual Studio .NET. Thí dụ, bạn có thể mở một tập tin trong binary viewer trong khi bình thường bạn có thể dùng resource viewer. Có thể hữu ích hơn cả, là bạn có thể khai báo editor mặc nhiên đối với một item, bằng cách cho hiện lên khung đối thoại **Open With** đối với một item nào đó, chọn một mục trên danh sách rồi click nút <Set As Default>. Thí dụ, bạn có thể làm cho một web form được mở trong code view thay vì trong design view như theo mặc nhiên.



Hình 14-15: Solution Explorer

14.2.3.2 Solution Explorer (Ctrl+Alt+L)

Các dự án và giải pháp sẽ được quản lý bằng cách sử dụng **Solution Explorer.** Cửa

sổ này tượng trưng giải pháp và dự án, và tất cả các tập tin, thư mục và item được chứa trong các tập tin này và được trình bày theo cây đẳng cấp nhìn thấy được.

Nếu cửa sổ Solution Explorer không hiện lên, bạn ra lệnh **View | Solution Explorer** từ trình đơn chính của Visual Studio .NET, hoặc ấn tổ hợp phím <Ctrl+Alt+L>. Hình 14-15 cho thấy một cửa sổ Solution Explorer. Trên đỉnh cửa sổ này, có một số nút thuộc loại cảnh ứng, xuất hiện hay không tùy thuộc item hiện được chọn ra trên Solution Exploer. Bảng sau đây cho biết ý nghĩa của các nút này. Chúng tôi cho đánh số thứ tự nút từ trái qua phải trên hình 14-15 để giải thích chi tiết các nút trên bảng 14-04.

Bảng 14-04: Các nút Solution Explorer

Nút số…	Tên nút	Shortcut Key	Mô tả
1	**View Code**	F7	Cho hiển thị mã nguồn trong cửa sổ chính. Chỉ hiện lên đối với các tập tin mã nguồn.
2	**View Designer**	Shift+F7	Cho hiển thị cửa sổ thiết kế trên cửa sổ chính. Chỉ hiện hình đối với những item với visual component.
3	**Refresh**	none	Cho làm tươi lại hiển thị Solution Explorer.
4	**Show All Files**	none	Cho hiện lên tất cả các tập tin trên Solution Explorer. Theo mặc nhiên, một số lớn các tập tin sẽ không hiển thị. Nếu bạn click nút Show All Files này thì hình 14-15 trở thành hình 14-16.
5	**Properties**	Alt+Enter	Nếu item hiện được ngời sáng là một solution hoặc một project, thì nó sẽ cho hiển thị một trang Properties đối với item này. Bằng không, di chuyển cho nháy lên cửa sổ đối với item này.
none	**Copy Project**	none	Nút này cho hiện lên một khung đối thoại dùng sao chép một dự án, với mục chọn cho biết chỉ chép những tập tin cần thiết để chạy hoặc tất cả các tập tin dùng cho việc triển khai về sau.

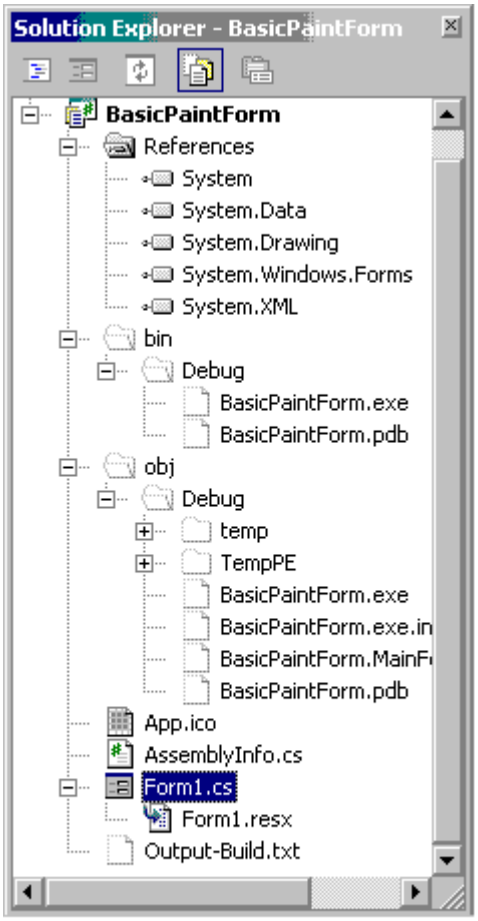
Ta cũng có khả năng cho hiển thị những tập tin linh tinh trên Solution Explorer. Muốn thế, bạn ra lệnh **Tools | Options | Environment | Documents,** rồi cho On ô control check box **Show Miscellaneous Files in Solution Explorer**.

Phần lớn các chức năng của Solution Explorer thường trùng lắp với các Visual Studio .NET menu item, mặc dù thường là dễ và trực giác hơn thực hiện điều gì đó trong Solution Explorer hơn là trên trình đơn chính. Bạn chỉ cần right click lên bất cứ item nào trên Solution Explorer, thì trình đơn shortcut sẽ hiện lên cho bạn làm việc. Hình 14-17 cho thấy các trình đơn shortcut đối với một solution, một project và một tập tin nguồn (*.cs chẳng hạn).

Nhiều điểm cần được giải thích đối với các lệnh trên các trình đơn shortcut này:

- Lệnh **Add** đối với solution và project cung cấp những submenu cho phép thêm item mới hoặc hiện hữu (**Add New Item** & **Add Existing Item**). Các lệnh này cũng có mặt trên trình đơn **Project** chính.

- **Set Startup Projects** và **Exclude From Project** cũng hiện diện trên trình đơn **Project** chính.



Hình 14-16: Solution Explorer sau khi nút Show All Files bị click

- Các lệnh Build và **Rebuild** trùng lắp với các lệnh cùng tên hiện diện trên trình đơn **Build** chính.

- Lệnh **Debug** bao gồm hai lệnh con (**Start new instance & Step into new instance**) cũng hiện diện trên trình đơn **Debug** chính.

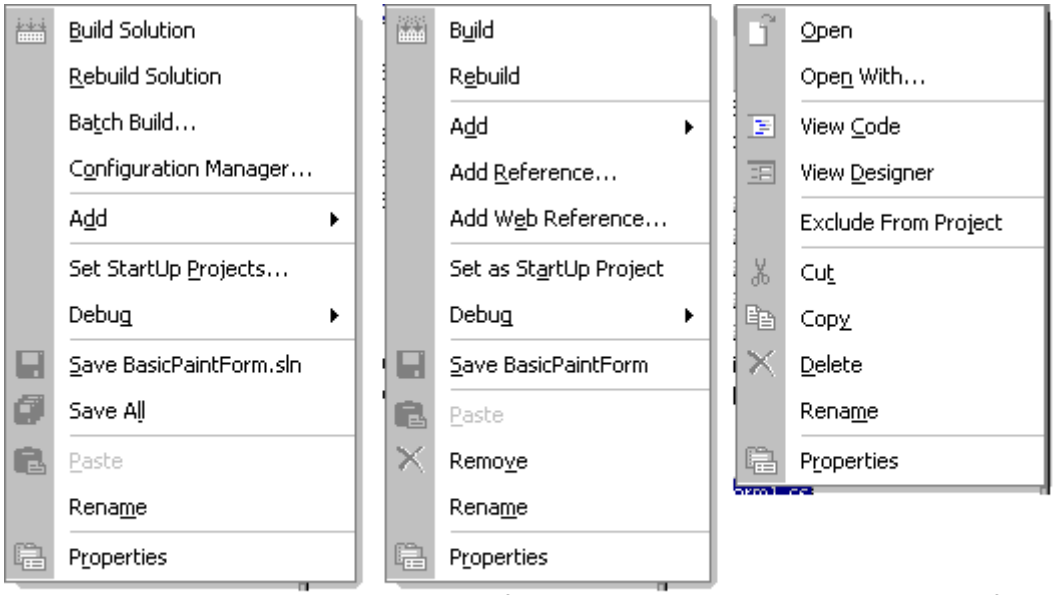
- Nếu mục **Properties** bị click đối với một tập tin nguồn, con nháy sẽ chuyển về cửa sổ **Properties**. Còn nếu mục **Properties** bị click đối với một solution hoặc project thì **Property Pages** đối với item này sẽ hiện lên.

14.2.3.3 Properties Windows (F4)

Cửa sổ **Properties** sẽ cho hiển thị tất cả các thuộc tính đối với item hiện được chọn ra. Một số thuộc tính (chẳng hạn **Font** và **Location**) sẽ có những subpropertie, được chỉ dẫn bởi dấu cộng (+) cạnh mục vào trên cửa sổ. Các trị thuộc tính ghi ở phía tay phải cửa sổ thuộc loại khả dĩ hiệu đính được.

Một điều có thể gây lúng túng là một vài item có thể có nhiều bộ thuộc tính thay vì chỉ một bộ. Thí dụ, một tập tin nguồn **Form** có thể cho thấy hai bộ thuộc tính tùy thuộc việc liệu xem bạn chọn tập tin nguồn trên Solution Explorer hoặc biểu mẫu như bạn thấy trong Design View.

Hình 14-18 cho thấy một cửa sổ Properties điển hình với thuộc tính Font được bung ra.



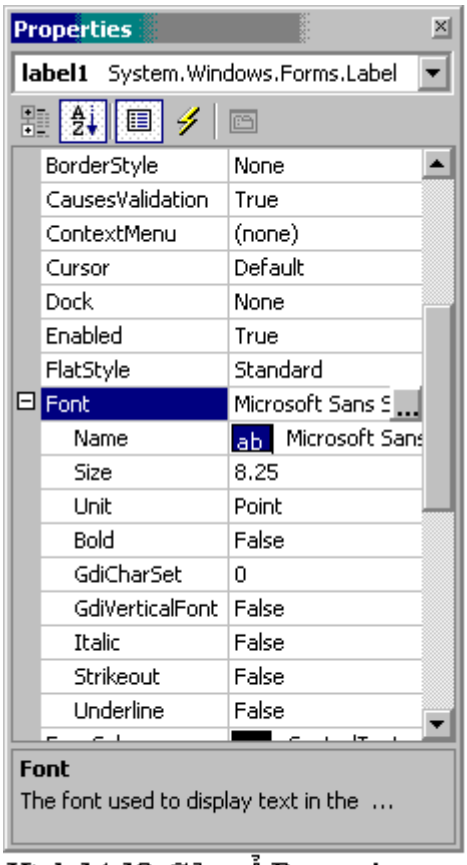
Hình 14-17: Các trình đơn shortcut đối với solution, project, và tập tin nguồn trên Solution Explorer.

Tên và kiểu dữ liệu của đối tượng hiện hành sẽ được hiển thị trên ô combo box trên đỉnh cửa sổ. Trên hình 14-18, bạn thấy đối tượng là Label1 kiểu dữ liệu Label, nằm trong namespace System.Windows.Forms.

Thuộc tính **Font** có những thuộc tính con mà bạn có thể đặt để trực tiếp (bằng cách khỏ vào) lên cửa sổ hoặc bằng cách click lên những ô liệt kê kéo xuống hoặc nút với 3 dấu chấm (ellipsis "**…**").

Cửa sổ **Properties** cũng có một số nút trên đỉnh: **Categorized, Alphabetic, Properties, Events,** và **Property Pages.** Hai nút đầu tiên từ trái qua phải (**Categorized, Alphabetic**) chuyển bật danh sách liệt kê theo category hoặc theo thứ tự ABC đối với các tên thuộc tính. Nút kế tiếp, **Properties**, sẽ cho hiển thị các thuộc tính của đối tượng. Còn nút cuối cùng phía tay phải, **Property Page**, sẽ hiển thị các trang thuộc tính đối với đối tượng, nếu có các trang này. Một vài đối tượng có cả hai cửa sổ **Properites** và **Property Page**. **Property Pages** sẽ cho hiển thị những thuộc tính bổ sung từ những thuộc tính đã cho thấy ở cửa sổ **Properties**.

Nếu dự án nằm trên C#, thì có thêm một nút mang icon hình sấm chớp, thường dùng để tạo những hàm thụ lý tình huống đối với một item.



Hình 14-18: Cửa sổ Properties

Đối với vài ô control khác, như **Calendar** chẳng hạn, sẽ có thêm một panel như là thành phần của cửa sổ **Properties** với những động từ, chẳng hạn **AutoFormat**.

Nằm ở cuối cửa sổ **Properies**, là một khung dùng mô tả giải thích công dụng thuộc tính bạn chọn ra ở trên.

14.2.3.4 Server Explorer (Ctrl+Alt+S)

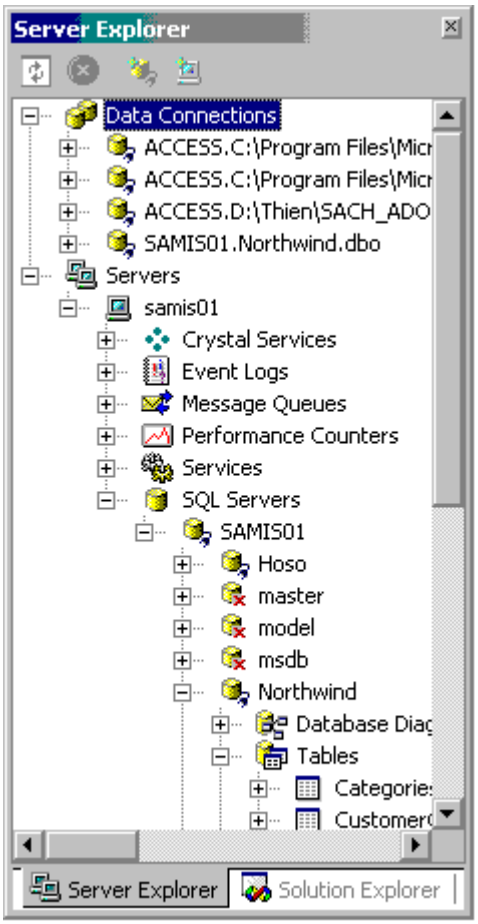
Server Explorer cho phép bạn truy cập bất cứ server nào trên mạng bạn kết nối về. Nếu bạn được phép, bạn có thể đăng nhập (log on), truy cập các dịch vụ hệ thống, mở những kết nối về căn cứ dữ liệu, truy cập và hiệu đính thông tin căn cứ dữ liệu, v.v.. Bạn cũng có thể lôi các mắt gút từ Server Explorer lên các dự án Visual Studio .NET, tạo những cấu kiện (component) qui chiếu về dữ liệu nguồn.

Hình 14-19 cho thấy một Server Explorer điển hình. Đây cũng là một cây đẳng cấp cho thấy các server có sẵn trên máy tính của bạn. Trong hình này chỉ có một server có sẵn: SAMIS01. Hình cho thấy ta xuống lần SQL Servers, cho thấy những bảng dữ liệu của căn cứ dữ liệu Northwind. Các bảng dữ liệu này, và các đối tượng khác trên TreeView này có thể được truy cập trực tiếp cũng như hiệu đính được từ cửa sổ.

14.2.3.5 Class View (Ctrl+Shift+C)

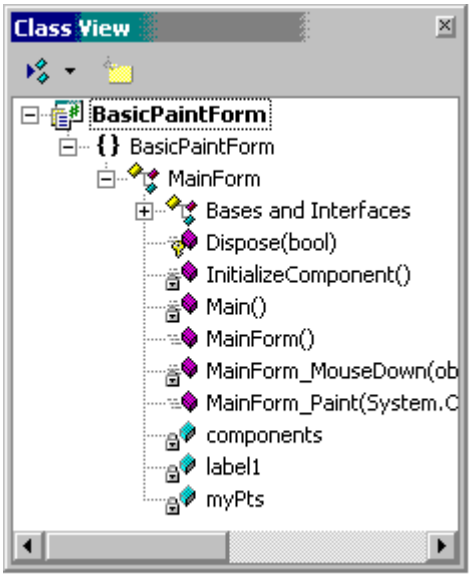
Cửa sổ **Class View** sẽ cho thấy tất cả các lớp trong solution theo kiểu cây đẳng cấp. Hình 14-20 cho thấy một **Class View** điển hình được bung ra. Các icon được sử dụng đối với cửa sổ này cũng tương tự như với IntelliSense.

Giống như với **Solution Explorer**, bất cứ item nào trong **Class View** có thể right click được để cho hiện lên một trình đơn shortcut với những mục chọn trình đơn cảnh ứng. Các trình đơn shortcut này cho phép bạn sắp xếp việc hiển thị các lớp trên solution hoặc dự án theo ABC, theo Type hoặc theo Access, hoặc thêm một hàm hành sự, thuộc tính hoặc vùng mục tin vào một lớp.



Hình 14-19: Server Explorer

Ngoài ra, trên Class View cũng có một nút tận cùng phía tay trái cho phép sắp xếp các lớp được liệt kê theo ABC, theo Type, theo Access, và gộp thành nhóm theo Type. Bạn chỉ cần click lên nút này chỗ mũi tên chúc xuống để cho hiện lên một trình đơn rồi chọn mục sắp xếp nào bạn muốn.



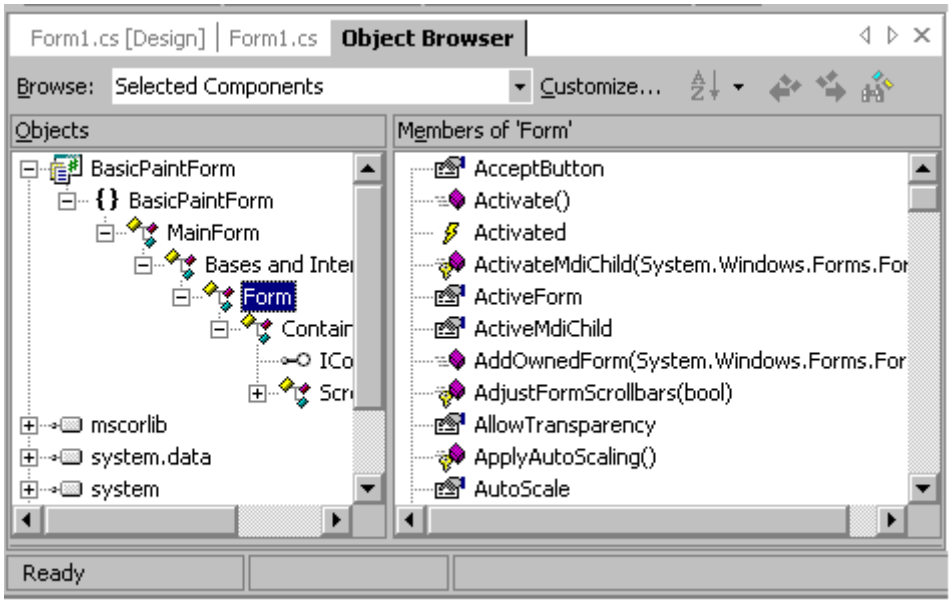
Hình 14-20: Class View

Còn nút thứ hai phía tay phải (hình 14-20), Folder, cho phép bạn tạo một thư mục ảo dùng tổ chức các lớp được liệt kê. Các thư mục này sẽ được cất trữ trong tập tin *.suo, như là thành phần của solution

Các thư mục này là ảo chỉ dùng để nhìn xem danh sách các lớp. Như vậy, các thư mục này không ảnh hưởng lên các items hiện hành. Các item được chép lên thư mục sẽ không bị di chuyển về mặt vật lý, và nếu các thư mục bị gỡ bỏ, các item sẽ không bị đánh mất. Bạn để ý là nếu bạn đổi tên hoặc gỡ bỏ một đối tượng khỏi đoạn mã nằm trong một thư mục, bạn cần lôi bằng tay item vào thư mục lại lần nữa để xoá sạch mắt gút sai lầm.

14.2.3.6 Object Browser (Ctrl+Alt+J)

Object Browser (bạn ra lệnh **Edit | Other Windows | Object Browser**) là một công cụ dùng quan sát các đối tượng (chẳng hạn namespace, lớp, và giao diện), cùng với các thành viên (chẳng hạn hàm hành sự, thuộc tính, biến, và tình huống). Hình 14-21 cho thấy một Object Browser điển hình.



Hình 14-21: Object Browser

Các đối tượng được liệt kê trong khung cửa tay trái theo kiểu cây đẳng cấp kéo lần xuống, còn các thành viên nếu có sẽ được liệt kê trên khung cửa tay phải. Các mắt gút của cây đẳng cấp này cũng mang những icon như bạn đã thấy ở IntelliSense.

Bằng cách right click lên một đối tượng trên khung trái hoặc một một thành viên trên

khung cửa tay trái bạn sẽ cho hiện lên một trình đơn shortcut với những mục chọn khác nhau.

14.2.3.7 Other Windows

Có nhiều cửa sổ khác đã được dồn một cục vào trình đơn **Other Windows**. Trình đơn này như vậy sẽ có một số submenu như sau:

Macro Explorer (Alt+F8)

Visual Studio .NET cho phép bạn có khả năng tự động hóa một số công tác mang tính lặp đi lặp lại bằng cách sử dụng macro. Một macro là một set những chỉ thị viết theo VB.NET được hoặc tạo ra bằng tay hoặc được ghi nhận bởi IDE và được đem trữ lên một tập tin. **Macro Explorer** là một trong những công cụ chính dùng nhìn xem (viewing), quản lý, và thi hành macro. Nó cung cấp việc truy cập vào Macro IDE. Macros sẽ được giải thích trong Tool menu.

Document Outline (Ctrl+Alt+T)

Cửa sổ **Document Outline** sẽ được dùng khi thiết kế các biểu mẫu Web để cung cấp một outline view của tài liệu HTML.

Task List (Ctrl+Alt+K)

Trong những ứng dụng lớn, việc giữ một danh sách những việc phải làm có thể trở thành cần thiết và hữu ích. Visual Studio .NET cung cấp chức năng này với cửa sổ **Task List**. Bạn cũng có thể cung cấp những shortcut đến những comment trên **Task List** kèm theo những token string, chẳng hạn TODO, HACK hoặc UNDONE. Ngoài ra, trình biên dịch cũng cho điền **Task List** với bất cứ những sai lầm biên dịch.

Command Windows (Ctrl+Alt+A)

Cửa sổ **Command** hoạt động theo hai chế độ: Command và Immediate.

Command mode thường được dùng đưa trực tiếp những lệnh, hoặc bỏ qua hệ thống trình đơn hoặc thi hành các lệnh không nằm trong hệ thống trình đơn. (bạn có thể thêm bất cứ lệnh nào vào trình đơn hoặc thanh công cụ bằng cách sử dụng **Tools | Customize…**).

Immediate mode thường được dùng khu gỡ rối để định trị những biểu thức, cũng như nhìn xem và thay đổi các biến, và các công tác gỡ rối khác. Cửa sổ Immediate và gỡ rối đã được đề cập đến trong Chương ***13 trong tập I bộ sách này.

Muốn biết thông tin chi tiết về việc sử dụng command window, đề nghị tham khảo SDK Documentation.

Output (Ctrl+Alt+O)

Cửa sổ Output dùng hiển thị các thông điệp tình trạng từ IDE dành cho lập trình viên, bao gồm các thông điệp gỡ rối, các thông điệp trình biên dịch, kết quả của các stored procedures (thủ tục làm sẵn), v.v..

14.2.4 Project Menu

Project menu cung cấp chức năng liên quan đến việc quản lý các dự án. Tất cả các chức năng được trình bày trong **Project** menu cũng hiện diện trong Solution Explorer. Thường thì bạn nên dùng các lệnh Project trong Solution Explorer vì nó mang tính trực giác hơn.

14.2.4.1 Add…

Add menu bao gồm nhiều lệnh cho phép bạn thêm một item mới hoặc hiện hữu vào dự án. Các lệnh này khá rõ ràng khỏi giải thích dài dòng, cung cấp cùng chức năng như là những item tương đương đã được mô tả trước đây trong lệnh **File**. Đây bao gồm:

- **Add Windows Form**: thêm một biểu mẫu Windows
- Add Inherited Form: thêm một biểu mẫu được kế thừa
- **Add User Control**: thêm một ô control thuộc lớp User Control.
- Add Inherited Control: thêm ô control được kế thừa từ lớp Control
- **Add Component**: thêm một cấu kiện .DLL
- **Add Class**: thêm một lớp
- **Add New Item** (Ctrl+Shift+A)
- **Add Existing Item** (Shift+Alt+A).

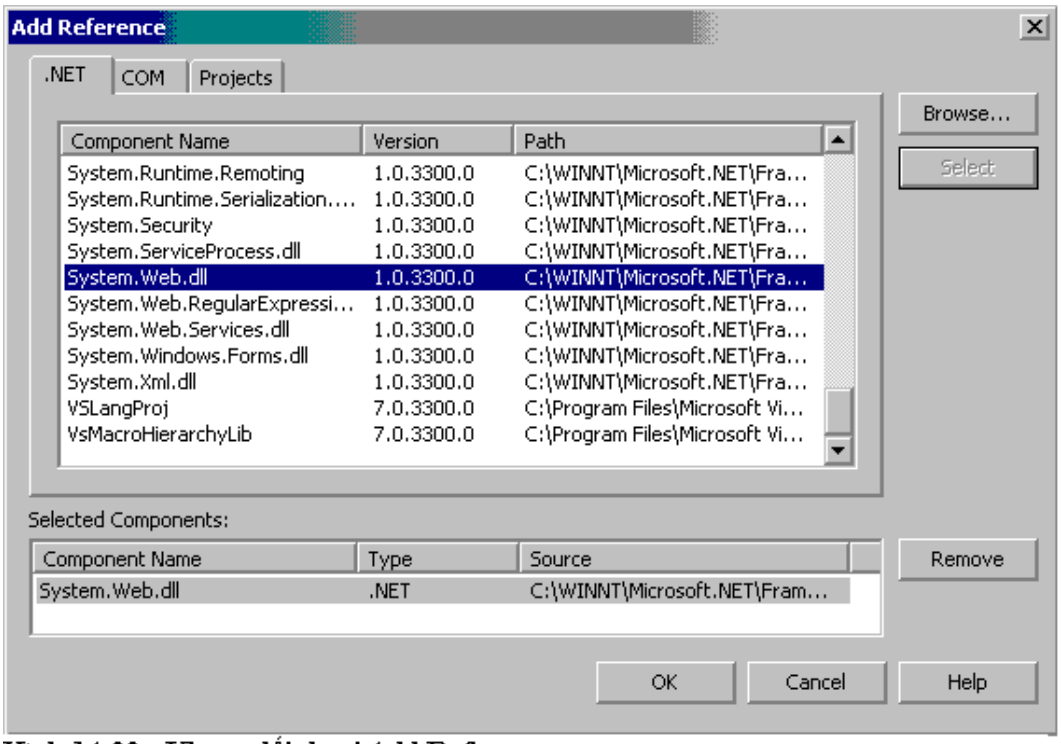
14.2.4.2 Exclude From Project

Lệnh **Exclude From Project** này sẽ gỡ bỏ tập tin khỏi dự án nhưng để yên tập tin trên đĩa cứng. Việc này khác lệnh **Delete** mà ta tìm thấy trong các trình đơn shortcut trên Solution Explorer. Lệnh **Delete** gỡ bỏ tập tin khỏi dự án và đồng thời gỡ bỏ tập tin khỏi đĩa cứng luôn (và cho về Recycle Bin). Nếu có một tập tin nguồn lực (resource file) được gắn liền với tập tin, thì nó cũng sẽ bị gỡ bỏ luôn khỏi dự án và khỏi đĩa cứng nếu sử dụng **Delete**.

Lệnh **Exclude From Project** cũng có sẵn trên Solution Explorer bằng cách right click lên tên tập tin.

14.2.4.3 Add Reference…

Lệnh **Add Reference** cũng có sẵn trên Solution Explorer bằng cách right click lên tên một dự án. Trong cả hai trường hợp (menu hoặc Solution Explorer) thì một khung đối thoại **Add Reference**, như theo hình 14-22, sẽ hiện lên:



Hình 14-22: Khung đối thoại Add Reference

Khung đối thoại này cho phép bạn qui chiếu về những assembly hoặc DLL nằm ngoài dự án đối với ứng dụng của bạn, bằng cách làm cho những lớp, hàm hành sự và thành viên mang tính public được chứa trong nguồn lực được qui chiếu sẵn sàng được sử dụng bởi ứng dụng của bạn.

Bạn thấy là trên khung đối thoại **Add Reference,** có 3 Tab: .NET, COM, và Project. Nguồn lực được qui chiếu thuộc thành phần nào thì bạn chọn Tab nấy để tiến hành chọn lựa.

14.2.4.4 Add Web Reference…

Lệnh **Add Web Reference** cũng có sẵn trên Solution Explorer bằng cách right click lên một dự án, cho phép bạn thêm một web reference vào dự án, trở thành một ứng dụng tiêu thụ dịch vụ web của một Web Service.

Chương 9 tập 2 bộ sách này sẽ đề cập đến Web Services và các ứng dụng phân tán.

14.2.4.5 Set as StartUp Project

Lệnh này cũng có sẵn trên Solution Explorer bằng cách right click lên một dự án, cho phép bạn khai báo dự án nào là startup project. Dự án nào được ngời sáng trên Solution Explorer khi lệnh này được thi hành sẽ là startup project.

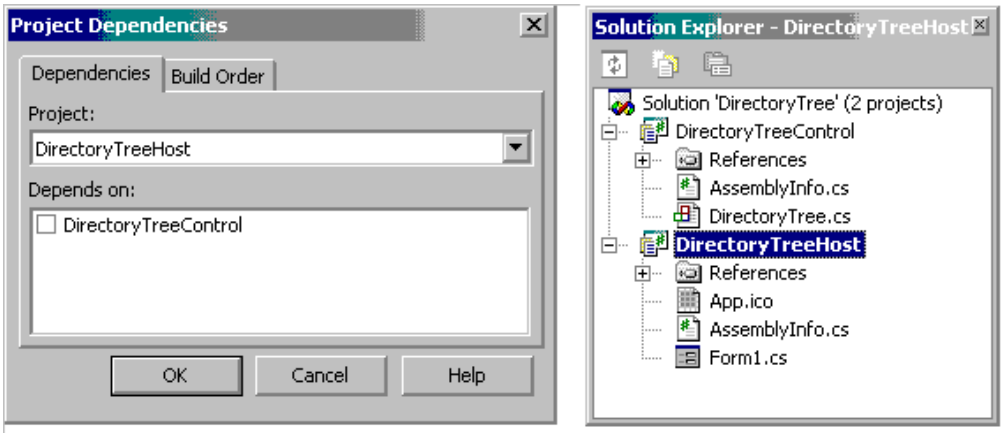
Nếu trong một solution có nhiều dự án, thì bạn phải khai báo dự án nào là dự án khởi động (startup project). Dự án khởi động là dự án sẽ được thi hành khi Bạn Build | Start.

14.2.4.6 Project Dependencies/Project Build Order

Các lệnh này chỉ nhìn thấy được khi một solution chứa nhiều dự án (hình 14-23), và cũng có sẵn trên Solution Explorer bằng cách right click lên tên dự án, cho hiện lên khung đối thoại như theo hình 14-23 cho phép bạn điều khiển việc xây dựng thứ tự Build của các dự án trong một solution. Khung đối thoại có hai Tab: Tab **Dependencies** và Tab **Build Order**.

Lệnh **Project Dependencies** cho phép bạn khai báo, đối với mỗi dự án trong solution, thì những dự án nào tùy thuộc dự án này. Các dự án bị lệ thuộc phải được build trước. Còn lện **Project Build Order** sẽ trình bày danh sách tất cả các dự án theo thứ tự được build.

Bạn để ý là nếu những sử dụng Project References (như được thêm vào với khung đối thoại Add Reference đã được đề cập trước đây) bạn không thể có khả năng hiệu đính những qui chiếu này. **Project Dependencies** được suy ra khi có những qui chiếu giữa các dự án trên cùng một solution. Ngoài ra, bạn cũng không thể thay đổi **Build Order** trong bất cứ trường hợp nào. Nó bao giờ cũng được suy ra từ những lệ thuộc này, cho dù những lệ thuộc này là tự động được suy ra hay không.



Hình 14-23: Solution Explorer với hai dự án và khung đối thoại Project Dependencies và Build Order.

14.2.5 Build Menu

Build menu có những mục chọn lo việc xây dựng các dự án hoặc solution. Ngoài ra, nó còn có mục trình đơn **Configuration Manager** lo cấu hình hoá tiến trình build. **Build** menu sẽ được đề cập đến chi tiết ở Chương 20, Tập 5, "Lập trình ASP.NET" khi đề cập đến việc triển khai ứng dụng.

14.2.6 Debug Menu

Debug menu cho phép bạn khởi động một ứng dụng có gỡ rối hoặc không gỡ rối, cho đặt để những chốt ngừng (breakpoint) trong đoạn mã và điều khiển chầu gỡ rối. Trong chương 3 chúng tôi đã đề cập chi tiết trình đơn **Debug** này. Đề nghị bạn trở lui lại chương kể trên để tham khảo.

14.2.7 Data Menu

Trình đơn **Data** thuộc loại cảnh ứng, chỉ hiện hình khi đang ở chế độ thiết kế, và không có sẵn khi hiệu đính các trang đoạn mã. Trình đơn **Data** chỉ hiện diện khi có những ô control data trên biểu mẫu.

Chương 12, "Data Controls", tập 3, "GUI và User Controls", và các chương trên tập 4, "Lập trình căn cứ dữ liệu với ADO.NET", thuộc bộ sách này sẽ đề cập đến Data Control và data binding nên bạn sẽ có dịp tìm hiểu trình đơn **Data** này.

14.2.8 Format Menu

Format menu chỉ hiện hình khi bạn đang ở chế độ thiết kế, và các lệnh thuộc trình đơn này chỉ có sẵn khi một hoặc nhiều ô control trên biểu mẫu của tuyển chọn.

Format menu cho phép bạn điều khiển kích thước và cách bố trí (layout) của các ô control, mặc dù nhiều mục chọn bị xám đi đối với những ô control trên web form. Bạn có thể:

- Canh ngay thẳng các ô control với một khung lưới (**Align | To Grid**) hoặc với những ô control khác theo sáu cách khác nhau: **Align | Lefts, Align | Centers, Align | Rights, Align | Tops, Align | Middle, Align | Bottoms.**

- Canh giữa biểu mẫu theo chiều ngang hoặc đứng: **Center in Form | Vertically, Center in Form | Horizontally**.

- Thay đổi kích thước của một hoặc nhiều ô control lớn hơn, nhỏ thua hoặc bằng nhau: **Make Same Size (Width, Size To Grid, Height, Both).**

- Kiểm soát khoảng cách theo chiều ngang (**Horizontal Spacing**) hoặc chiều đứng (**Vertical Spacing**).

- Cho di chuyển ô control tiến về trước hoặc lui về sau theo vertical plane (Z order) của biểu mẫu: **Order | Bring To Front, Order | Send to Back.**

- Khóa chặt một ô control không cho thay đổi kích thước hoặc vị trí: **Locks Control**.

Muốn hoạt động trên nhiều ô control, bạn cho chọn các ô control theo một trong nhiều cách sau đây:

- Bạn giữ rịt phím <Shift> hoặc <Ctrl> trong khi tiếp tục click lên các ô control mà bạn muốn chọn ra.

- Bạn dùng con chuột để click và lôi selection box xung quanh tất cả các ô control cần được chọn ra. Nếu bất cứ phần nào của một ô control rơi vào lòng selection box, thì ô control này sẽ được bao gồm.

- Muốn rã chọn (unselect) một ô control, bạn ấn giữ phím <Shift> hoặc <Ctrl> trong khi click lên ô control.

- Muốn rã chọn tất cả các ô control, bạn chọn một ô control khác hoặc ấn phím <Esc>.

Khi hoạt động trên nhiều ô control, ô control nào được chọn ra chót nhất sẽ được dùng làm baseline. Nói cách khác, nếu bạn muốn làm cho tất cả các ô control đều cùng kích thước, thì các ô control này sẽ giống kích thước của ô control chót nhất được chọn ra. Tương tự như thế, nếu canh một nhóm ô control, thì chúng sẽ canh theo với ô control chót nhất được chọn ra.

Khi ô control được chọn ra, sẽ hiện lên 8 cái "quai xách" (resizing handle) có màu đen ngoại trừ baseline có những quai xách mang màu trắng.

14.2.9 Tools Menu

Tools menu cho thấy những lệnh cho phép bạn truy cập một số rộng lớn chức năng, đi từ kết nối với các căn cứ dữ liệu đến truy cập những công cụ ngoại lai dùng đặt để các mục chọn trên IDE. Sau đây một số lệnh hữu ích sẽ được mô tả:

14.2.9.1 Connect to Device…

Lệnh này sẽ cho hiện lên một khung đối thoại cho phép bạn kết nối với hoặc một thiết bị vật lý mobile hoặc một emulator.

14.2.9.2 Connect to Database…

Lệnh **Connect to Database** mặc nhiên cho hiện lên một khung đối thoại **DataLink Properties** cho phép bạn chọn ra một server, đăng nhập vào server này, rồi kết nối căn cứ dữ liệu vào server. Microsoft SQL Server là căn cứ dữ liệu, nhưng Tab Provider cho phép bạn kết nối về bất cứ căn cứ dữ liệu nào đó khác đi, bao gồm Oracle, ODBC, hoặc OLE DB provider. Tập IV bộ sách này sẽ nói về ADO.NET. Bạn tha hồ nghiên cứu lệnh này.

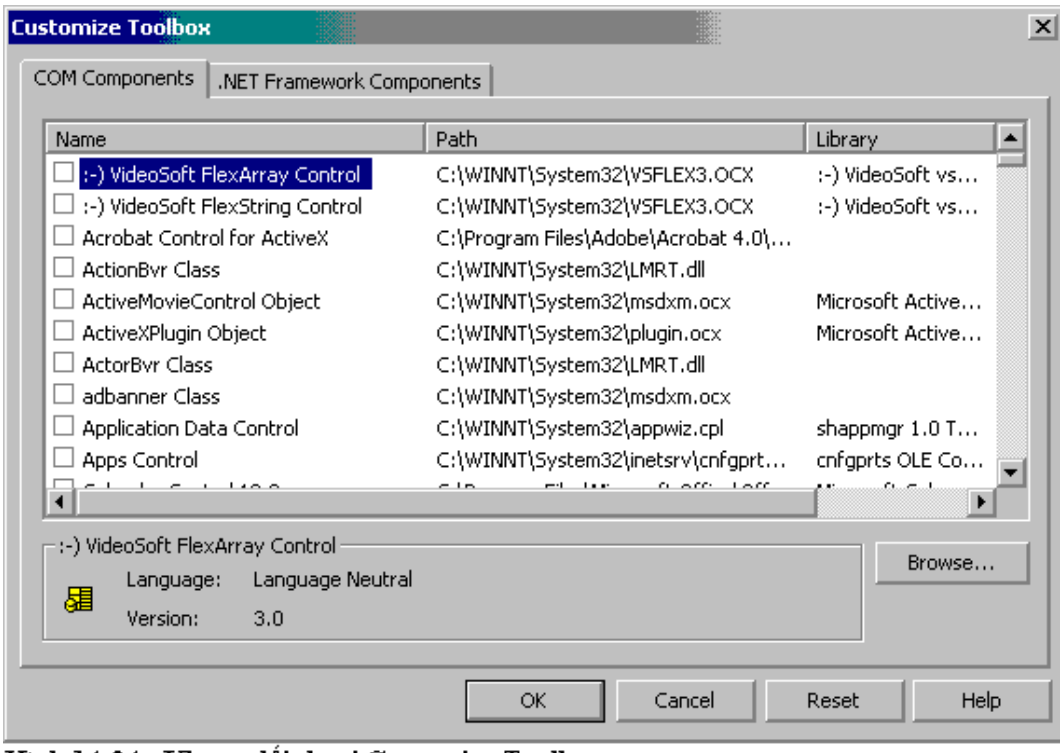
14.2.9.3 Connect to Server…

Lệnh **Connect To Server** này sẽ cho hiện lên khung đối thoại **Add Server** cho phép bạn khai báo một server phải nối về, hoặc theo tên hoặc theo địa chỉ IP. Nó cũng để cho bạn kết nối sử dụng một username và password khác.

Cũng khung đối thoại **Add Server** này hiện lên khi bạn right click lên mắt gút **Servers** trên Server Explorer, rồi chọn mục **Add Server…** trên trình đơn shortcut.

14.2.9.4 Add/Remove Toolbox Item…

Lệnh này cho hiện lên một khung đối thoại như theo hình 14-24. Khung đối thoại này có hai Tab: **NET Framework component** và **COM components**. Tab **Com component** dùng thêm vào những cấu kiện COM được thừa hưởng từ những phần mềm đi trước để lại (gọi là legacy component), còn Tab **NET Framework component** dùng thêm vào toolbox những cấu kiện thuần túy .NET CLR. Các cấu kiện hiện có sẵn trên mày của bạn sẽ được liệt kê lên trên hai Tab này phân theo loại COM hoặc .NET CLR. Ô control check box bên cạnh trái của mỗi cấu kiện cho phép bạn check hoặc uncheck cho biết là bao gồm hoặc loại bỏ cấu kiện vào toolbox hay không.



Hình 14-24: Khung đối thoại Customize Toolbox

14.2.9.5 Build Comment Web Pages…

Lệnh này sẽ cho hiện lên một khung đối thoại cho phép bạn sưu liệu ứng dụng của bạn thông qua các trang HTML. Các trang HTML sẽ tự động hiển thị cấu trúc đoạn mã của ứng dụng. Các dự án sẽ được liệt kê như là những hyperlink. Click lên một dự án sẽ cho hiện lên một trang HTML cho thấy tất cả các lớp như là hyperlink nằm ở cạnh trái

trang. Việc click lên bất cứ lớp nào sẽ liệt kê ra ở phía tay phải tất cả các thành viên lớp với những mô tả.

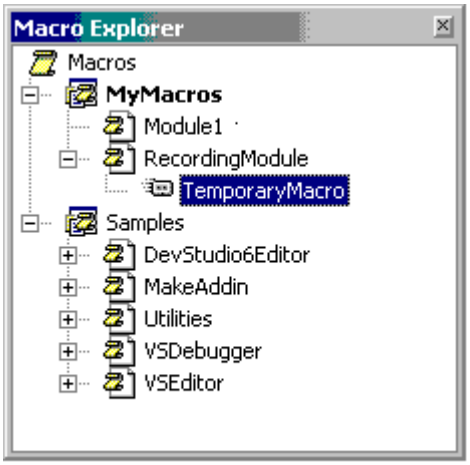
Nếu ngôn ngữ lập trình của bạn chịu hỗ trợ XML Code Comment (như với C#, nhưng VB.NET thì không), thì bạn có thể thêm chú giải (comment) riêng của bạn vào mã nguồn và những chú giải này sẽ được hiển thị trên các trang web.

Comment Web Pages sẽ được tạo theo mặc nhiên trên một thư mục con của dự án mang tên CodeCommentReport.

14.2.9.6 Macros

Macro là một chức năng ngon lành cho phép bạn tự động hoá các công việc trên IDE. Macro có thể hoặc được mã hoá bằng tay hoặc được ghi nhận bởi IDE khi bạn thực hiện các công việc này. Nếu bạn để cho IDE ghi nhận, thì về sau bạn có thể xem xét và hiệu đính macro code mà IDE đã tạo ra. Việc này cũng tương tự như chức năng macro trên Word hoặc Excel.

Bạn để ý: bạn nên nhớ là macro recording không làm gì cả đối với bất cứ việc gì xảy ra trong lòng một khung đối thoại. Thí dụ, nếu bạn ghi nhận thay đổi thuộc tính trong một Property Pages của một dự án, thì macro được ghi nhận sẽ mở Property Pages nhưng không làm gì cả ở đây.



Hình 14-25: Macro Explorer

Bạn có thể ghi dễ dàng một macro tạm thời bằng cách ra lệnh **Macros | Record Temporary Macro**, hoặc ấn tổ hợp phím <Ctrl+Shift+R>. Macro tạm thời này có thể sau đó được chơi lại bằng cách ra lệnh **Macros | Run Temporary Macro**, hoặc ấn tổ hợp phím <Ctrl+Shift+P>. Macro tạm thời có thể được cất trữ bằng cách ra lệnh **Macros | Save Temporary Macro**, và tự động cho hiện lên **Macro Explorer**, mà chúng tôi sẽ giải thích trong chốc lát.

Macro sẽ được quản lý bởi cửa sổ **Macro Explorer**, như theo hình 14-25. Muốn cho cửa sổ này hiện lên bạn ra lệnh **Macros | Macro Explorer** hoặc ấn tổ hợp phím <Alt+F8>. Hình 14-25 cho thấy sau khi ghi một macro tạm thời. Bằng cách right click lên một macro trên **Macro Explorer**, một trình đơn shortcut sẽ hiện lên với 4 mục chọn: **Run, Edit, Rename** và **Delete:**

Run

Mục chọn này cho thi hành macro được ngời sáng. Ta cũng có thể cho thi hành macro bằng cách double click lên tên maco.

Edit

Lệnh này sẽ cho hiện lên cửa sổ IDE hiệu đính macro, nơi mà bạn có thể hiệu đính macro. Ngôn ngữ macro là VB.NET, không cần biết đến ngôn ngữ lập trình nào bạn đang dùng. Macro editing IDE cũng có thể được triệu gọi bằng cách ra lệnh **Macros | Macro IDE** hoặc ấn tổ hợp phím <Alt+F11>.

Rename

Cho phép bạn đổi lại tên của macro.

Delete

Cho gỡ bỏ macro khỏi tập tin macro.

Tất cả các macro đều được chứa trong một *macro project* mang tên **MyMacros** theo mặc nhiên. Dự án này bao gồm một tập tin nhị phân mang tên MyMacros.vsmacros (trừ khi bạn chọn chuyển nó thành multiple files format), và tập tin này nằm trong thư mục *Documents and Settings* đối với mỗi người sử dụng. Bạn có thể tạo một macro project mới bằng cách ra lệnh **Macros | New Macro Project** hoặc bằng cách right click lên mắt gút gốc trên Macro Explorer rồi chọn mục New Macro Project trên trình đơn shortcut. Trong trường hợp nào đi nữa, khi khung đối thoại New Macro Project hiện lên bạn có thể khai báo tên và nơi cư ngụ của tập tin macro project.

Macro project có thể chứa những module, là những đơn vị đoạn mã. Mỗi module chứa nhiều subroutine, tương ứng với macro. Thí dụ, macro mang tên **TemporaryMacro** như theo hình 14-25 là subroutine **TemporaryMacro** được chứa trong module mang tên **RecordingModule**, và là thành phần của **MyMacros** project.

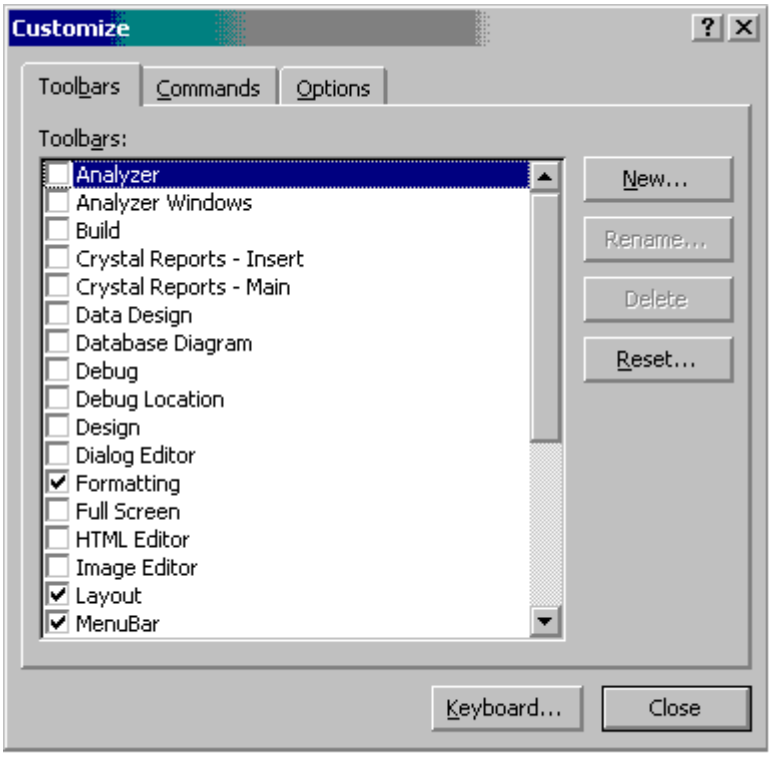
14.2.9.7 External Tools…

Tùy thuộc vào những mục được chọn ra vào lúc Visual Studio .NET được cài đặt trên máy của bạn, có thể bạn có một trong những công cụ ngoại lai có sẵn trên **Tools** menu của bạn. Đây có thể bao gồm các công cụ như Create GUID, ATL/MFC Trace Tool, hoặc Spy++. Việc sử dụng các công cụ này là ngoài tầm tay bộ sách này.

Lệnh **Tools | External Tools…** cho phép bạn thêm vào Tools menu những công cụ ngoại lai. Khi click lên lệnh này, một khung đối thoại **External Tools** hiện lên với những vùng mục tin dành cho tool title, lệnh thi hành công cụ, bất cứ đối mục nào cần có, và thư mục ban đầu, cũng những ô control check box cho những hành xử khác nhau.

14.2.9.8 Customize…

Lệnh **Customize…** cho phép bạn "uốn nắn" (customize) những khía cạnh khác nhau của giao diện IDE đối với người sử dụng (Lệnh **Options**… sẽ được mô tả trong mục tới, cho phép bạn đặt để một loạt những mục chọn đối với các chương trình khác). Khi click lệnh này thì một khung đối thoại **Customize** hiện lên, như theo hình 14-26, với 3 Tab: **Toolbars, Commands,** và **Options,** và một nút <Keyboard…> cho phép bạn uốn nắn giao diện IDE trên 4 địa hạt khác nhau**.**



Hình 14-26: Khung đối thoại Customize- Tab Toolbars

Toolbars

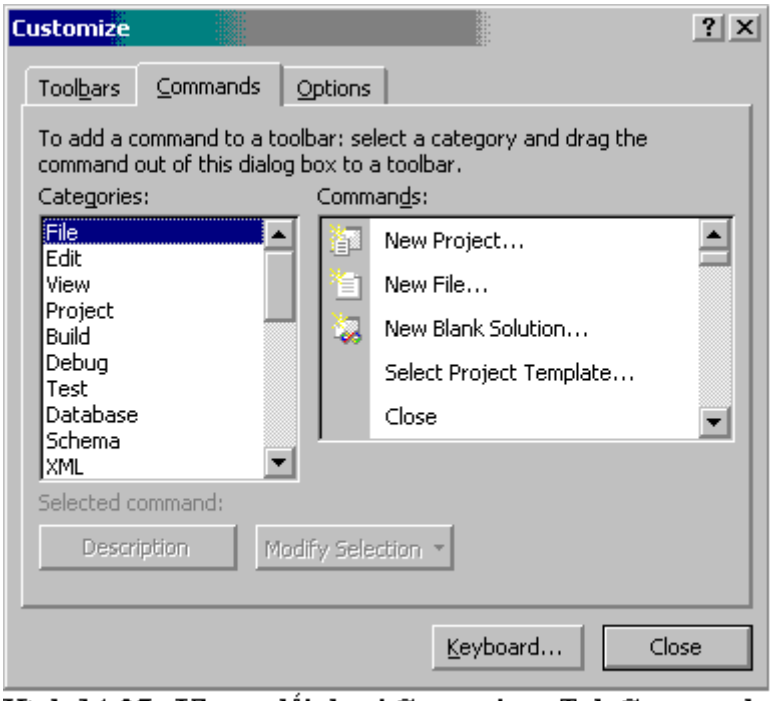
Tab này, như theo hình 14-26, cho trình bày một danh sách các thanh công cụ có sẵn, kèm theo ô check box phía tay trái, cho biết những thanh công cụ nào hiện nhìn thấy được. Bạn có thể điều khiển việc nhìn thấy được hay không của một thanh công cụ bằng cách cho On hoặc Off một item trên danh sách này. Một phương án khác là ra lệnh **View | Toolbars** trên trình đơn **View**, rồi chọn check/uncheck một thanh công cụ nào đó.

Bạn cũng có thể tạo mới những thanh công cụ (sử dụng nút <New…>), cho đổi tên (nút <Rename…>), hoặc gỡ bỏ (nút <Delete>) những thanh công cụ hiện hữu,

hoặc reset tất cả các thanh công cụ về phiên bản cài đặt nguyên thủy trên Tab này (nút <Reset…>).

Commands

Tab Commands, như theo hình 14-27, cho phép bạn thêm vào hoặc gỡ bỏ những lệnh khỏi một thanh công cụ hoặc thay đổi những nút đã nằm sẵn trên thanh công cụ.



Hình 14-27: Khung đối thoại Customize - Tab Commands

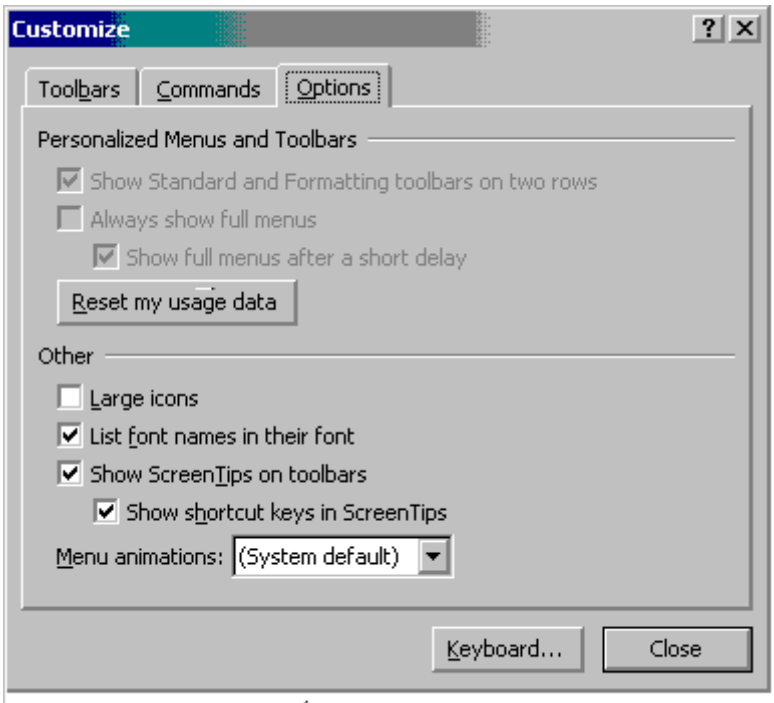
Muốn thêm một command lên thanh công cụ, bạn chọn category và command từ hai ô liệt kê trên khung đối thoại, rồi sau đó lôi command đặt lên thanh công cụ mong muốn.

Muốn gỡ bỏ một command khỏi thanh công cụ, bạn lôi nó đi khỏi thanh công cụ rồi bỏ mặc đâu đó trên IDE trong khi khung đối thoại Customize Command đang hiện diện.

Nút <Modify Selection> chỉ hiệu lực khi một nút trên thanh công cụ hiện hữu được tuyển chọn. Nó cho phép bạn thực hiện những việc như đổi tên nút, gỡ bỏ nút, đổi hình ảnh được hiển thị trên nút, thay đổi style hiển thị của nút (chỉ hình ảnh mà thôi, chỉ văn bản mà thôi, v.v..), và tổ chức các nút thành nhóm.

Options

Tab Options, như theo hình 14-28, cho phép bạn thay đổi dáng dấp bên ngoài của thanh công cụ. Các ô control check box của **Personalized Menus and Toolbars** bao giờ cũng không có sẵn và mang màu xám xịt. Còn các ô check box trong nhóm **Others** cho phép chọn kích thước icon trên các nút, điều khiển tool tips và cách các trình đơn xuất hiện thế nào (menu animation).



Hình 14-28: Khung đối thoại Customize -Tab Options

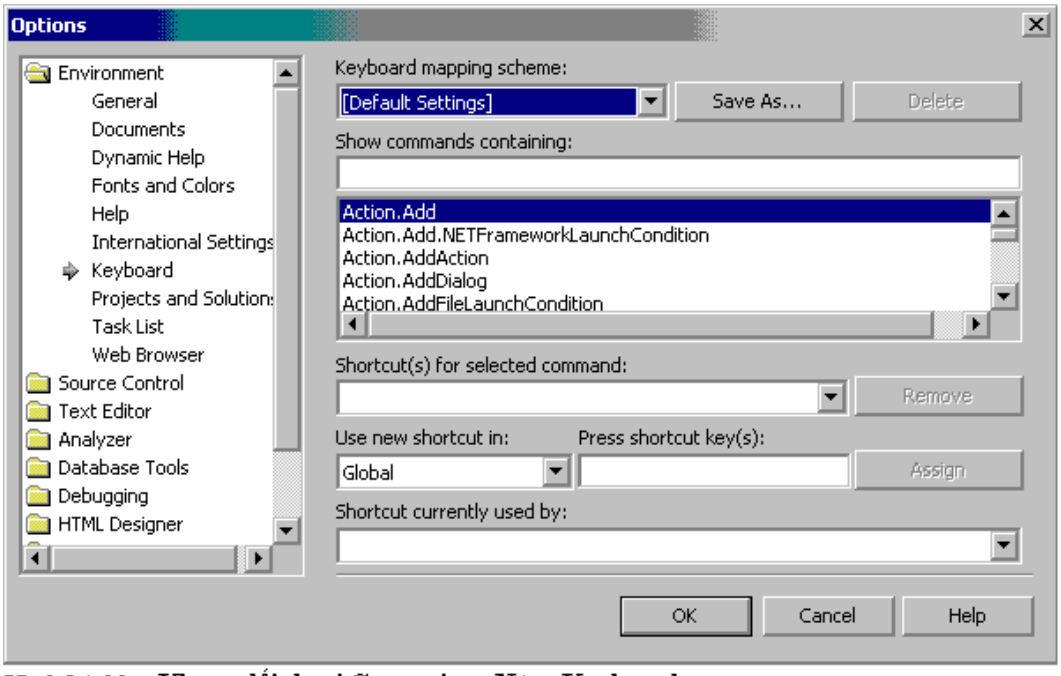
Keyboard…

Nút <Keyboard…> sẽ cho hiện lên trang Environment | Keyboard, như theo hình 14-29. Khung đối thoại này cũng hiện lên khi bạn ra lệnh **Tools | Options** mà chúng tôi sẽ mô tả trong chốc lát. Trang này cho phép bạn định nghĩa và thay đổi các phím shortcut đối với những command.

14.2.9.9 Options…

Lệnh **Options…** cũng cho hiện lên khung đối thoại Options giống như hình 14-29. Khung đối thoại này cho phép đặt để vô số mục chọn.

Khung đối thoại cho hiển thị một cấu trúc cây đẳng cấp theo category nằm ở khung cửa phía tay trái. Chọn bất cứ category nào cho phép bạn kéo dài cái cây. Còn khi click lên một mắt gút sẽ cho hiện lên những thuộc tính có sẵn nằm trên khung cửa phía tay phải khung đối thoại.



Hình 14-29: Khung đối thoại Customize - Nút <Keyboard...>

Phần lớn các mục chọn có sẵn quá rõ ràng khỏi giải thích chi thêm. Nếu bạn lúng túng thì nút <Help> ở cuối khung đối thoại sẽ cho hiện lên một Help cảnh ứng liên quan đến tất cả các thuộc tính có ý nghĩa đối với item hiện hành.

14.2.10 Window Menu

Window menu là lệnh Window của một ứng dụng Windows chuẩn. Nó hiển thị một danh sách các cửa sổ hiện được mở, cho phép bạn đem bất cứ cửa sổ nào về mặt tiền bằng cách click lên tên cửa sổ. Bạn để ý là tất cả các cửa sổ tập tin hiện được hiển thị trên

IDE đều mang một tab nằm ở cạnh phía trên đỉnh của cửa sổ design, dưới các thanh công cụ, và bạn có thể chọn cửa sổ bằng cách click lên tab mang tên cửa sổ.

Window menu thuộc loại cảnh ứng. bảng 14-05 sau đây cho liệt kê các menu item có sẵn trong các trường hợp khác nhau:

Bảng 14-05: Window menu item commands

Cửa sổ hiện hành	Mô tả các lệnh có sẵn
Design	**Auto Hide All** sẽ cho cất giấu tất cả các cửa sổ dockable. Cho click lên pushpin icon của cửa sổ sẽ cho **AutoHide** về off đối với cửa sổ này. **New Horizontal/Vertical Tab Group** sẽ tạo một bộ cửa sổ khác với bộ tabs riêng của mình. **Close All Documents** cho đóng lại tất cả các tài liệu. **Windows…**cho liệt kê tất cả các cửa sổ được mở.
Code	Giống như với cửa sổ Design, nhưng có thêm những lệnh sau đây: **New Window** cho tạo một cửa sổ mới chứa cùng tập tin như là cửa sổ hiện hành (sử dụng lệnh này để mở hai cửa sổ cùng một tập tin nguồn); **Split:** cho mở một cửa sổ thứ hai trên cửa sổ hiện hành dùng cho hai view khác nhau của cùng một tập tin; và **Remove Split** cho gỡ bỏ cửa sổ bị chẻ.
Dockable	Loại cửa sổ này bao gồm cửa sổ Solution Explorer, Properties, Class View, Toolbox, v.v.. Các cửa sổ này là dockable (cập bến được), như được báo bởi pushpin icon nằm trên góc top right của mỗi cửa sổ.Các menu iem có sẵn giống như với một cửa sổ design, với việc bổ sung những lệnh liên quan đến dock, hide hoặc float một cửa sổ.

14.2.11 Help Menu

Help menu cho phép bạn truy cập một số submenu, mà chúng tôi sẽ giải thích dưới đây đối với những mục nào cần giải thích chi tiết.

14.2.11.1 Dynamic Help (Ctrl+F1)

Nếu bạn đang triển khai ứng dụng trên một máy tính với đầy đủ sức ngựa, thì Dynamic Help là một điều kỳ diệu. Bằng không, nó ngốn mất thành tích. (bạn có thể vô hiệu hóa nó bằng cách ra lệnh **Tools | Options | Environment | Dynamic Help**, rồi uncheck tất cả các ô control check box). Một cách khác, chỉ cần đóng lại cửa sổ là đủ

ngăn không cho thành tích xuống dốc, và theo cách này dynamic help vẫn còn nằm đó khi bạn cần đến nó.

Việc sử dụng Dynamic Help rất đơn giản. Bạn mở cửa sổ Dynamic Help bằng cách ra lệnh **Help | Dynamic Help** hoặc ấn tổ hợp phím <Ctrl+F1>. Sau đó bất cứ nơi nào có focus, cho dù ở cửa sổ thiết kế, cửa sổ mã nguồn hoặc cửa sổ dockable, thì hyperlink cảnh ứng sẽ xuất hiện trên cửa sổ Dynamic Help. Bạn chỉ cần click lên những link này thì đề mục help tương ứng sẽ hiện lên trên một cửa sổ khác.

14.2.11.2 Contents…/Index…/Search…

3 lệnh trên cho thấy những view khác nhau đối với hệ thống Help SDK, cho phép bạn truy tìm theo một bản mục lục (table of content - TOC), theo một chỉ mục incremental, hoặc theo một câu truy tìm. Kiểu truy tìm thứ nhất là một indexed search, trong khi hai kiểu truy tìm chót là full text search, do đó có thể bạn sẽ nhận kết quả khác nhau sử dụng những kiểu truy tìm khác nhau sử dụng cùng câu.

Bạn để ý: Hệ thống Help được trưng ra bởi những lệnh này đều là y chang đối với hệ thống Help được trưng ra bởi hai chỗ khác nhau bởi lệnh Start:

- Programs | MS Visual Studio .NET 2003 | MS Visual Studio .NET Documentation
- Programs | MS .NET Framework SDK | Documentation

Công cụ Help này sử dụng giao diện kiểu browser, với **Forward & Backward** navigation và **Favorites**. Danh sách các đề mục nằm ở khung cửa tay trái, còn nội dung bản thân đề mục thì nằm ở khung cửa tay phải, bao gồm cả hyperlink.

14.2.11.3 Index Results… (Shift+Alt+F2)

Khi lục tìm các Help topics theo Index, sẽ cò nhiều đề mục đối với một mục vào index. Trong trường hợp này, cửa sổ Index Results sẽ liệt kê các topics này, và tự động hiện lên (nếu nó bị đóng lại).

14.2.11.4 Search Results… (Shift+Alt+F3)

Cửa sổ Search Results cũng giống như Index Results, ngoại trừ danh sách các topics là do lục tìm theo giai đoạn Search.

14.2.11.5 Edit Filters…

SDK Help System là một hệ thống rất đồ sộ, với thông tin nằm trên toàn bộ bản dãy các topics có thể tìm thấy trong bất cứ cài đặt .NET nào. Lệnh Edit Filters cho phép bạn giới hạn việc truy tìm các topic. Thí dụ, nếu bạn chỉ làm việc thuần túy với C#, thì bạn có thể cho sàng lọc hoặc Visual C# hoặc Visual C# and Related.

14.2.11.6 Check For Updates

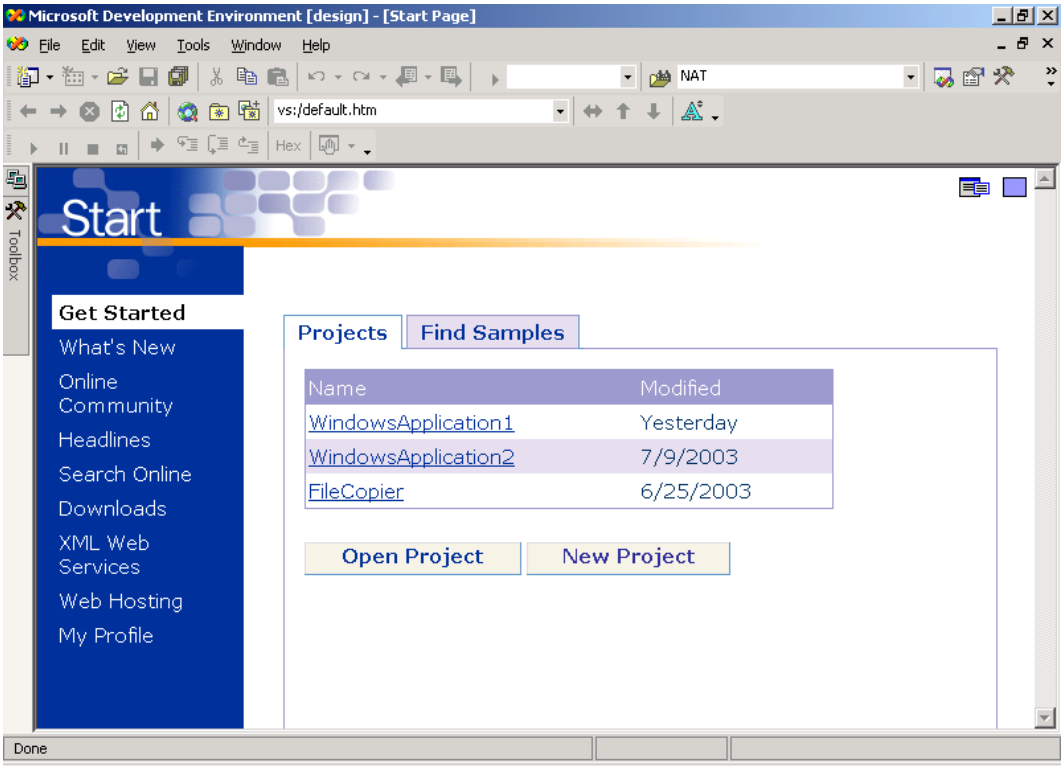
Lệnh này sẽ cho kiểm tra đối với những release services đối với phiên bản Visual Studio .NET hiện được cài đặt. Muốn cho lệnh này chạy, máy tính của bạn phải được kết nối với Internet. Nếu có sẵn một update, bạn sẽ được nhắc nhở cho đóng IDE lại trước khi service release được cài đặt.

14.3 Tạo một dự án

Một khi Visual Studio .NET đã được cài đặt, bạn có thể bắt đầu tạo dự án đầu tiên. Với Visual Studio .NET, ít khi bạn khởi đầu với một tập tin trắng mà bạn sẽ khỏ vào đoạn mã C# từ tay không, như bạn đã làm trong các chương của tập I của bộ sách này. Thay vào đó, bạn sẽ bảo cho Visual Studio .NET biết loại dự án nào bạn muốn tạo, và Visual Studio .NET sẽ tự động kết sinh đoạn mã C# phác thảo một khuôn giá (framework), một khung làm việc, đối với dự án này. Sau đó bạn thêm vào đoạn mã của bạn vào khuôn giá này. Thí dụ, bạn muốn viết một ứng dụng dựa trên Windows GUI, giao diện người sử dụng (hoặc theo từ ngữ Visual Studio .NET là một ứng dụng Windows Form), thì Visual Studio .NET sẽ bắt đầu tạo cho bạn một tập tin chứa đoạn mã nguồn C# cho phép tạo một biểu mẫu (form) cơ bản. Biểu mẫu này có khả năng "nói chuyện" với Windows, và tiếp nhận những tình huống. Nó cho phép được thu nhỏ hoặc phình lớn hoặc thay đổi kích thước v.v.., nhưng nó hiện chưa có bất cứ ô control nào cả hoặc một chức năng nào ra hồn. Và lúc này bạn sẽ thêm vào các chức năng bạn mong muốn có đối với biểu mẫu. Còn nếu bạn muốn một ứng dụng được dùng theo kiểu dòng lệnh (command line), nghĩa là một ứng dụng console, thì Visual Studio .NET cũng sẽ tạo cho bạn một namespace, một lớp với hàm hành sự **Main()** để bạn bắt đầu khởi đi. Lẽ dĩ nhiên là nếu bạn muốn bắt đầu từ số không, thì Visual Studio .NET cũng làm vừa lòng bạn bằng cách cho bạn một ứng dụng rỗng.

Khi bạn tạo dự án, Visual Studio .NETsẽ dàn dựng (set up) những mục chọn biên dịch mà có thể bạn cần cung cấp cho trình biên dịch C#. Ngoài ra, những thư viện lớp cơ bản nào bạn sẽ cần đến. Lẽ dĩ nhiên bạn có thể thay đổi những đặt để (setting) kể trên khi bạn hiệu đính nếu thấy cần thiết. Lần đầu tiên khi bạn khởi động Visual Studio .NET, bạn sẽ thấy hiện lên **Start Page**. (Hình 14.30).

Trang này là một trang HTML chứa những kết nối (link) khác nhau, nằm ở phía tay trái, cho phép bạn liên lạc với những Web site hữu ích, cho phép bạn tạo dáng và cấu hình của Visual Studio .NET (My Profile link), hoặc mở những dự án hiện hữu (**Open Project**) hoặc khởi động một dự án mới (**New Project**). Trên giữa màn hình Start Page, bạn thấy liệt kê một số dự án mà bạn đã làm việc trong vài ngày qua, nghĩa là những dự án được hiệu đính gần đây nhất.



Hình 14-30: Màn hình khởi đi Start Page

14.3.1 Chọn một loại dự án

Bạn có thể bắt đầu tạo một dự án bằng cách click lên **New Project** trên Start Page, hoặc ra lệnh **File | New | Project**. Cả hai cách đều cho hiện lên khung đối thoại **New Project** (hình 14.9) cho bạn thấy những loại dự án khác nhau mà bạn có thể tạo.

Khung đối thoại này đòi hỏi bạn muốn đoạn mã khuôn giá nào cần được kết sinh, cũng như muốn dùng trình biên dịch nào VB .NET, C# hoặc C++. Thí dụ, ta chọn Console Application và Visual C# Project.

Bạn để ý: Chúng tôi không thể đề cập đến tất cả các loại dự án ỏ đây. Phía C++, thì tất cả các dự án C++ cũ - ứng dụng MFC, dự án ATL v.v.., đều có ở đây. Đối với VB, thì những mục chọn có thay đổi, thí dụ bạn có thể tạo những ứng dụng console, điều mà VB 6 không thể làm được. Bạn có thể tạo những .NET component (Class Library) hoặc .NET Control (Widows Control Library), nhưng bạn không thể tạo những ô control dựa trên COM kiểu xưa.

Vì đây là sách nói về C#, nên chúng tôi liệt kê tất cả các mục chọn có sẵn đối với Visual C# project. Bạn cũng nên để ý là có một số dự án template chuyên hóa hơn nằm ở mục chọn **Other Projects,** trên khung cửa Project Types. bảng 14-06 liệt kê ý nghĩa các mục chọn trên khung cửa Template.

Bảng 14-06: Các mục chọn trên khung Template, khung đối thoại New Project

Nếu bạn chọn…	**Bạn sẽ nhận…**
Windows Application	Một biểu mẫu cơ bản trống rỗng có thể phản ứng trước những tình huống.
Class Library	Một lớp .NET có thể được triệu gọi bởi đoạn mã khác
Windows Control Library	Một lớp .NET có thể được triệu gọi bởi đoạn mã khác, và có một giao diện người sử dụng (GUI), giống như ActiveX control cũ xưa.
Web Applications	Một Web site dựa trên ASP .NET: các trang ASP .NET và các lớp kết sinh ra phản ứng HTML được gởi cho browser từ các trang này.
Web Service	Một lớp C# hoạt động như là một web service trọn vẹn.
Web Control Library	Một ô control mà các trang ASP .NET có thể triệu gọi, để kết sinh một đoạn mã HTML mang dáng dấp một ô control khi được hiển thị trên một browser.
Console Application	Một ứng dụng chạy trên dấu nhắc của dòng lệnh (command line) hoặc trên cửa sổ console.
Windows Services	Một dịch vụ chạy ở hậu trường trên Windows NT hoặc Windows 2000.
Empty Project	Không có gì cả. Bạn phải viết từ zero, nhưng bạn vẫn

Empty Web Project	hưởng những tiện nghi của Visual Studio .NET.
New Project In Existing Folder	Giống như mục chọn Empty Project, nhưng những đặt để biên dịch yêu cầu trình biên dịch kết sinh đoạn mã đối với những trang ASP.NET. Những tập tin dự án đối với một dự án trống rỗng. Bạn dùng mục chọn này khi bạn có những đoạn mã nguồn C# mà bạn muốn chuyển thành những dự án Visual Studio .NET.

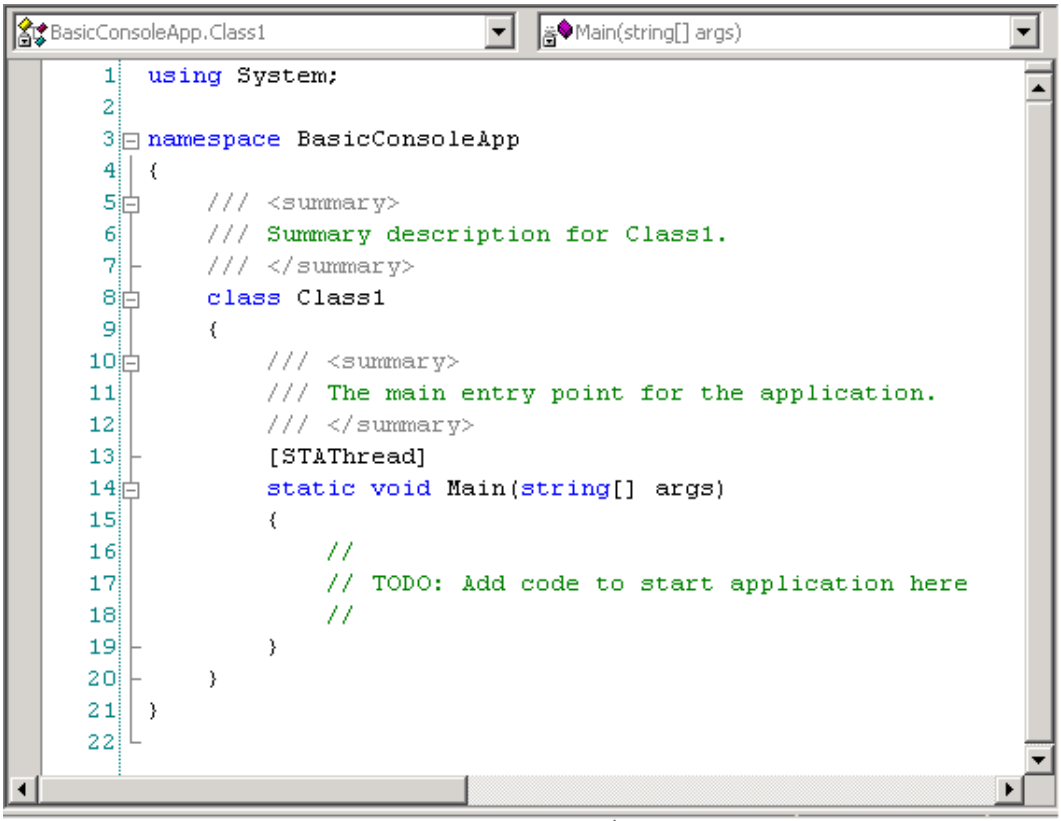
14.3.2 Dự án Console Application

Giả sử bạn chon **Visual C# Projects** và **Console Application**, cho đặt tên dự án là **BasicConsoleApp**, và cho ghi ra thư mục tiếp nhận các tập tin assembly (D;\thien\sach_C#), rồi ấn nút <OK>.

Việc gì sẽ xảy ra. Visual Studio .NET sẽ kết sinh một ứng dụng chạy trên console. Bạn sẽ nhận một số tập tin, bao gồm một tập tin nguồn, **Class1.cs**, chứa khung sườn đoạn mã. Ở đây, chúng tôi sẽ bàn cải đến một ứng dụng console, nhưng cũng những nguyên tắc này sẽ được áp dụng đối với các loại dự án khác hình 14.31, cho thấy đoạn mã mà Visual Studio .NET kết sinh ra.

Như bạn có thể thấy, ta có một chương trình C# chính hiệu, nhưng hiện là vô tích sự, vì ta chưa thổi hồn cho nó. Tuy nhiên, nó chứa đựng một số điều cơ bản đòi hỏi bất cứ chương trình C# nào cũng phải có. Giống như bạn vừa đẻ ra một đứa con sơ sinh, mà khỏi mang nặng đẻ đau. Nó có đầy đủ bộ sậu, nhưng chưa làm gì được ngoài việc khóc oe oe và ỉa đái lung tung. Trên chương trình C# này, bạn có một namespace, **BasicConsoleApp**, và một lớp **Class1** chứa một hàm **Main()** như là điểm đột nhập vào chương trình**.** Đoạn mã này sẵn sàng chờ đợi bạn biên dịch và cho chạy thử. Bạn chỉ cần ấn nút <F5>, hoặc ra lệnh **Debug | Start**. Tuy nhiên, trước khi bạn làm gì đó, ta thử thêm một dòng lệnh - để chương trình bạn làm cái gì đó:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
    Console.WriteLine("Anh em SAMIS xin chao Ban");
}
```



Hình 14-31: Đoạn mã BasicConsoleApp được kết sinh

Nếu bạn cho biên dịch rồi chạy liền (ấn phím <F5>), thì bạn thấy màn hình console (màu đen) hiện lên trong tích tắc (không đủ thời giờ để bạn ngắm nghía "con đầu lòng" của bạn) rồi biến mất. Lý do sự việc diễn ra như thế là vì, Visual Studio .NET nhớ đến những đặt để bạn khai báo khi bạn tạo dự án, được biên dịch như là một ứng dụng console rồi cho chạy liền. Windows sau đó nhận ra rằng cần chạy một ứng dụng console, nhưng lại không có cửa sổ console nào để chạy. Do đó, Windows tạo ra một cửa sổ console rồi chạy chương trình. Khi chương trình vừa thoát, Windows thấy là không cần đến cửa sổ console này nữa, ra lệnh hủy nó đi. Việc này rất lô gic, nhưng lại không giúp ích gì bạn, vì bạn muốn xem mặt mũi "đứa con đầu lòng của bạn"ra sao. Một cách hay nhất để giải quyết vấn đề của bạn là thêm một dòng lệnh **Console.ReadLine()**, theo sau dòng lệnh chào hỏi, nhưng trước khi rời khỏi hàm Main():

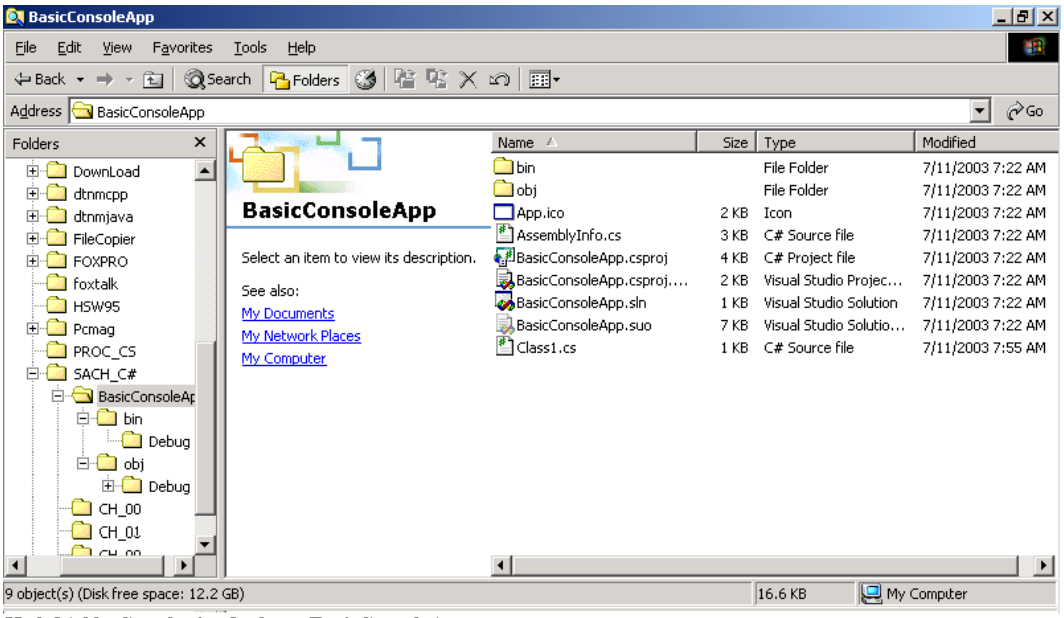
```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
    Console.WriteLine("Anh em SAMIS xin chao Ban");
```

```
        Console.ReadLine();
    }
```

Như vậy, khi cho chạy lại chương trình, nó sẽ cho hiển thị kết xuất rồi đụng câu lệnh **Console.ReadLine();,** nó chờ bạn khỏ phím *return* trước khi chương trình thoát đi. Nghĩa là cửa sổ console cứ trơ mắt chờ bạn ấn phím *return.* Đây là giải pháp duy nhất giải quyết việc ứng dụng console chạy thử nghiệm trên Visual Studio .NET. Còn nếu bạn chạy ứng dụng console trên command line, thì bạn sẽ không gặp trắc trở liên quan đến màn hình biến mất.

14.3.3 Các tập tin khác được tạo ra

Tập tin mã nguồn **Class1.cs** không phải là tập tin duy nhất mà Visual Studio .NET tạo ra. Nếu bạn nhìn vào thư mục D:\THIEN\SACH_C# chẳng hạn, nơi bạn yêu cầu Visual Studio .NET tạo dự án cho bạn, bạn sẽ thấy, như theo hình 14.32, cấu trúc như sau:



Hình 14-32: Các tập tin của dự án BasicConsoleApp

Hai thư mục **bin** và **obj**, có sẵn để chứa các tập tin được biên dịch và trung gian. Thư mục con của **obj** chứa những tập tin tạm thời và trung gian có thể được kết sinh, còn thư mục con của **bin** sẽ chứa những assembly được biên dịch. Các tập tin còn lại chứa những thông tin liên quan đến dự án, như vậy Visual Studio .NET biết sẽ làm gì để biên dịch dự án, cũng như làm thế nào đọc lại lần tới khi bạn cho mở dự án.

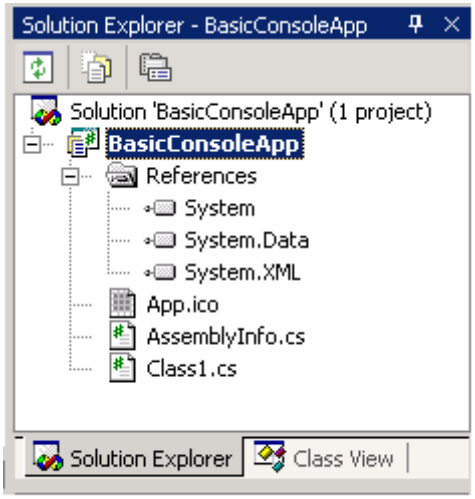
14.3.4 Solutions và Projects

Điểm quan trọng ở đây là phân biệt giữa một dự án (project) và một giải pháp (solution):

- Một **dự án** là một tập hợp tất cả các tập tin nguồn và nguồn lực (resource) sẽ được biên dịch thành một assembly đơn lẻ (hoặc trong vài trường hợp, một đơn thể đơn lẻ - single module). Thí dụ, một dự án có thể là một thư viện lớp (class library) hoặc một ứng dụng Windows GUI.

- Một **giải pháp** là một tập hợp những dự án tạo thành một package phần mềm đặc biệt (ứng dụng).

Muốn thấy sự khác biệt, chúng tôi thử lấy thí dụ khi bạn phân phối một ứng dụng cho người sử dụng, có thể nó gồm nhiều assembly hơn là chỉ là một. Thí dụ, có thể là một giao diện người sử dụng; nó sẽ có vài ô control "cây nhà lá vườn" và những cấu kiện khác được gỡi đi như là những thư viện thuộc thành phần ứng dụng. Kể cả việc có thể có một giao diện khác dành cho những người quản lý. Mỗi một những thành phần này có thể được chứa thành những assembly riêng rẽ, và do đó đối với Visual Studio .NET như là một dự án riêng biệt. Tuy nhiên, có thể là bạn thực hiện những dự án này song hành và phối hợp với nhau giữa các dự án. Do đó, xem ra tiện lợi khi ta có khả năng hiệu đính chúng như là một đơn vị duy nhất trong Visual Studio .NET. Visual Studio .NET cho phép điều này bằng cách xem tất cả các dự án như là một giải pháp (solution), và đối xử solution như là một đơn vị được đọc vào và cho phép bạn làm việc trên ấy.



Hình 14-33: Solution Explorer

Mãi tới giờ ta nói đến việc tạo một dự án console. Thực thế, trong thí dụ kể trên, Visual Studio .NET đã tạo ra một solution cho chúng ta - một solution đặc biệt chỉ chứa duy nhất một dự án. Bạn có thể thấy tình trạng này trên một cửa sổ trên Visual Studio .NET được gọi là Solution Explorer, chứa một cấu trúc cây định nghĩa giải pháp của bạn, hình 14.33:

Hình này cho thấy dự án chứa tập tin nguồn **Class1.cs**, mà còn một tập tin nguồn khác, **AssemblyInfo.cs**, chứa thông tin liên quan đến việc biên dịch dự án. Trên cửa sổ này bạn còn thấy những assembly khác mà dự án qui chiếu về (References).

Nếu bạn không thay đổi những đặt để mặc nhiên trên Visual Studio .NET, thì cửa sổ **Solution Explorer** sẽ xuất hiện ở góc phải trên đầu màn hình của bạn. Nếu bạn không thấy, thì có thể gọi nó vào bằng cách ấn phím <Ctrl+Alt+L> hoặc ra lệnh **View | Solution Explorer**.

Tập tin solution mang phần đuôi là .sln, (tắt chữ solution) trong trường hợp của chúng ta là **Basic ConsoleApp.sln**. Dự án được mô tả bởi nhiều tập tin khác nhau trên thư mục chính của dự án. Nếu bạn cất công hiệu đính những tập tin này sử dụng Notepad chẳng hạn, bạn sẽ thấy chúng toàn là những tập tin plain text – và theo nguyên tắc của .NET cũng như công cụ .NET là dựa trên chuẩn mở nghĩa là dựa trên dạng thức XML.

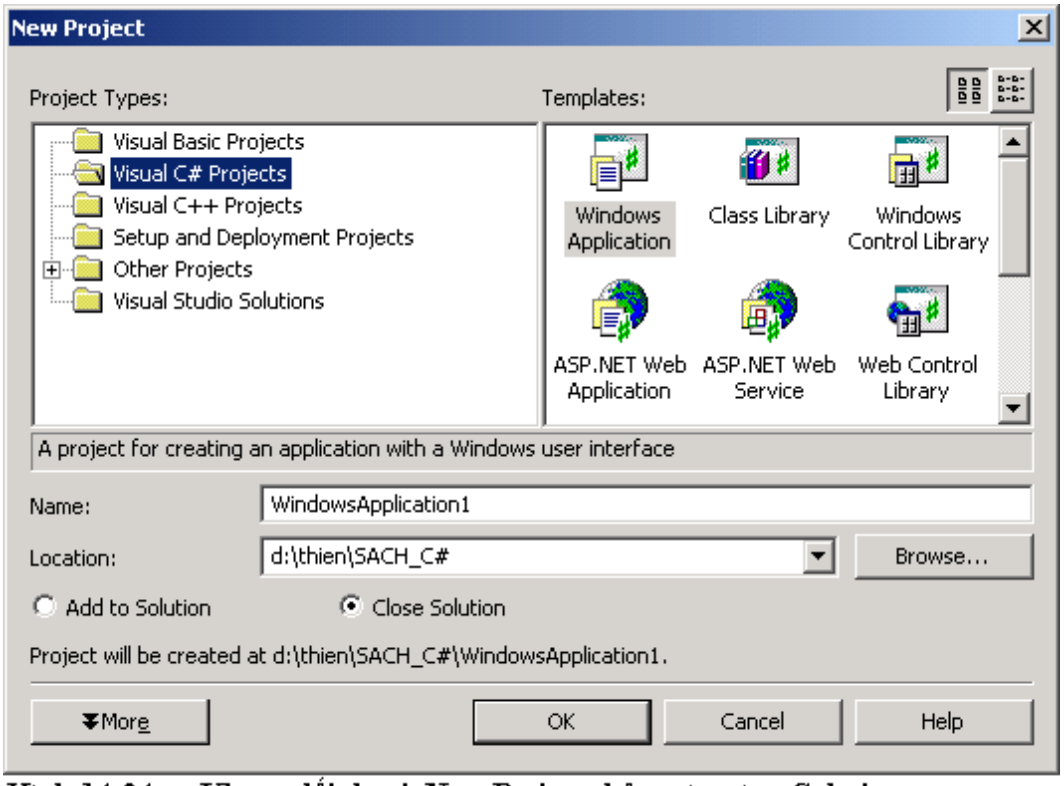
14.3.5 Thêm một dự án khác lên giải pháp

Chúng tôi muốn cho bạn thấy Visual Studio .NET hoạt động thế nào với ứng dụng Windows cũng như với ứng dụng console.

Chúng tôi sẽ tạo một dự án Windows mang tên **BasicWindows**, nhưng thay vì đưa nó vào một solution mới, ta yêu cầu Visual Studio .NET đưa nó vào solution hiện hành **BasicConsoleApp**. Nghĩa là ta sẽ có một solution với một ứng dụng Windows và một ứng dụng console. Đấy là cách tạo một trình tiện ích mà bạn muốn cho chạy trên Windows hoặc chạy trên dòng lệnh command line.

Có hai cách để thực hiện điều này. Cách thứ nhất là right-click lên tên của solution trên cửa sổ **Solution Explorer**, để cho hiện lên trình đơn shortcut rồi chọn **Add | New Project**. Cách thứ hai là ra lệnh **File | New | New Project**. Cả hai cách đều cho hiện lên khung đối thoại **New Project** như ta đã thấy. Hình 14.34

Nếu ta ra lệnh **File | New | Projects**, thì khung đối thoại **New Project** lần này có hai nút radio button: **Add To Solution** và **Close Solution** cho phép bạn hoặc tạo một solution mới đối với dự án hoặc thêm vào solution hiện hữu. Nếu ta chọn nút **Add to Solution**, ta sẽ có một dự án mới và solution bây giờ chứa một ứng dụng console và một ứng dụng Windows. Lẽ dĩ nhiên trong trường hợp này, tên cũ **BasicConsoleApp** không còn thích hợp nữa. Ta có thể cho thay đổi bằng cách right-click lên tên của solution rồi chọn mục **Rename** trên trình đơn shortcut. Thí dụ, ta đổi thành **DemoSolution** chẳng hạn. Solution Explorer giờ đây như sau: (hình 14.35).



Hình 14-34: Khung đối thoại New Project thêm vào cùng Solution

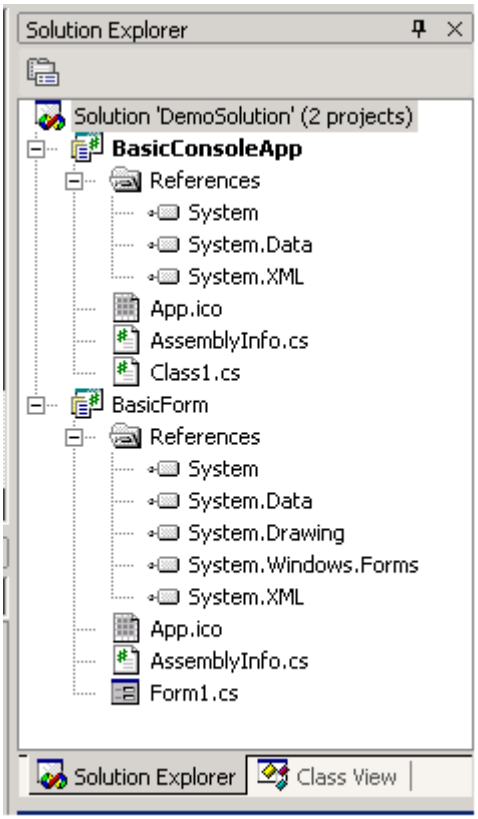
Trên cửa sổ Solution Explorer này ta có thể thấy Visual Studio .NET tự động đưa một số qui chiếu về các lớp cơ bản quan trọng đối với chức năng của biểu mẫu Windows nằm trong các namespace System.Drawing, System.Windows.Forms.

Ngoài ra, bạn cũng sẽ thấy là tập tin solution cũng đổi thành **DemoSolution.sln.** Nói chung, nếu bạn muốn thay đổi bất cứ tên tập tin nào, thì Solution Explorer là nơi tốt nhất để tiến hành những thay đổi, vì Visual Studio .NET sẽ tự động cho nhật tu bất cứ những qui chiếu nào về tập tin trong tất cả các tập tin thuộc dự án. Bạn chớ nên sử dụng Window Explorer để thay đổi các tập tin dự án, vì bạn sẽ phá vỡ solution.

14.3.5.1 Cho đặt để một Startup Project

Một điểm bạn nên nhớ là cho dù bạn có nhiều dự án trong cùng một solution, thì chỉ một trong những dự án này chạy mà thôi trong một lúc. Khi bạn cho biên dịch một solution, thì tất cả các dự án trong solution sẽ được biên dịch. Tuy nhiên, bạn phải khai báo cho Visual Studio .NET biết dự án nào sẽ bắt đầu chạy khi bạn muốn gỡ rối chương trình. Nếu bạn có một EXE và nhiều thư viện mà EXE sẽ triệu gọi, thì đương nhiên EXE

sẽ là dự án khởi động. Trong trường hợp của chúng ta, ta có hai EXE độc lập, ta chỉ cần lần lượt gỡ rối từng dự án một.



Hình 14-35: Solution Explorer mới

Bạn có thể bảo Visual Studio .NET dự án nào phải chạy trước, bằng cách right-click lên tên solution để cho hiện lên trình đơn shortcut, rồi bạn chọn click mục **Set Startup Project**…để cho hiện lên khung đối thoại **Solution "DemoSolution" Property Pages**. Bạn có thể cho biết dự án Startup hiện hành, vì nó sẽ là dự án hiện lên in đậm trên cửa sổ Solution Explorer - ở đây là **BasicConsole App**.

14.3.6 Đoạn mã ứng dụng Windows

Một ứng dụng Windows chứa đoạn mã khởi động phức tạp hơn nhiều so với một ứng dụng chạy trên console, vì tạo một cửa sổ là một tiến trình phức tạp. Chúng tôi sẽ không đề cập chi tiết đến đoạn mã của một ứng dụng Windows. Sẽ có một chương dành cho vấn đề này. Trong tạm thời, chúng tôi cho in ra ở đây bảng liệt in (listing) đoạn mã kết sinh bởi Visual Studio .NET đối với dự án **BasicForm**. Bạn để ý lớp ở đây được gọi là **Form1**, tượng trưng cho cửa sổ chính.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace BasicForm
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1: System.Windows.Forms.Form
    {
        /// <summary>
```

```
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public Form1()
        {   //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent
            // call
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            if(disposing)
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
        #endregion

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

14.3.7 Đọc vào các dự án Visual Studio 6

Nếu bạn lập trình theo C#, rõ ràng là bạn không cần đọc đến những dự án cũ xưa viết trên Visual Studio 6, vì C# không có trên Visual Studio 6. Tuy nhiên, liên thông ngôn ngữ (language interoperability) là phần chủ chốt của .NET Framework, do đó có thể bạn muốn đoạn mã C# của bạn làm việc tay trong tay với VB .NET hoặc C++. Trong tình trạng như thế, có thể bạn cần hiệu đính những dự án được tạo ra trong Visual Studio 6.

Khi đọc vào những dự án và workspace viết theo Visual Studio 6, Visual Studio .NET sẽ cho nâng cấp lên thành những solution Visual Studio .NET. Tình trạng lại khác so với các dự án C++, VB hoặc J++:

- Trên C++, không cần thiết thay đổi đối với mã nguồn. Tất cả các chương trình C++ cũ xưa sẽ còn hoạt động tốt với trình biên dịch C++ mới. Rõ ràng là không phải đoạn mã được giám quản (managed code), nhưng vẫn biên dịch chạy ngoài .NET Runtime. Nếu bạn muốn đoạn mã này hội nhập vào .NET Framework, thì bạn cần hiệu đính lại. Nếu bạn yêu cầu Visual Studio .NET đọc vào một dự án cũ xưa C++, thì nó đơn giản thêm vào một tập tin solution mới và nhật tu những tập tin dự án. Nó để yên các tập tin .dsw và .dsp không thay đổi để dự án có thể hiệu đính bởi Visual Studio 6 nếu thấy cần thiết.

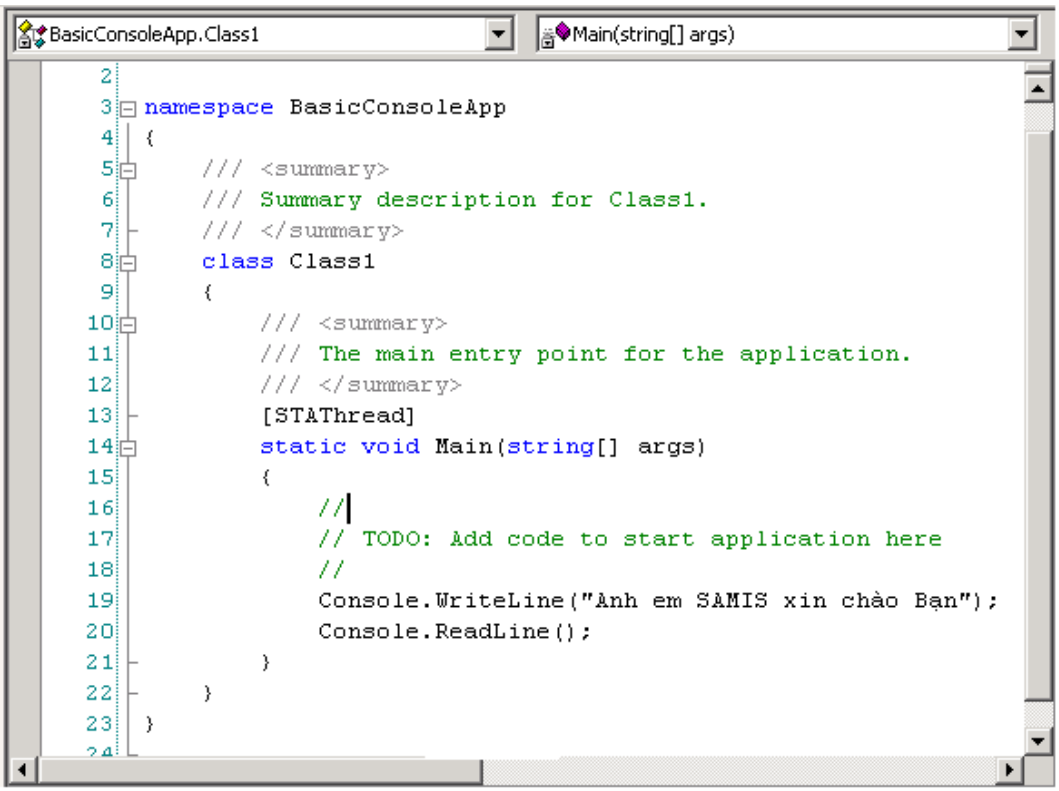
- Đối với Visual Basic thì có vấn đề, vì VB đã bị thay thế bởi VB .NET. Mặc dù VB .NET được thiết kế xoay quanh VB 6 và chia sẻ cùng cú pháp, nhưng trong thực tế VB .NET hoàn toàn là một ngôn ngữ mới. Trên VB 6, mã nguồn phần lớn bao gồm những trình thụ lý tình huống (event handler) đối với những ô control. Đoạn mã hiện lo việc hiển lộ cửa sổ chính và phần lớn những ô control trên ấy, không phải là thành phần của VB, nhưng lại nằm ẩn ở hậu trường như là thành phần cấu hình của dự án. Ngược lại, VB .NET hoạt động giống như C#, bằng cách trưng ra toàn bộ chương trình mở toang như là mã nguồn, do đó tất cả đoạn mã sẽ hiển thị cửa sổ chính và tất cả các ô control trên ấy, đều nằm trong tập tin mã nguồn. Giống như C#, VB .NET đòi hỏi mọi việc phải thiên đối tượng và thuộc lớp, trong khi ấy VB 6 không công nhận khái niệm về lớp như đúng ý nghĩa của nó. Nếu bạn cố thử đọc một dự án VB6 với Visual Studio .NET nó sẽ "nâng cấp" toàn bộ mã nguồn theo Visual Studio .NET, và như vậy sẽ có nhiều thay đổi đối với mã nguồn của VB6. Visual Studio .NET cũng có thể tự động thực hiện những thay đổi này và tạo một solution VB .NET mới và lúc này nó sẽ khác nhiều so với mã nguồn VB6 bị chuyển đổi và bạn sẽ phải kiểm tra kỷ bảo đảm đoạn mã kết sinh hoạt động đúng đắn. Có nhiều khúc trên đoạn mã Visual Studio .NET còn ghi chú những chú giải khi nó không biết sẽ làm gì với đoạn mã này, và buộc lòng bạn phải hiệu đính bằng tay.

14.4 Khảo sát và thảo đoạn mã một dự án

Trong phần này, chúng tôi sẽ khảo sát những chức năng Visual Studio .NET cho phép bạn thêm đoạn mã vào dự án.

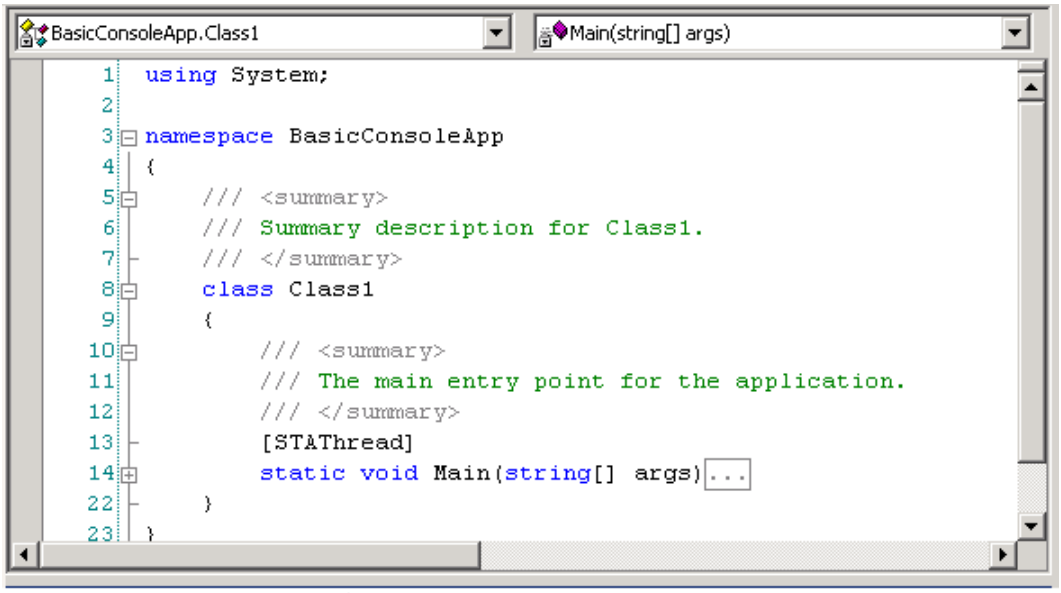
14.4.1 Folding Editor

Điểm khá lý thú trên Visual Studio .NET là việc sử dụng một folding editor như là code editor mặc nhiên. Bạn thử nhìn xem hình 14.36 sau đây, cho thấy đoạn mã kết sinh bởi ứng dụng console.



Hình 14-36: Đoạn mã kết sinh của BasicConsoleApp

Bạn có nhận thấy ở phía tay trái màn hình, những ô vuông nhỏ có dấu trừ (-) ở trong; Những ký hiệu này cho biết một khối lệnh bắt đầu. Bạn có thể click lên ký hiệu này, nó biến thành dấu cộng (+), và khối lệnh teo lại với một ô hình chữ nhật với 3 dấu chấm. Cách hoạt động này, được gọi là *outlining*, giống như trên cây thư mục. Bạn xem hình 14.37.

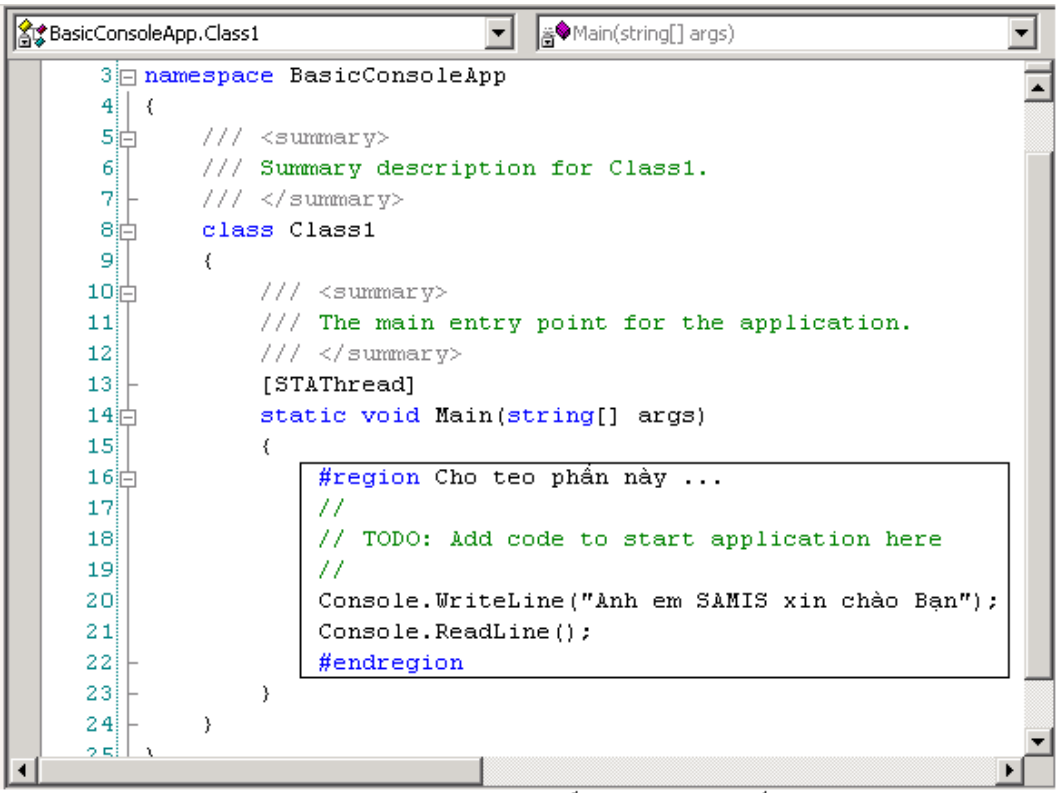


Hình 14-37: Teo lại một khối lệnh

Đây có nghĩa là khi bạn đang hiệu đính, bạn có thể tập trung chỉ vào vùng nào bạn quan tâm, cho đóng lại (bằng cách click lên ô có dấu trừ) những phần đoạn mã nào bạn không quan tâm đến. Không những chỉ thế, nhưng nếu bạn không hài lòng cách editor chọn phân thành khối lệnh, bạn có thể cho biết một cách khác, bằng cách sử dụng những chỉ thị tiền xử lý C# (C# preprocessor), chẳng hạn **#region** và **#endregion** mà chúng tôi đã đề cập ở chương 4, "Căn bản ngôn ngữ C#", mục 4.8.4. Thí dụ, ta quyết định ta muốn có khả năng cho teo lại đoạn mã nằm trong hàm hành sự **Main()**, ta chèn thêm #region và #endregion như theo hình 14.38.

Code editor sẽ tự động phát hiện khối #region, và cho đặt dấu trừ ngay hàng có chỉ thị #region. Cho bao một khối lệnh nằm giữa cặp chỉ thị **#region** và **#endregion** có nghĩa là ta muốn cho teo lại tùy ý khối lệnh này, như theo hình 14.39. Bạn thấy dấu cộng với chú giải ta thêm vào.

Ngoài chức năng folding editor, trình soạn thảo văn bản Visual Studio .NET còn đem lại một chức năng khác khá lý thú khi bạn đang khỏ vào lệnh C#, đó là **Intellisense**. Thí dụ, nếu bạn khỏ vào tên của một thể hiện lớp theo sau bởi dấu chấm (.), Intellisense sẽ cho hiện lên một ô liệt kê nhỏ cho phép bạn chọn một trong những thành viên lớp để đưa vào lệnh. (bạn có thể cho hiện lên ô liệt kê này bằng cách ấn <Ctrl+Space>). Khi bạn khỏ vào dấu ngoặc nghéo mở '{' thì Intellisense sẽ cho hiện lên danh sách các thông số. Điều này không những giúp bạn giảm thiểu việc khỏ phím, nhưng còn giúp bạn nhận đúng thông số.

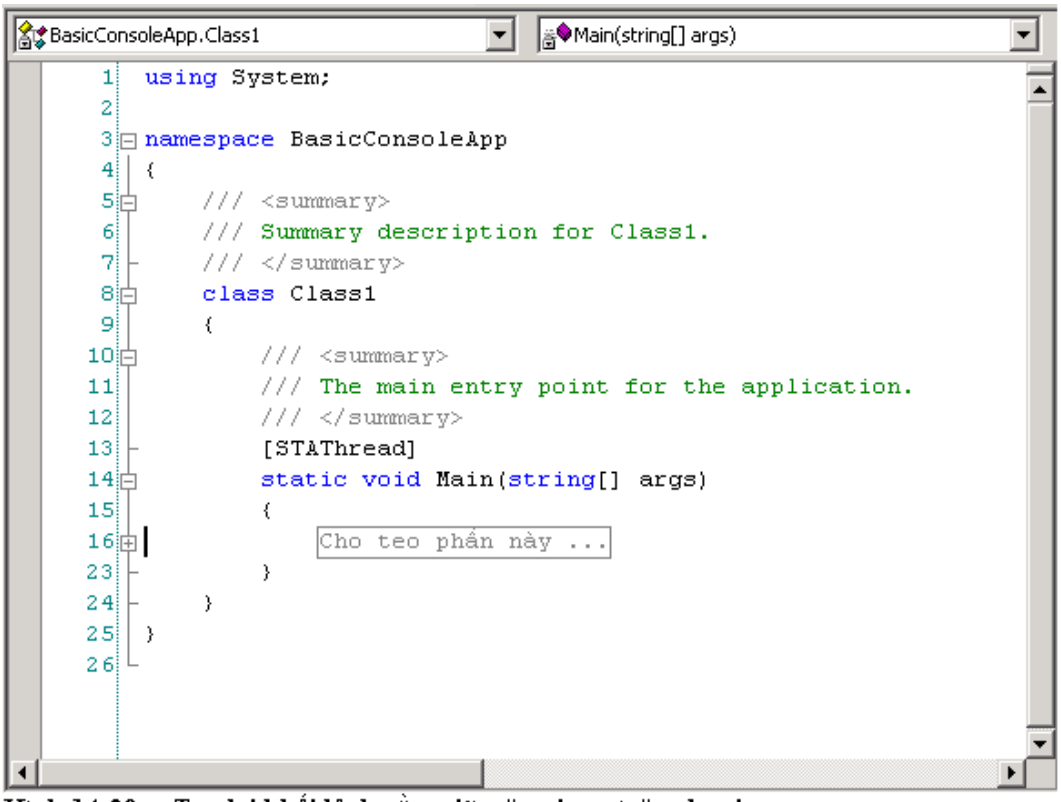


Hình 14-38: Thêm #region và #endregion chuẩn bị cho teo khối lệnh

Code editor còn tiến hành một số kiểm tra cú pháp các câu lệnh bạn khỏ vào.

14.4.2 Các cửa sổ khác

Ngoài code editor,Visual Studio .NET còn cung cấp một số cửa sổ khác giúp bạn nhìn xem dự án theo nhiều góc độ khác nhau. Nếu những cửa sổ được mô tả ở đây không hiện lên trong setup của Visual Studio .NET, bạn có thể tìm đến trình đơn **View** rồi click cửa sổ nào bạn quan tâm đến. Ngoại lệ duy nhất đối với điều này là *design view* và *code editor*, vì chúng được xem như là 2 trang tab trên một cửa sổ. Bạn có thể cho hiển thị một trong hai cửa sổ này hoặc **View Designer** hoặc **Code View**, bằng cách click lên hai nút thanh công cụ nằm trên đầu cửa sổ Solution Explorer, hoặc right click lên tên tập tin trên cửa sổ **Solution Explore**r, rồi chọn mục **View Designer** hoặc **View Code** trên trình đơn shortcut.



Hình 14-39: Teo lại khối lệnh nằm giữa #region và #endregion

14.4.2.1 Cửa sổ Design View

Khi bạn thiết kế một ứng dụng Windows (hoặc một ứng dụng ASP.NET), một cửa sổ mà bạn sẽ sử dụng nhiều nhất là **Design View**. Cửa sổ cho phép bạn hình dung toàn bộ mặt mũi ứng dụng của bạn có thể nhìn thấy được (chúng tôi gọi là "nhìn tận mắt bắt tận tay"). Thông thường, bạn sử dụng phối hợp cửa sổ **Design View** với cửa sổ **Toolbox**, cửa sổ hộp đồ nghề. **Toolbox** chứa vô số cấu kiện .NET (gọi là ô control) mà bạn có thể lôi thả (drag-and-drop) dễ dàng lên mặt bằng biểu mẫu của bạn.

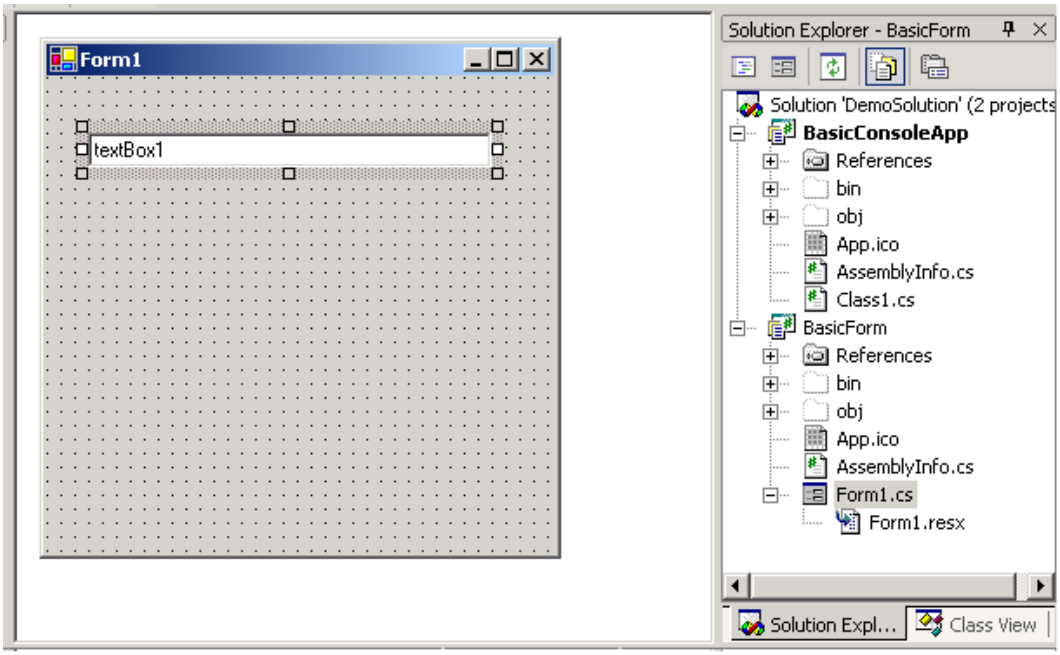
.

Chúng tôi đã đề cập **Toolbox** ở mục 14.1.17 trong chương này. Bạn có thể xem lại. Ở đây chúng tôi chỉ bổ sung vài điều.

Bạn có thể thêm riêng của bạn một loại ô control nào đó, gọi là custom control, vào toolbox, bằng cách right-click lên bất cứ loại nào trên toolbox rồi chọn mục **Add Tab** từ trình đơn shortcut. Bạn có thể thêm công cụ khác vào toolbox bằng cách chọn mục **Customize Toolbox** cũng trên trình đơn shortcut này. Điều này rất hữu ích khi bạn muốn thêm những cấu kiện COM hoặc ActiveX control không có trên toolbox theo mặc nhiên.

Nếu bạn muốn thêm một cấu kiện COM lên dự án bạn có thể click cấu kiện này lôi thả lên biểu mẫu giống như bạn đã làm với các ô control .NET. Lúc này Visual Studio .NET sẽ tự động thêm đoạn mã liên thông COM cần thiết cho phép dự án có thể triệu gọi hàm ô control COM.

Bây giờ bạn muốn thấy toolbox hoạt động thế nào. Bạn muốn đặt một ô text box lên dự án **BasicForm. Bạn** cho hiện lên Toolbox (icon của nó là hình búa kềm) bằng cách click lên icon hoặc ấn tổ hợp phím <Ctrl+Alt+X>, hoặc ra lệnh **View | Toolbox**. Rồi bạn click tiếp lên biểu mẫu đang ở chế độ Design. Bây giờ biểu mẫu mang dáng dấp như sau, hình 14.40.



Hình 14-40: Thêm một ô Textbox

Tuy nhiên, điểm lý thú là nếu ta nhìn vào đoạn mã, ta thấy IDE đã thêm đoạn mã lo hiển lộ (instantiate) một đối tượng textbox được đặt lên biểu mẫu. Có thêm một biến thành viên trên lớp Form1:

```
public class Form1: System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox textBox1;
```

Ngoài ra, một đoạn mã được thêm vào hàm hành sự **InitializeComponent()**, khởi gán cấu kiện. Hàm này sẽ được triệu gọi bởi hàm constructor của Form1:

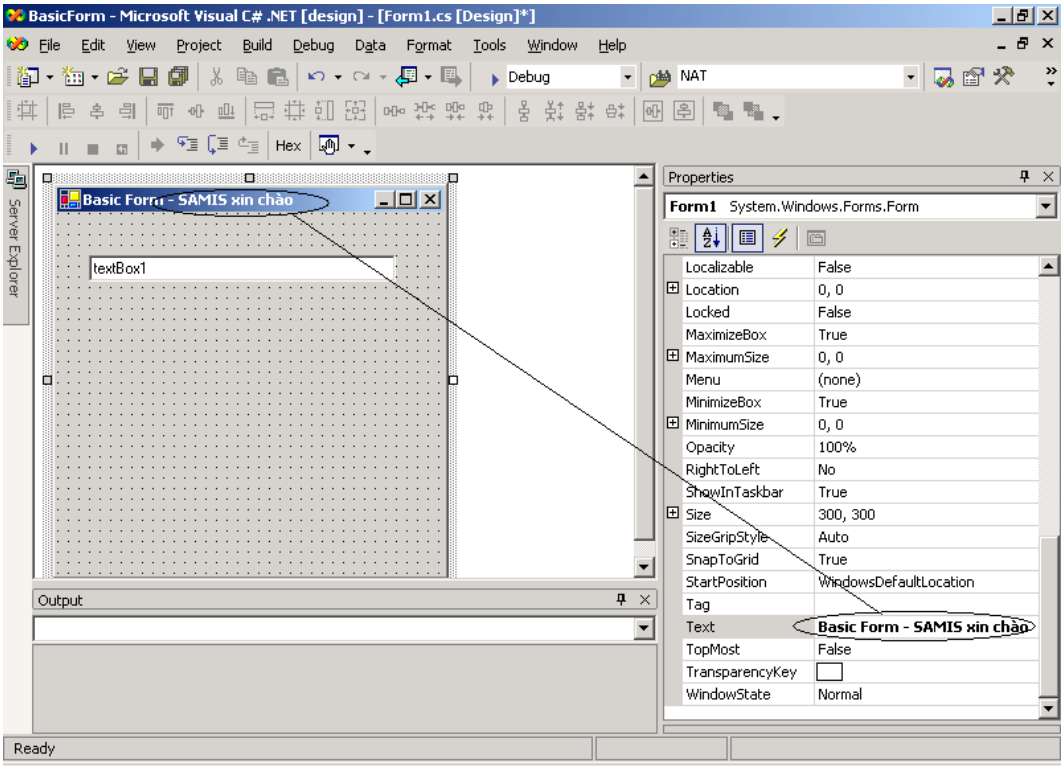
```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>

private void InitializeComponent()
{
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(24, 32);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(232, 20);
    this.textBox1.TabIndex = 0;
    this.textBox1.Text = "textBox1";
```

Nếu muốn, bạn cũng có thể thêm bằng tay đoạn mã như trên vào tập tin mã nguồn, và Visual Studio .NET sẽ phát hiện ra đoạn mã của bạn và cho hiện lên ô control tương ứng. Tuy nhiên, tốt hơn là thêm bằng mắt những ô control trên rồi để cho Visual Studio .NET lo phần còn lại là thêm những đoạn mã thích hợp. Việc click vào toolbox rồi click tiếp lên biểu mẫu, điều chỉnh vị trí và kích thước các ô control này xem ra là nhanh hơn là suy nghĩ và khỏ vào các lệnh.

Một lý do là nên thêm bằng mắt các ô control là vì Visual Studio .NET yêu cầu những đoạn mã thêm vào bằng tay phải tuân thủ một số qui tắc, mà có thể bạn không tuân thủ. Đặc biệt, bạn để ý trên hàm hành sự **InitializeComponent()** lo khởi gán ô textbox có dòng chú giải ở đầu cảnh cáo bạn là chớ thay đổi. Nói thế, nhưng nếu cẩn thận, bạn cũng có thể hiệu đính chẳng hạn trị của vài thuộc tính như vậy ô control có thể hiển thị khác đi. Nói tóm lại, muốn tiến hành những thay đổi như thế, bạn phải có kinh nghiệm dày dạn và phải cẩn thận hiểu mình muốn làm gì.

14.4.2.2 Cửa sổ Properties

Có một cửa sổ khác, được gọi là **Properties** window, xuất xứ từ IDE của VB. Ta đã biết là trên các lớp đều có thiết đặt những thuộc tính (property). Những lớp cơ bản của .NET tượng trưng cho biểu mẫu và những ô control đều có khá nhiều thuộc tính định nghĩa những hành động hoặc dáng dấp - chẳng hạn thuộc tính **Width** (chiều rộng), **Height** (chiều cao), **Enabled** (cho biết người sử dụng có thể khỏ nhập liệu hay không), **Text** (văn bản hiển thị bởi ô control) v.v.. Thông qua cửa sổ **Properties** bạn có thể hiệu đính những trị ban đầu của một số lớn thuộc tính, đối với những ô control mà Visual Studio .NET có khả năng phát hiện bằng cách đọc mã nguồn của bạn. Xem hình 14.41.



Hình 14-41: Cửa sổ Properties Window

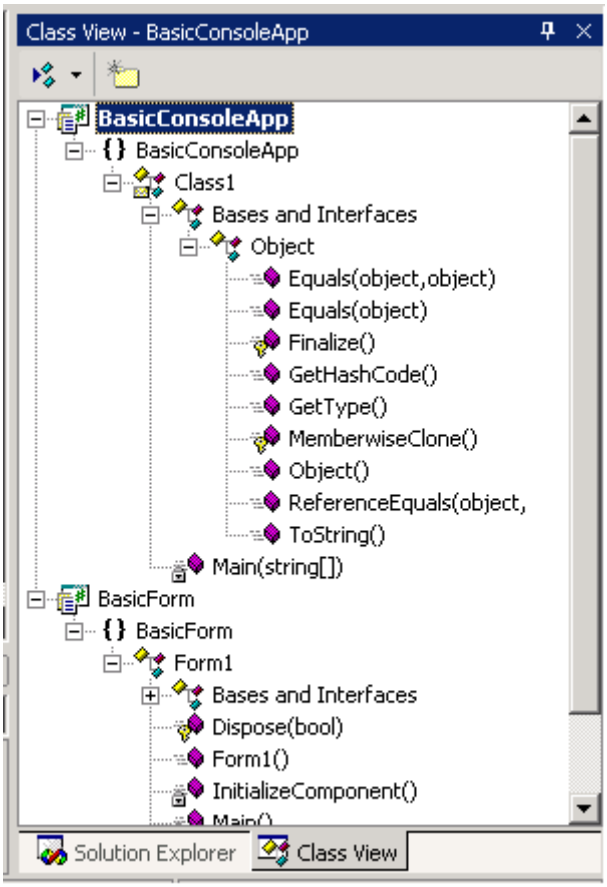
Cửa sổ Properties Windows cũng có thể cho thấy những events. Bạn có thể nhìn thấy events bằng cách click lên icon thứ 4 từ trái qua phải (hình sấm chớp) ở trên đầu cửa sổ.

Trên đầu cửa sổ **Properties Windows** là một ô list box cho phép bạn chọn ô control nào để quan sát hoặc hiệu đính. Chúng tôi chọn biểu mẫu Form1, biểu mẫu chính trong dự án BasicForm, và cho hiệu đính thuộc tính Text "Basic Form – SAMIS xin chào". Khi thay đổi thuộc tính Text như thế thì tiêu đề của biểu mẫu cũng thay đổi theo luôn và trong đoạn mã nguồn cũng thế. Nếu bạn cho kiểm tra lại mã nguồn, bạn có thể thấy hiện chúng ta đã hiệu đính mã nguồn, thông qua giao diện thân thiện hơn:

```
//
// Form1
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(292, 273);
this.Controls.AddRange(new System.Windows.Forms.Control[] {

this.textBox1});
this.Name = "Form1";
this.Text = "Basic Form - SAMIS xin chào";
this.ResumeLayout(false);
```

Không phải tất cả các thuộc tính mà bạn thấy trên cửa sổ Properties Windows sẽ được ghi rõ ra trên mã nguồn. Đối với những thuộc tính nào không thấy trên mã nguồn, Visual Studio .NET sẽ hiển thị những trị mặc nhiên được đặt để khi biểu mẫu được tạo ra, và sẽ được đặt để khi biểu mẫu hiện được khởi gán. Rõ ràng là khi bạn thay đổi trị của một thuộc tính nào đó trên cửa sổ Properties Window, thì một câu lệnh đặt để rõ ra thuộc tính này sẽ xuất hiện lên mã nguồn của bạn, và ngược lại.

Cửa sổ **Properties Windows** được xem như là cách tiện lợi nhất để có cái nhìn về tất cả các thuộc tính của một ô control nào đó. Ở cuối cửa sổ **Properties Windows** có một vùng dùng giải thích ngắn gọn ý nghĩa thuộc tính mà bạn chọn xem. Bạn tha hồ mà tìm hiểu. Bạn cho ngời sáng một thuộc tính nào đó, bạn sẽ thấy giải thích ở cuối cửa sổ; nếu chưa hiểu kỹ, bạn có thể ấn phím <F1> Help thì cửa sổ Help liên quan đến thuộc tính này sẽ được hiện lên cho bạn tha hồ nghiền ngẫm.



Hình 14-42: Class View

14.4.2.3 Cửa sổ Class View

Khác với cửa sổ **Properties Window**, cửa sổ **Class View** xuất xứ từ môi trường triển khai Visual C++. Hình 14.42. Trên Visual Studio .NET, cửa sổ **Class View** được xem như là trang tab của cửa sổ **Solution Explorer**, cho thấy cây đẳng cấp của các namespace và các lớp trong đoạn mã của bạn. Nó cho bạn một tree view mà bạn có thể cho bung ra (bằng cách click lên các ô có dấu cộng) để xem namespace nào chứa những lớp nào, và trong lớp chứa những thành viên nào

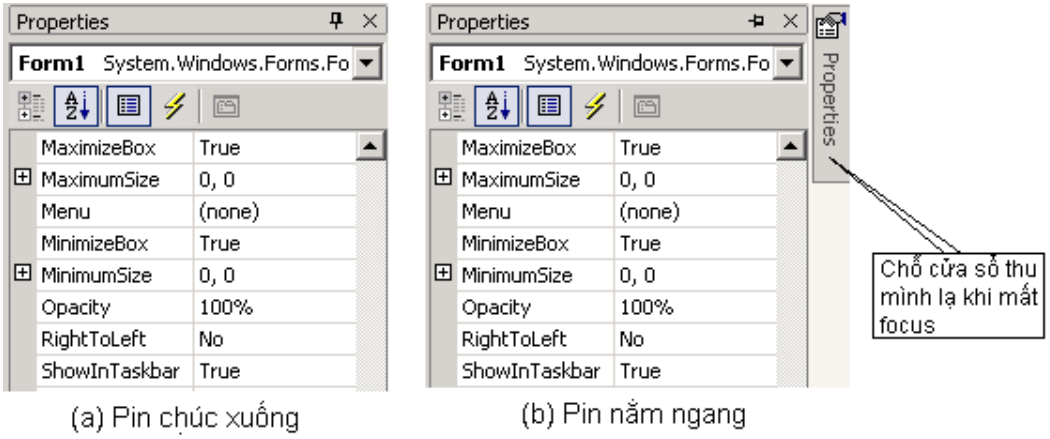
Một chức năng "ngon lành" của cửa sổ **Class View** là nếu bạn right click lên tên của bất cứ mục tin (item) nào bạn có thể thâm nhập vào đoạn mã nguồn, bạn chọn mục **Go To Definition**, trên trình đơn shortcut, thì trên code editor sẽ hiện lên đoạn mã định nghĩa mục tin này. Hoặc bạn double click lên mục tin

này thì cũng đi đến kết quả như trên.

Nếu bạn right click lên **Form1** trên cửa sổ **Class View**, thì trình đơn shortcut hiện lên. Bạn click mục chọn **Add**, thì bạn thấy xuất hiện 4 mục chọn phụ: **Add Method, Add Property, Add Indexe**r hoặc **Add Field**, cho phép bạn thêm các hàm hành sự, thuộc tính, indexer hoặc vùng mục tin (field). Có nghĩa là khi bạn chọn một trong những mục chọn phụ kể trên thì một khung đối thoại thích ứng sẽ hiện lên để bạn thêm những thành viên lớp Form1 kể trên. Điều này sẽ giúp ích rất nhiều đối với bạn.

14.4.3 Pin Buttons

Khi khảo sát Visual Studio .NET, có thể bạn nhận thấy có nhiều cửa sổ mà chúng tôi mô tả có vài chức năng mang dáng dấp của những thanh công cụ (toolbar). Đặc biệt, ngoại trừ code editor, thì các cửa sổ có thể "dời bến" được neo đâu đó, gọi là docking. Có một icon, giống như đầu kim cúc (pin) cạnh icon nút minimize button nằm ở góc phải phía trên mỗi cửa sổ. Icon này khi chúc đầu xuống, thì cửa sổ hoạt động như một cửa sổ bình thường mà bạn đã quen. Nhưng khi icon này nằm ngang, thì cửa sổ này chỉ hiện lên khi nó có focus.Nhưng khi nó mất focus, nghĩa là con trỏ chỉ vào chỗ khác, thì cửa sổ biến mất, thu mình lại qua phía tay phải hoặc tay trái. Bạn xem hình 14.43:



Hình 14-43: Pin button

14.5 Xây dựng một dự án

Trong phần này, chúng tôi sẽ xét qua những lựa chọn mà Visual Studio .NET cho phép ta có được để xây dựng một dự án.

14.5.1 Building, Compiling và Making

Trước khi ta đi sâu vào việc xem xét những lựa chọn khác nhau liên quan đến việc xây dựng (building) một dự án, thiết tưởng ta nên làm sáng tỏ những từ ngữ liên quan đến tiến trình biến mã nguồn của bạn thành một loại đoạn mã khả thi. Bạn thường biết đến 3 từ: *compiling* (biên dịch), *building* (xây dựng) và *making* (làm thành). Nguyên ủy của những từ ngữ khác nhau là do việc gần đây chuyển một mã nguồn thành một mã khả thi đòi hỏi không chỉ một bước mà thôi. Lý do là vì phần lớn, một chương trình có thể chứa nhiều tập tin nguồn. Trên C++ chẳng hạn mỗi tập tin nguồn phải được biên dịch riêng rẽ, cho ra một *tập tin đối tượng*[54] (object file), mang dáng dấp đoạn mã khả thi (executable, EXE), nhưng mỗi tập tin đối tượng chỉ liên hệ với một tập tin nguồn. Muốn cho ra một mã kết sinh, các tập tin đối tượng này phải được liên kết (linked) lại, một tiến trình mà người ta gọi là *linking*. Tiến trình phối hợp thường được gọi là *xây dựng* (building) đoạn mã theo từ ngữ Windows. Tuy nhiên, theo từ ngữ C#, thì trình biên dịch phức tạp hơn nhiều có khả năng đọc và xử lý tất cả các tập tin nguồn như là một khối. Do đó, đối với C# không có giai đoạn liên kết riêng rẽ, nên trong phạm trù C# các từ *compiling* và *building* coi như là giống nhau, có thể dùng từ này hay từ kia, thay thế nhau, đều được cả.

Còn từ "making" cũng có nghĩa là xây dựng, nhưng nó không được dùng trong phạm trù C#. Từ này xuất xứ từ những máy tính mainframe (máy tính lớn), cho biết khi một dự án gồm nhiều tập tin nguồn, thì có một tập tin riêng rẽ được viết ra chứa những chỉ thị dành cho trình biên dịch biết cách xây dựng một dự án - những tập tin nào phải cho bao gồm và thư viện nào phải kết nối v.v.. Tập tin riêng rẽ này được gọi là một *make file*, do đó từ này tiếp tục được sử dụng như là chuẩn trên Unix hoặc Linux. Các tập tin make thường không cần đến trên Windows, mặc dù bạn có thể viết chúng hoặc yêu cầu Visual Studio .NET kết sinh chúng nếu bạn muốn.

14.5.2 Debug Build và Release Build

Khi bạn đang gỡ rối, thường bạn muốn chương trình khả thi có một cách ứng xử khác đi khi bạn cho phân phối chương trình. Khi phân phối chương trình, ngoại trừ việc chương trình phải chạy tốt không có lỗi, nó còn phải có kích thước càng nhỏ càng tốt và còn phải chạy thật nhanh. Rất tiếc là những đòi hỏi trên không hợp với nhu cầu gỡ rối chương trình, với nhiều lý do sau đây.

[54] Từ này không dính dáng với lập trình thiên đối tượng.

14.5.2.1 Tối ưu hoá

Hiệu năng cao của đoạn mã phần lớn là do việc trình biên dịch đã tối ưu hoá đoạn mã. Đây có nghĩa là khi biên dịch, trình biên dịch sẽ nhìn vào mã nguồn xem có thể nhận diện những đoạn mà nó có thể thay thế bởi một đoạn mã khác cho ra kết quả y chang nhưng tối ưu hơn, chạy hiệu quả hơn. Thí dụ, trình biên dịch gặp phải một khúc như sau:

```
double InchesToCm (double Ins)
{
   return Ins*2.54;
}

// về sau trên đoạn mã
Y = InchesToCm(X);
```

Trình biên dịch có thể thay thế câu lệnh trên bởi:

```
Y = X*2.54;
```

Hoặc chẳng hạn trình biên dịch gặp phải đoạn mã như sau:

```
{
   string Message = "Hì hì!";
   Console.WriteLine(Message);
}
```

thì nó cho thay thế bởi:

```
Console.WriteLine("Hì hì!");
```

Như vậy, ta có thể tiết kiệm những nơi nào có những khai báo qui chiếu đối tượng một cách không cần thiết.

Các nhà sản xuất phần mềm không hề hé lộ những thủ thuật tối ưu hoá chương trình của họ, nhưng việc làm này là có thật, kết quả là hoạt động của chương trình của bạn không bị ảnh hưởng, chỉ có nội dung chương trình khả thi là có bị ảnh hưởng. Nhưng chương trình của bạn chạy nhanh hơn, đó là điều tốt lành bạn mong chờ. Tuy nhiên, điều này lại không tốt cho việc gỡ rối. Giả sử với thí dụ thứ nhất kể trên, bạn muốn đặt một chốt ngừng trong lòng hàm hành sự **InchesToCm()**, để xem việc gì xảy ra trong hàm. Làm thế nào bạn có thể gỡ rối khi thực thụ hàm trên không hiện diện do việc tối ưu hoá mà trình biên dịch đã gỡ bỏ đi rồi, hoặc làm thế nào bạn có thể đặt để một quan sát (watch) đối với biến Message khi nó cũng bị gỡ bỏ bởi trình biên dịch.

14.5.2.2 Các ký hiệu debugger

Khi bạn gỡ rối chương trình, thường bạn cần xét đến trị của những biến mà bạn sẽ khai báo chúng theo tên trên mã nguồn. Khổ nỗi là chương trình khả thi EXE lại không chứa tên biến vì trình biên dịch đã cho thay thế bởi những vị chỉ ký ức. .NET đã thay đổi tình trạng trên bằng cách cho trữ vài tên trên assembly nhưng chỉ một số nhỏ liên quan đến các lớp và hàm hành sự. Yêu cầu debugger cho bạn biết trị của biến **HeightInInches** chẳng hạn thì chả đi đâu xa vì khi debugger xem xét chương trình khả thi nó chỉ thấy những vị chỉ ký ức chứ chả thấy cái nào qui chiếu về HeightInInches cả. Do đó, muốn gỡ rối đúng đắn, bạn cần có những thông tin phụ trợ (extra) nằm sẵn trên EXE. Thông tin này sẽ bao gồm tên biến, và số thứ tự hàng cho phép debugger so khớp chỉ thị ngôn ngữ máy tương ứng với chỉ thị mã nguồn nguyên thủy. Và bạn chả muốn những thông tin này trên phiên bản phân phối vì lý do tiết lộ bí mật (vì người ta có thể rã hợp – diassembly - đoạn mã của bạn) và vì kích thước sẽ tăng do những thông tin gỡ rối.

14.5.2.3 Các chỉ thị gỡ rối extra trên mã nguồn

Khi bạn đang gỡ rối, trong đoạn mã của bạn sẽ có phụ thêm (extra) những dòng lệnh cho phép hiển thị những thông tin cốt tử liên quan đến gỡ rối. Lẽ dĩ nhiên là trước khi gởi đi phân phối chương trình của bạn, bạn muốn cho gỡ bỏ những dòng lệnh extra này. Bạn có thể làm điều này bằng tay, nhưng tốt hơn là bạn cho đánh dấu những dòng lệnh này làm thế nào trình biên dịch sẽ bỏ qua khi biên dịch đoạn mã cần được gởi đi. Chúng ta đã biết qua những chỉ thị tiền xử lý (preprocessor directive) được trình bày ở Chương 4, "Căn bản về C#", mục 4.8 "Các chỉ thị tiền xử lý**".** Các chỉ thị này có thể dùng phối hợp với thuộc tính `Conditional`, xem như là điều kiện biên dịch.

Cuối cùng thì bạn thấy là biên dịch một phiên bản dùng "xuất xưởng" một sản phẩm phần mềm khác so với phiên bản đang được gỡ rối. Visual Studio .NET thực hiện việc này bằng cách trữ nhiều chi tiết hơn để có thể hỗ trợ hai loại xây dựng khác nhau. Những bộ chi tiết khác nhau về thông tin xây dựng được gọi là **configurations** (cấu hình). Khi bạn tạo một dự án mới, Visual Studio .NET sẽ tự động tạo cho bạn hai cấu hình được cho mang tên là `Debug` và `Release`:

- **Debug configuration** thường sẽ cho biết không cần tối ưu hoá, thông tin gỡ rối extra sẽ hiệu đính trên đoạn mã khả thi, và trình biên dịch giả định là chỉ thị tiền xử lý `Debug` hiện diện trừ khi nó được ghi rõ ra là `#undefined` trong mã nguồn.

- **Release configuration** thường cho trình biên dịch biết là phải tối ưu hóa, và như vậy sẽ không có những thông tin extra liên quan đền gỡ rối, và trình biên dịch không được giả định là bất cứ chỉ thị tiền xử lý hiện diện trong đoạn mã.

14.5.3 Chọn một cấu hình

Một câu hỏi được đặt ra là Visual Studio .NET trữ những chi tiết trên hai cấu hình, thế thì cấu hình nào sẽ được chọn khi xây dựng một dự án. Câu trả lời là bao giờ cũng có một cấu hình hiện dịch (active configuration) mà Visual Studio .NET phải dùng đến. Theo mặc nhiên, khi bạn tạo một dự án, cấu hình `Debug` bao giờ cũng là cấu hình hiện dịch. Bạn có thể thay đổi cấu hình hiện dịch bằng cách ra lệnh **Build | Configuration Manager...** để cho hiện lên khung đối thoại **Configuration Manager**, rồi bạn chọn mục **Debug** hoặc **Release** trên ô liệt kê **Active Solution Configuration**, hoặc trên thanh công cụ chính có ô liệt kê cho phép bạn chọn **Debug** hoặc **Release** hoặc **Configuration Manager…**

14.5.4 Hiệu đính cấu hình

Ngoài việc chọn một cấu hình hiện dịch, bạn cũng có thể xem xét và hiệu đính cấu hình. Muốn thế, bạn chọn trên cửa sổ Solution Explorer dự án bạn muốn hiệu đính cấu hình, rồi ra lệnh **Projects | Properties** hoặc bạn right-click lên tên dự án trên cửa sổ Solution Explorer, để cho hiện lên trình đơn shortcut, rồi bạn chọn mục **Properties**. Lúc này khung đối thoại **Property Pages** hiện lên, hình 14.44:

Khung đối thoại này chứa một tree view phía tay trái cho phép bạn lựa chọn khá nhiều vùng khác nhau để quan sát hoặc hiệu đính. Chúng tôi không thể chỉ cho thấy tất cả các lĩnh vực, nhưng chúng tôi sẽ cho thấy vài lĩnh vực quan trọng.

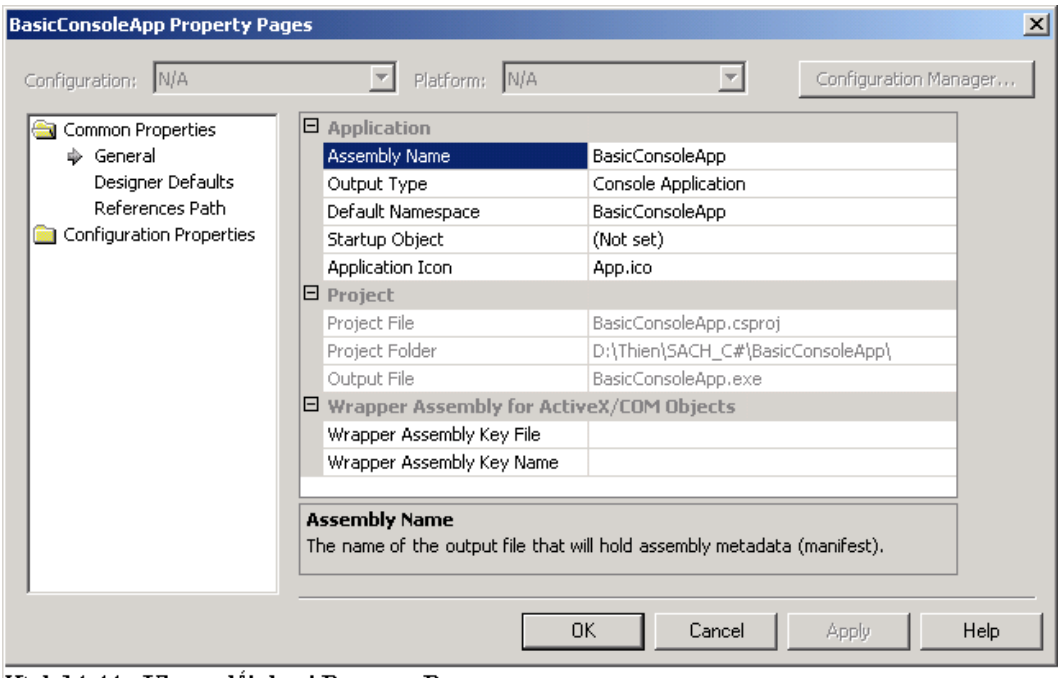
Trên hình 14.44, tree view cho thấy có hai mắt gút cấp cao: **Common Properties** (các thuộc tính thông dụng đối với tất cả các cấu hình) và **Configuration Properties** (thuộc tính cụ thể đối với một cấu hình đặc biệt).

Hình 14.44 cho thấy **Common Properties | General** đối với dự án **Basic ConsoleApp**. Bạn có thể chọn tên assembly cũng như loại assembly cần được kết sinh. Mục chọn **Output type** ở đây là: console application, windows application và class library. Lẽ dĩ nhiên, bạn có thể thay đổi loại assembly nếu bạn muốn, nhưng xem ra hơi vô lý, vì lúc ban đầu bạn đã chọn rồi trên khung đối thoại **New Project**.

Hình 14.45 kế tiếp cho thấy những thuộc tính cấu hình xây dựng.

Trên đầu khung đối thoại, bạn thấy có một list box “Configuration” cho phép bạn khai báo cấu hình nào bạn muốn quan sát. Trong trường hợp cấu hình **Debug**, ta có thể thấy là những chỉ thị tiền xử lý DEBUG và TRACE được giả định là có, việc tối ưu hóa

không được thực hiện và những thông tin gỡ rối (General Debugging Information) sẽ được kết sinh.



Hình 14-44: Khung đối thoại Property Pages

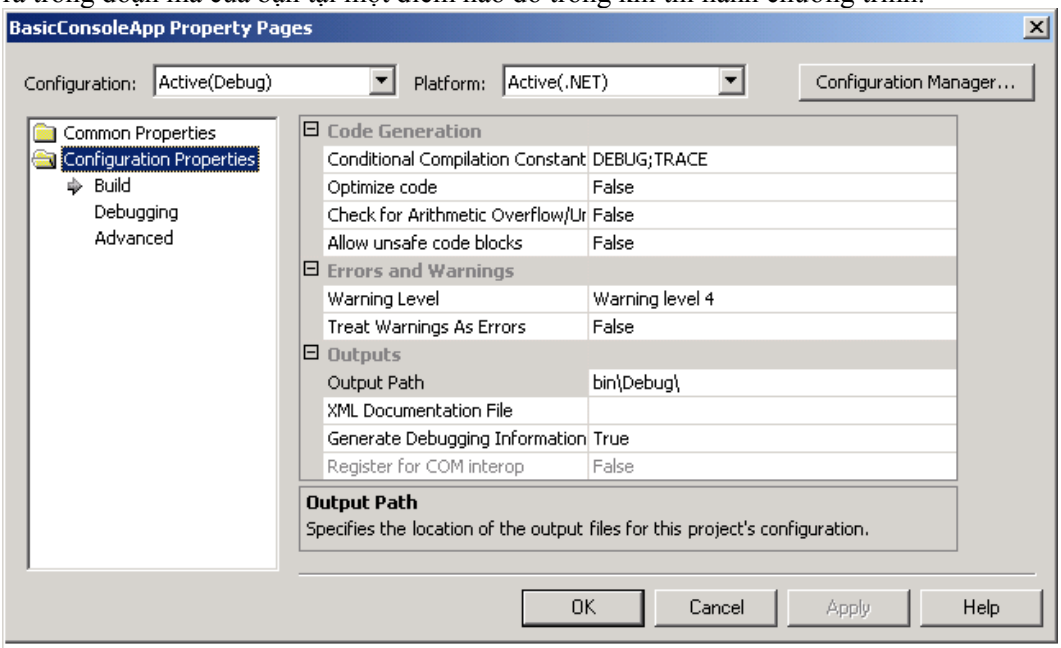
Nhìn chung, sở dĩ chúng tôi đi vào chi tiết liên quan đến cấu hình, nhung phần lớn trường hợp ít khi bạn phải điều chỉnh chúng. Tuy nhiên, bạn phải hiểu để chọn đúng cấu hình tùy theo việc bạn xây dựng thế nào dự án của bạn, và cũng là điều bổ ích khi biết tác dụng của những cấu hình khác nhau.

14.6 Gỡ rối chương trình

Chương 3, "Sử dụng Debugger thế nào?", phần I bộ sách này, đã đề cập đến việc sử dụng Debugger của Visual Studio .NET. Ở đây chúng tôi không trở lại vấn đề, nhưng chỉ lược ngắn gọn lại vài chức năng mà Visual Studio .NET cung cấp. Chúng tôi cũng sẽ đề cập chi tiết đến cách xử lý các biệt lệ vì chúng có thể gây ra vấn đề đối với việc gỡ rối.

Trong chương 12, "Thụ lý biệt lệ", chúng tôi cũng đã đề cập đến việc thụ lý biệt lệ về mặt lập trình.

Trên C#, cũng như trên các ngôn ngữ đi trước .NET, kỹ thuật chính trong việc gỡ rối chương trình là đơn giản đặt những chốt ngừng, rồi sử dụng chúng để xem xét việc gì xảy ra trong đoạn mã của bạn tại một điểm nào đó trong khi thi hành chương trình.



Hình 14-45: Khung đối thoại Property Pages: Configuration Properties

14.6.1 Các chốt ngừng (breakpoint)

Bạn có thể cho đặt các chốt ngừng từ Visual Studio .NET lên bất cứ dòng lệnh nào trên đoạn mã hiện đang được thi hành. Cách đơn giản nhất là click lên dòng lệnh trên code editor, trên biên trái màu xám (hoặc ấn phím <F9> khi bạn cho con nháy chĩa lên dòng lệnh thích ứng). Dòng lệnh sẽ đổi màu với một chấm tròn ở biên trái. Lúc này một chốt ngừng được đặt để trên hàng này, làm cho việc thi hành sẽ tạm ngưng khi debugger đạt đến hàng này và quyền điều khiển được trao qua cho debugger. Nếu bạn click lên chấm tròn của dòng lệnh này thì xem như chốt ngừng bị gỡ bỏ, dòng lệnh trở lại màu bình thường của văn bản.

Bạn có thể thiết đặt những chốt ngừng có điều kiện. Muốn thế, bạn ra lệnh **Debug | New Breakpoint**, để cho hiện lên khung đối thoại **New Breakpoint** với những chi tiết mà bạn có thể đặt để. Trong một số mục chọn bạn có thể:

- Khai báo việc thi hành chỉ được tạm ngưng sau khi chốt ngừng bị đụng theo một số lần nào đó, hay theo bội số lần nào đó. Bạn sử dụng Hit count để làm việc này.

- Cho đặt để chốt ngừng dựa theo trị của một biến thay vì theo chỉ thị. Ta gọi là data breakpoint. Trong trường hợp này chốt ngừng sẽ được điều khiển bởi sự thay đổi của trị của một biến nào đó. Việc kiểm tra sự thay đổi trị của một biến sẽ đòi hỏi nhiều thời gian của processor, do đó đoạn mã của bạn sẽ chạy chậm lại.

Bạn có thể xem lại chương 3, "Sử dụng Debugger thế nào?", để biết thêm chi tiết liên quan đến Hit count và Condition.

14.6.2 Các cửa sổ quan sát (Watches window)

Một khi chốt ngừng đã đạt đến, thường thì bạn muốn khảo sát trị của các biến. Cách đơn giản nhất là cho con nháy chĩa vào tên biến bạn muốn khảo sát. Lúc này một ô hình chữ nhật màu vàng hiện lên cho biết trị của biến. Tuy nhiên, có thể bạn lại muốn sử dụng cửa sổ **Watch Window** để xem nội dung của các biến Cửa sổ **Watch Window** thuộc loại cửa sổ dạng thẻ (tab window). Cuối màn hình code editor là một loạt thẻ các cửa sổ: **Local, Auto, Watch, Call Stack, Breakpoint, Command, Output** v.v..Các cửa sổ này chỉ hiện lên khi chương trình đang chạy dưới sự điều khiển của Debugger.

Có 3 loại cửa sổ kiểu tab dùng điều khiển những biến khác nhau:

- **Autos:** cho biết tình trạng của một vài biến được truy xuất khi chương trình đang thi hành.

- **Locals**: cho biết tình trạng của biến được truy xuất trong hàm hành sự đang được thi hành.

- **Watch**: cho biết tình trạng của bất cứ biến nào mà bạn đã khai báo rõ ra bằng cách khỏ vào tên biến trong cửa sổ Watch.

Một lần nữa, bạn xem lại chi tiết ở chương 3, "Sử dụng Debugger thế nào?".

14.6.3 Biệt lệ (exceptions)

Biệt lệ cho phép bạn thụ lý một cách thích ứng những điều kiện sai lầm xảy ra trong ứng dụng mà bạn muốn phân phối. Nếu được sử dụng đúng đắn, biệt lệ bảo đảm ứng dụng của bạn kiểm soát tốt hoạt động và người sử dụng không phải phiền lòng khi nhận những khung đối thoại mang tính quá kỹ thuật. Tuy nhiên, điều phiền toái là khi gỡ rối, biệt lệ gây không ít khó khăn cho bạn. Có hai vấn đề:

- Nếu biệt lệ xảy ra, thì liền sau đó khi bạn đang gỡ rối thường thì bạn không muốn biệt lệ được tự động thụ lý bởi chương trình của bạn - đặc biệt khi thụ lý có thể

chương trình sẽ được "hạ cánh một cách đẹp đẽ và an toàn". Thay vào đó, bạn muốn **Debugger** nhập cuộc để xem ra vì sao biệt lệ xảy ra, để có thể gỡ bỏ lý do sai lầm trước khi phân phối sản phẩm.

- Nếu một biệt lệ xảy ra mà bạn chưa hề viết một hàm thụ lý biệt lệ này, thì .NET Runtime sẽ đi tìm tiếp một hàm thụ lý biệt lệ. Tuy nhiên, đến khi .NET Runtime nhận ra là không có hàm nào, nó sẽ cho chấm dứt chương trình. Và lúc này, cửa sổ Call Stack cũng sẽ biến mất không còn gì để bạn có thể xem xét trị của các biến, vì chúng đã ra ngoài phạm vi.

Lẽ dĩ nhiên, bạn có thể thiết đặt những chốt ngừng ngay trên các khối **catch**. Nhưng việc này cũng không giúp ích chi nhiều đối với bạn, vì rằng khi bạn nhảy vào khối catch thì việc thi hành sẽ thoát khỏi khối **try** tương ứng và như thế các biến bạn muốn quan sát trị để xem vì sao sai lầm xảy ra, sẽ ra ngoài scope rồi. Bạn không có khả năng xem **Stack Trace** để biết hàm hành sự nào đã được thi hành khi lệnh **throw** xảy ra. Nếu bạn cho đặt chốt ngừng tại lệnh **throw**, bạn có thể giải quyết vấn đề, nhưng nếu có quá nhiều lệnh **throw** trong chương trình của bạn, thì làm sao biết lệnh **throw** nào đã tung ra biệt lệ ?

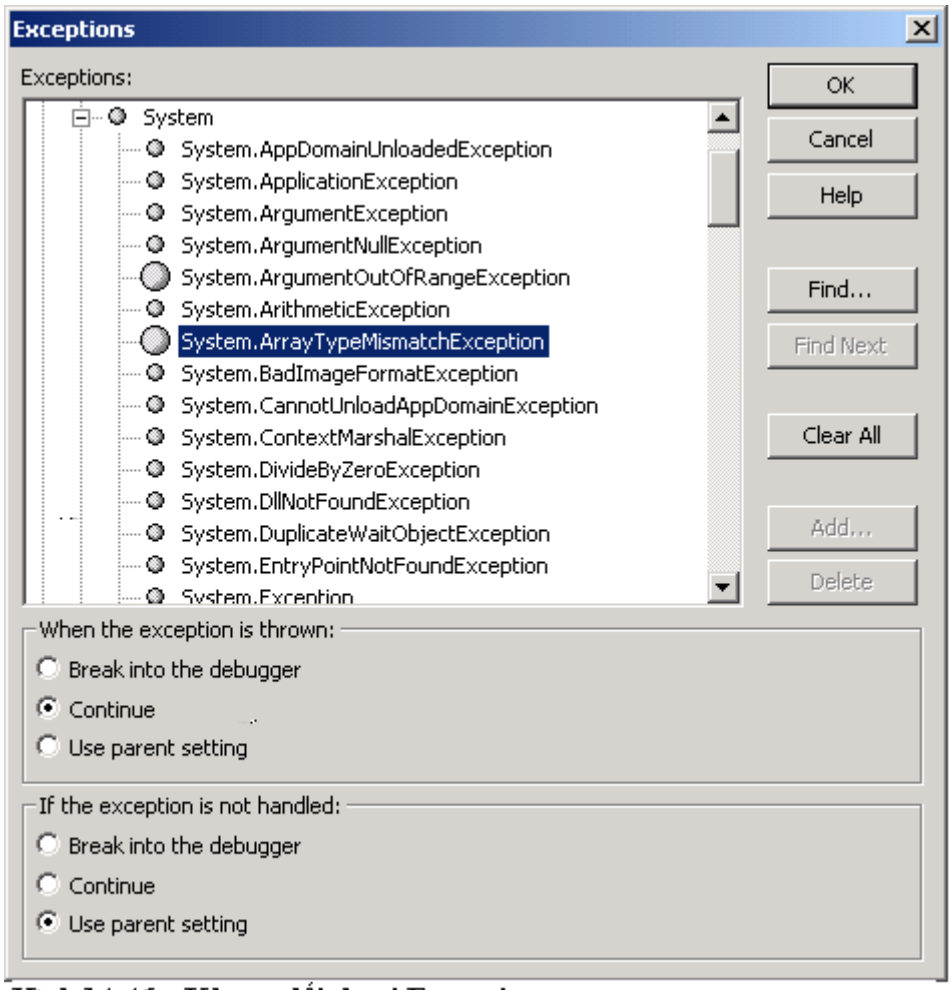
Thật ra, Visual Studio .NET có câu trả lời cho sự việc này. Nếu bạn ra lệnh **Debug | Exceptions**, bạn sẽ thấy hiện lên khung đối thoại **Exceptions** (hình 14.46) cho phép bạn khai báo việc gì sẽ xảy ra khi một biệt lệ được tung ra. Bạn có thể chọn tiếp tục thi hành (nút Continue) hoặc tự động ngưng và bắt đầu gỡ rối (nút Break into Debugger), trong trường hợp này việc thi hành sẽ ngưng và debugger nhập cuộc vào ngay lệnh throw.

Đây là một công cụ thật sự mạnh vì bạn có thể customize cách ứng xử tùy theo loại lớp biệt lệ nào được tung ra. Thí dụ, trên hình 14.46, ta có thể yêu cầu Visual Studio .NET cho tạm ngưng chạy, chuyển quyền điều khiển cho debugger bất cứ lúc nào một biệt lệ được tung ra bởi một lớp cơ bản .NET (cho thấy bởi dấu thập đỏ cạnh loại biệt lệ), nhưng không được ngưng chui vào debugger nếu biệt lệ thuộc loại `ArgumentOutOfRangeException` hoặc loại `ArrayTypeMismatchException`.

Visual Studio .NET biết tất cả các lớp biệt lệ có sẵn trong các lớp cơ bản .NET, cũng như một vài biệt lệ có thể được tung ra ngoài môi trường .NET. Visual Studio .NET không tự động biết đến những biệt lệ tự tạo của lập trình viên, nhưng bạn có thể đưa vào bằng tay những lớp biệt lệ vào danh sách, và như vậy khai báo cho biết những biệt lệ nào làm cho việc thi hành ngưng ngay lập tức. Muốn thế, bạn chỉ cần click nút **Add** ở trên (nút này có hiệu lực khi bạn click lên mắt gút cấp cao trên cây biệt lệ, chẳng hạn CLR Runtime) rồi khỏ tên của lớp biệt lệ của bạn.

14.7 Các công cụ .NET khác

Chúng ta mất khá nhiều thời giờ khảo sát Visual Studio .NET, vì đây là công cụ mà bạn phải sử dụng nhiều trong suốt thời gian triển khai hệ thống phần mềm của bạn. Tuy nhiên, có một số công cụ khác giúp bạn trong việc lập trình. Có hai chương trình mà bạn phải làm quen:

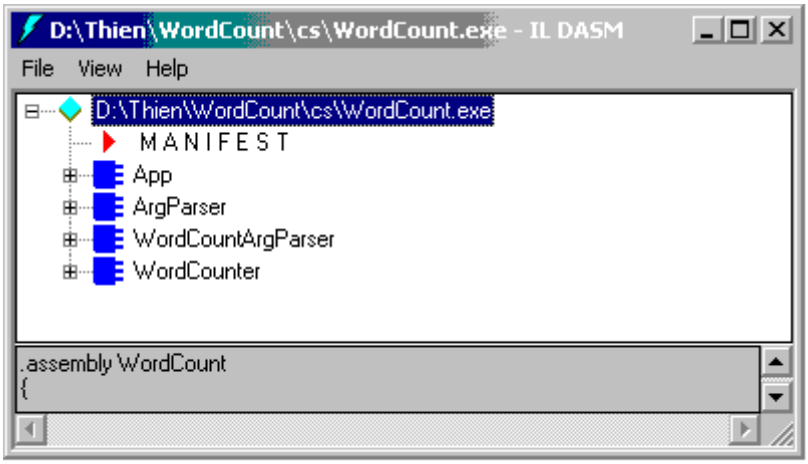


Hình 14-46: Khung đối thoại Exceptions

- Trình tiện ích ILDasm.exe, tắt chữ Intermediate Language Disassembler.
- Ứng dụng desktop WinCV, tắt chữ Windows Class Viewer
- Ứng dụng ClassView Web.

14.7.1 Sử dụng ILDasm.exe

ILDasm.exe (MSIL Disassembler) cho phép bạn nạp vào bất cứ .NET assembly nào để bạn có cơ hội khảo sát nội dung (bao gồm manifest liên đới, bộ chỉ thị IL và kiểu dữ liệu metadata) sử dụng giao diện GUI thân thiện. **ILDasm.exe** không những chỉ cho thấy đoạn mã IL mà còn hiển thị namespace, và kiểu dữ liệu bao gồm những giao diện của chúng. Bạn có thể dùng **ILDasm.exe** để quan sát các assembly bẩm sinh của NET Framework, chẳng hạn Mscorlib.dll, mà kể cả assembly mà bạn đã tạo ra. Phần lớn lập trình viên NET Framework sẽ thấy **ILDasm.exe** là rất cần thiết.



Hình 14-47: ILDASM.EXE đang hoạt động

Để bắt đầu, ta có thể sử dụng thí dụ WordCount trên thư mục **Microsoft NET\SDK\Samples\Applications\WordCount**. Bạn cho sao một bản WordCount lên thư mục nào đó của bạn, rồi cho Build trên IDE. Sau đó, bạn ra lệnh **Start | Run** rồi khỏ vào:

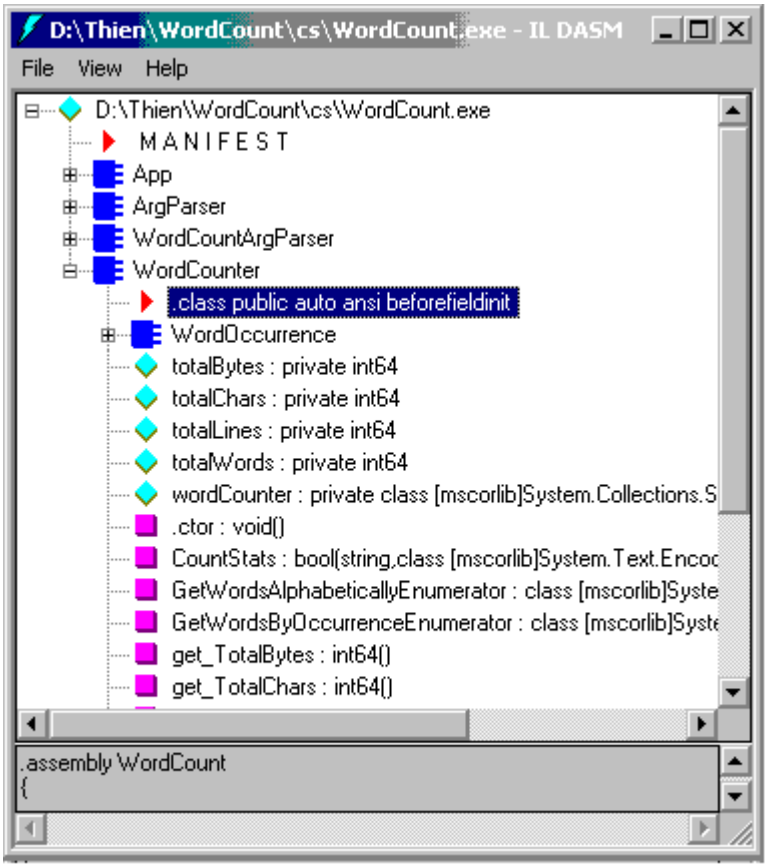
```
ildasm WordCount.exe
```

Lúc này, cửa sổ Ildasm.exe sẽ hiện lên như theo hình 14-47. Cấu trúc cây trên hình 14-47 cho thấy thông tin manifest của as sembly được chứa trong lòng WordCount exe và 4 lớp global: **App**, **ArgParser**, **WordCountArgParser,** và **WordCounter**. Bằng cách double click lên bất cứ kiểu dữ liệu nào trên cây này, bạn có thể thấy nhiều thông tin hơn liên quan đến kiểu dữ liệu này. Hình 14-48 sau đây cho bung lớp **WordCounter. Bạn** có thể thấy tất cả các thành viên của **WordCounter**, với những ký hiệu đồ hoạ. (Muốn biết ý nghĩa các ký hiệu này, đề nghị bạn tham khảo mục "Ildasm.exe tutorial" trên Help MSDN).

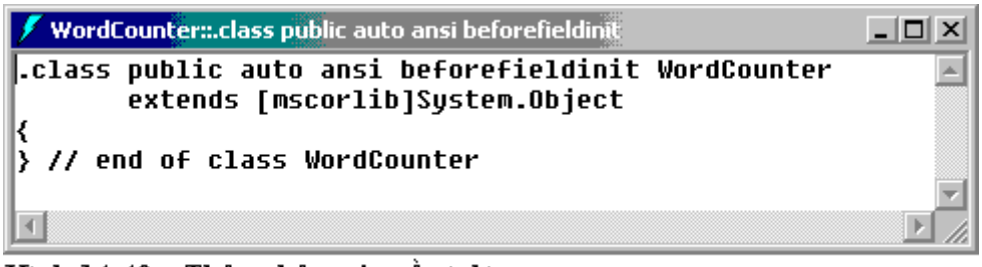
Bằng cách double click lên mục vào **.class public auto ansi beforefieldinit**, hình 14-49 hiện lên cho thấy thông tin sau đây. Ký hiệu mũi tên cho biết là cần thêm thông tin.

Trên hình 14-49, bạn có thể thấy kiểu dữ liệu **WordCounter** được dẫn xuất từ lớp **System.Object**. Ngoài ra, kiểu dữ liệu **WordCounter** còn chứa một kiểu dữ liệu khác

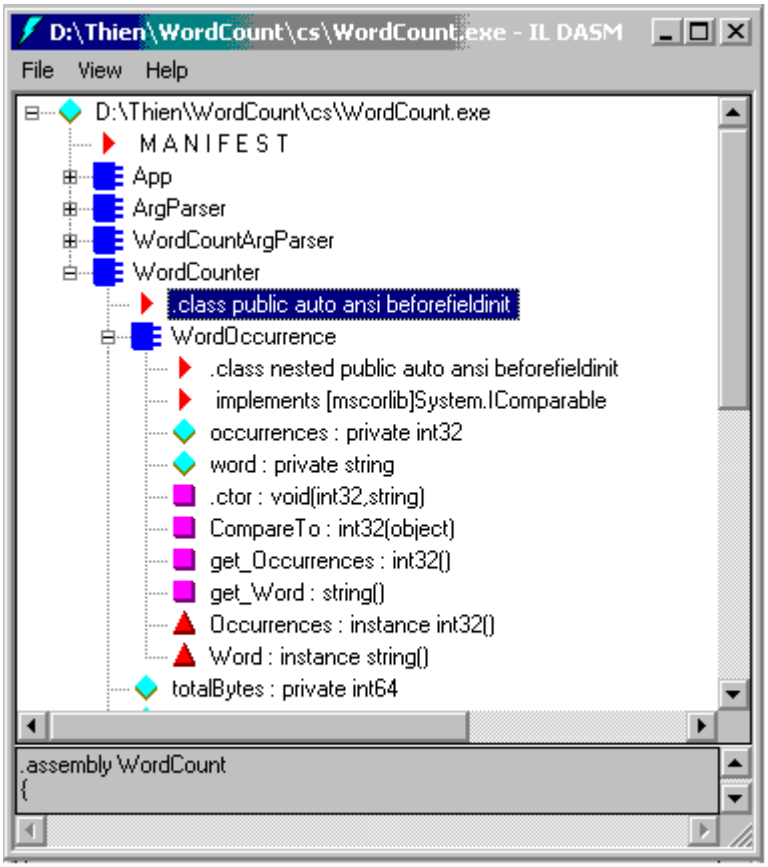
mang tên **WordOccurrence**. Trên hình 14-50, chúng tôi cũng đã cho bung kiểu dữ liệu **WordOccurrence** để xem các thành viên của nó.



Hình 14-48: WordCounter được bung ra



Hình 14-49: Thêm thông tin về gì đó.



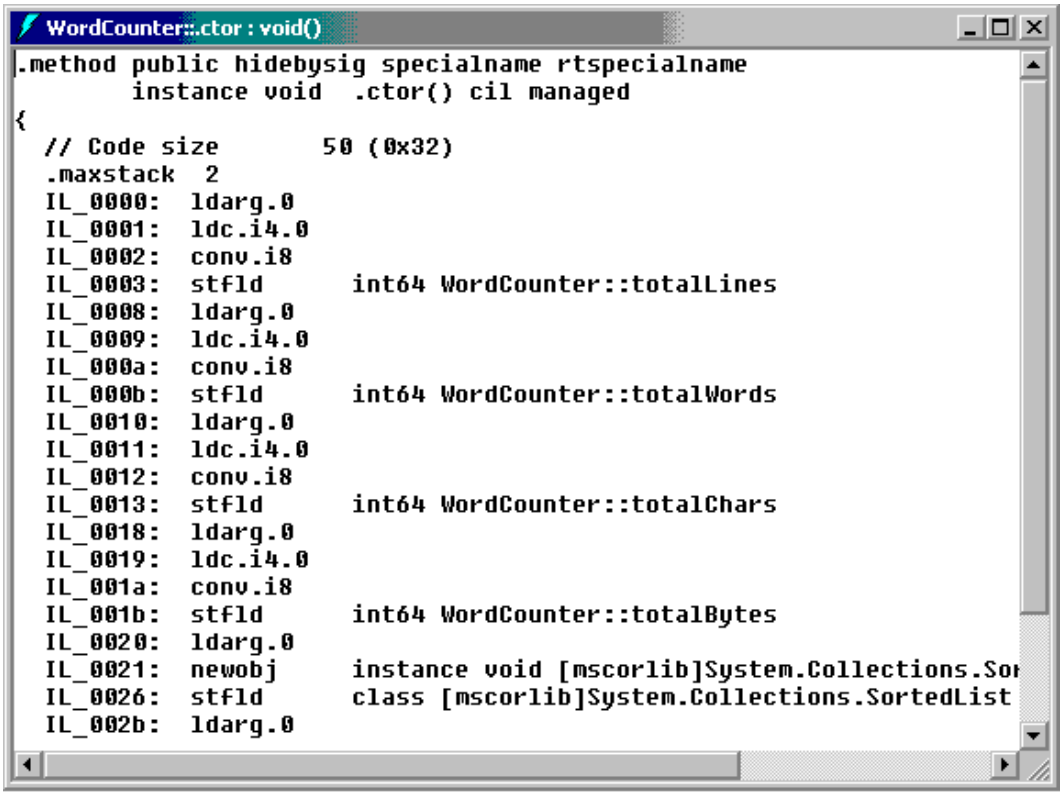
Hình 14-50: WordCounter và WordOccurrence được bung ra

Nếu nhìn vào cây hình 14-50, bạn có thể thấy là **WordOccurrence** cho thi công giao diện **System.IComparable** - đặc biệt hàm hành sự **CompareTo**. Trong phần này, chúng tôi không bàn đến **WordOccurrence**, mà chỉ tập trung vào lớp **WordCounter**.

Trên hình 14-48, bạn có thể thấy lớp **WordCounter** chứa 5 vùng mục tin **private**: **totalBytes**, **totalChars**, **totalLines**, **totalWords**, và **wordCounter**. Bốn vùng mục tin đầu tiên là những thể hiện của kiểu dữ liệu **int64**, trong khi vùng mục tin **wordCounter** là một qui chiếu về lớp **System.Collections.SortedList**.

Theo sau các vùng mục tin, bạn có thể thấy các hàm hành sự. Hàm đầu tiên **.ctor** là một hàm constructor. Lớp này chỉ có một hàm constructor duy nhất, nhưng các lớp khác có thể có nhiều hàm constructor - mỗi hàm mang một dấu ấn khác nhau. Hàm constructor **WordCounter** có một kiểu dữ liệu trả về là **void** không nhận thông số nào. Nếu bạn

double click hàm constructor này, thì một cửa sổ khác hiện lên cho hiển thị đoạn mã MSIL được chứa trong lòng hàm hành sự, như theo hình 14-51.



Hình 14-51: Đoạn mã IL của hàm constructor .ctor

Hiện thời đoạn mã MSIL đọc và hiểu cũng khá dễ dàng. (Muốn biết thêm chi tiết, đề nghị bạn tham khảo *CIL Instruction Set Specification*, nằm ở tập tin Partition III CIL.doc trên thư mục <FrameworkSDK>\Tool Developers Guide\Docs\ folder.).

Về phía đỉnh đầu hình 14-51, bạn có thể thấy hàm constructor đòi hỏi 50 bytes đối với đoạn mã MSIL. Từ con số này, bạn thật sự không biết đoạn mã nguyên sinh (native) sẽ chiếm bao nhiêu bytes bởi trình biên dịch JIT - vì nó tùy thuộc CPU mà trình biên dịch dùng kết sinh đoạn mã này.

CLR hoạt động theo stack. Do đó, muốn thi hành bất cứ một tác vụ nào, MSIL code trước tiên ấn các tác tố (operand) vào một virtual stack, rồi sau đó thi hành tác tử. Tác tử sẽ lấy tác tố từ stack, thi hành tác vụ được yêu cầu rồi trả kết quả về lại stack. Vào bất cứ một lúc nào, hàm này không thể có hơn 8 tác tố được ấn vào virtual stack. Bạn có thể

nhận diện con số này bằng cách nhìn vào attribute **.maxstack** ngay liền trước đoạn mã MSIL. (xem hình 14-51).

Bây giờ ta thử xét vài chỉ thị MSIL đầu tiên, được trích lại như sau 4 hàng sau đây:

```
IL_0000: ldarg.0 ; nạp con trỏ 'this' của đối tượng lên stack
IL_0001: ldc.i4.0 ; nạp trị hằng 4-byte của 0 lên stack
IL_0002: conv.i8 ; chuyển đổi 4-byte 0 thành một 8-byte 0
IL_0003: stfld int64 WordCounter::totalLines
```

Chỉ thị IL_0000 nạp thông số đầu tiên được trao qua hàm hành sự vào virtual stack. Mỗi thể hiện hàm hành sự bao giờ cũng được trao vị chỉ ký ức của đối tượng. Đối mục này được gọi là Argument Zero và không bao giờ ghi rõ ra trên dấu ấn của hàm hành sự. Do đó, cho dù hàm contructor **.ctor** xem ra không nhận argument nào, thì thật ra nó nhận một argument. Chỉ thị IL_0000, như vậy, cho nạp con trỏ (pointer) về đối tượng này vào virtual stack.

Chỉ thị IL_0001 cho nạp trị zero của một hằng số 4-byte vào virtual stack.

Còn chỉ thị IL_0002 thì lấy trị từ đỉnh stack (4-byte zero), và chuyển đổi nó thành một trị zero 8-byte — như vậy đưa 8-byte zero vào đỉnh stack.

Tới lúc này, stack chứa hai operands: 8-byte zero và con trỏ chĩa về đối tượng. Chỉ thị IL_0003 sẽ dùng cả hai operand này để trữ trị số lấy từ đỉnh stack (8-byte zero) lên vùng mục tin **totalLines** của đối tượng được nhận diện trên stack.

Loạt chỉ thị MSIL giống hệt như trên sẽ được lặp lại với các vùng mục tin **totalChars**, **totalBytes**, và **totalWords**.

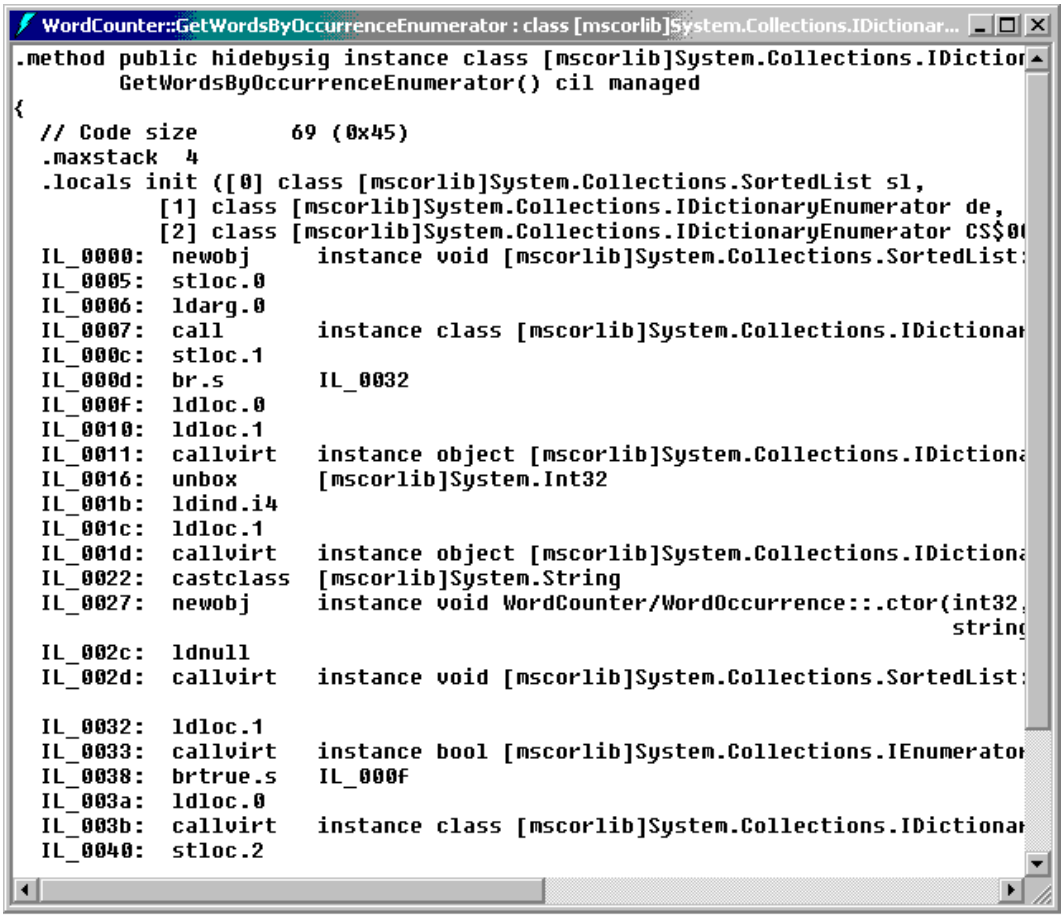
Việc khởi gán (initialization) vùng mục tin **wordCounter** bắt đầu với chỉ thị IL_0020, như sau:

```
IL_0020: ldarg.0
IL_0021: newobj instance void
[mscorlib]System.Collections.SortedList::.ctor()
IL_0026: stfld class [mscorlib]System.Collections.SortedList
WordCounter::wordCounter
```

Chỉ thị IL_0020 ấn con trỏ **this** chỉ về **WordCounter** vào virtual stack. Tác tố này không được sử dụng bởi chỉ thị **newobj** nhưng sẽ được dùng bởi chỉ thị **stfld** tại IL_0026. Chỉ thị tại IL_0021 yêu cầu CLR tạo ra một đối tượng mới kiểu dữ liệu **System.Collections.SortedList** và cho triệu gọi hàm constructor không có arguments. Khi **newobj** trả về, vị chỉ ký ức của đối tượng **SortedList** sẽ nằm trên stack. Tới lúc này,

chỉ thị **stfld** tại IL_0026 cho trữ con trỏ chỉ về đối tượng **SortedList** vào vùng mục tin **wordCounter** của đối tượng **WordCounter**.

Sau khi tất cả các vùng mục tin của đối tượng **WordCounter** đã được khởi gán, chỉ thị tại IL_002b sẽ ấn **this** pointer lên virtual stack, và IL_002c triệu gọi hàm constructor trên lớp cơ sở (**System.Object**).



Hình 14-52: GetWordByOccurrenceEnumerator được bung ra

Lẽ dĩ nhiên, chỉ thị cuối cùng tại IL_0031 là chỉ thị trở về cho phép hàm constructor **WordCounter** trở về đoạn mã đã tạo ra nó. Những hàm constructor nào trả về **void**, thì sẽ không có gì phải đưa vào stack trước khi hàm constructor trở về.

Bây giờ ta thử xem một thí dụ khác. Bạn cho double click hàm hành sự **GetWordsByOccurranceEnumerator** để xem đoạn mã MSIL sẽ ra sao như theo hình 14-52 ở trên.

Bạn thấy là đoạn mã đối với hàm hành sự chiếm dụng 69 bytes, với 4 slot đối với virtual stack. Ngoài ra, hàm hành sự này có 3 biến cục bộ: một biến thuộc kiểu dữ liệu **System.Collection.SortedList** còn hai biến kia thuộc kiểu dữ liệu **System.Collections.IDictionaryEnumerator.** Bạn để ý là các tên biến được khai báo trong mã nguồn sẽ không được phát hành tại đoạn mã MSIL, trừ khi assembly được biên dịch với flag **/debug.** Nếu **/debug** không được dùng, các tên biến **V_0**, **V_1**, và **V_2** sẽ được dùng thay vì **sl**, **de**, và **CS$00000003$00000000**.

Khi hàm hành sự này bắt đầu thi hành, việc làm đầu tiên là cho thi hành chỉ thị **newobj**, lo tạo một đối tượng mới **System.Collections.SortedList** và triệu gọi hàm constructor mặc nhiên của đối tượng này. Khi **newobj** trả về, vị chỉ của đối tượng được tạo ra sẽ nằm trong virtual stack. Chỉ thị **stloc.0** (tại IL_0005) cho trữ trị này vào biến cục bộ 0, hoặc **sl** (**V_0** without /debug) (thuộc kiểu dữ liệu **System. Collections.SortedList**).

Tại chỉ thị IL_0006 và IL_0007, con trỏ **this** của đối tượng **WordCounter** (trong Argument Zero được trao qua cho hàm hành sự) được nạp vào stack, và hàm hành sự **GetWordsAlphabeticallyEnumerator** được triệu gọi. Khi chỉ thị **call** trở về, vị chỉ ký ức của enumerator nằm trên stack. Chỉ thị **stloc.1** (tại IL_000c) cho trữ vị chỉ này ở biến cục bộ 1, hoặc **de** (**V_1** without /debug) thuộc kiểu dữ liệu **System.Collections. IDictionaryEnumerator**.

Chỉ thị **br.s** tại IL_000d là lệnh nhảy vô điều kiện về **IL** test condition của lệnh **while** statement. **IL** test condition này bắt đầu từ chỉ thị IL_0032. Tại IL_0032, vị chỉ của **de** (hoặc **V_1**) (**IDictionaryEnumerator**) bị ấn vào stack và, tại IL_0033, hàm hành sự **MoveNext** của nó sẽ được triệu gọi. Nếu **MoveNext** trả về **true**, một mục vào hiện hữu sẽ được liệt kê, và chỉ thị **brtrue.s** sẽ nhảy về chỉ thị tại IL_000f.

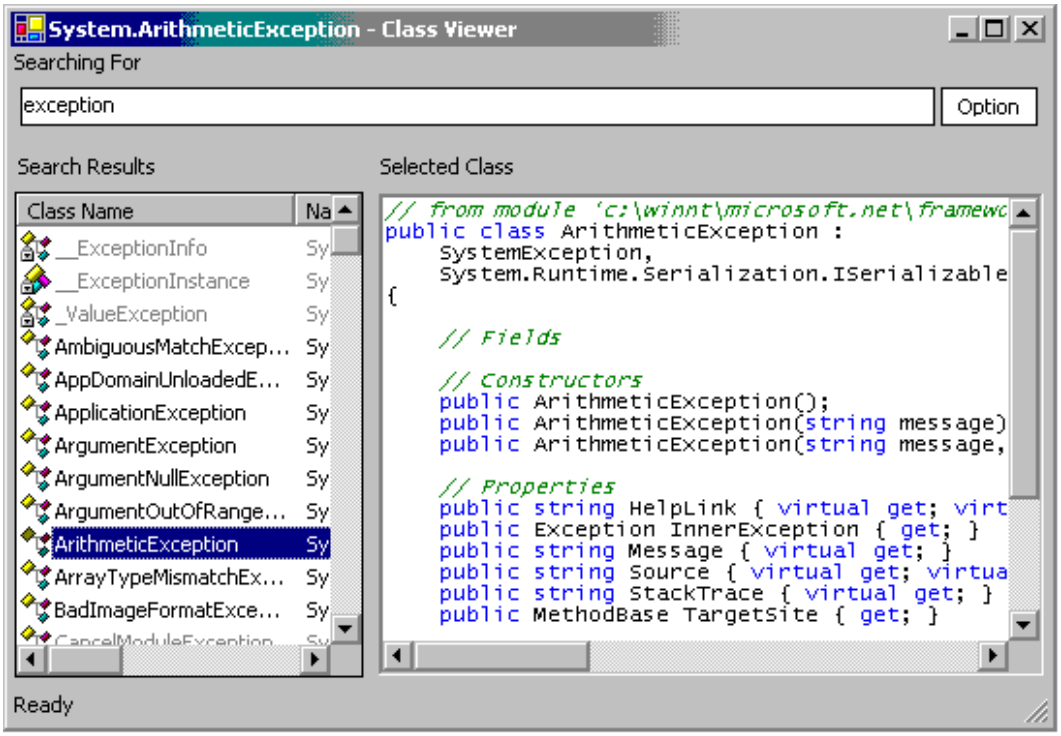
Tại các chỉ thị IL_000f và IL_0010, các vị chỉ của của các đối tượng trên **sl** (hoặc **V_0**) và **de** (hoặc **V_1**) được ấn vào stack. Sau đó, hàm thuộc tính **get_Value** của đối tượng **IdictionaryEnumerator** được triệu gọi để lấy số lần xuất hiện của mục vào hiện hành. Con số này là một trị 32-bit được trữ trong một **System.Int32**. Đoạn mã sẽ ép kiểu (cast) đối tượng **Int32** này thành một **int** value type. Casting một reference type thành một value type đòi hỏi chỉ thị **unbox** tại IL_0016. Khi **unbox** trả về, vị chỉ của trị được unboxed sẽ nằm trên stack. Chỉ thị **ldind.i4** (tại IL_001b) nạp trị 4-byte, chỉ về vị chỉ hiện hành trên stack. Nói cách khác, số nguyên unboxed 4-byte được đưa vào stack.

Tại chỉ thị IL_001c, trị của **sl** (hoặc **V_1**) – vị chỉ ký ức của **IDictionary Enumerator**) – được ấn vào stack, và hàm thuộc tính **get_Key** của nó được triệu gọi. Khi **get_Key** trả về, vị chỉ của **System.Object** nằm trên stack. Đoạn mã biết là dictionary chứa các chuỗi, do đó trình biên dịch ép kiểu **Object** này thành một chuỗi kiểu **String** sử dụng chỉ thị **castclass** tại IL_0022.

Vài chỉ thị kế tiếp (từ IL_0027 đến IL_002d) sẽ tạo một đối tượng mới **Word Occurrence**, và trao vị chỉ ký ức của đối tượng cho hàm hành sự **Add** của **Sorted Lists**.

Tại chỉ thị IL_0032, điều kiện trắc nghiệm của lệnh **while** lại được định trị một lần nữa. Nếu **MoveNext** trả về **true**, vòng lặp sẽ thi hành một iteration khác. Tuy nhiên, nếu **MoveNext** trả về **false**, việc thi hành thoát khỏi vòng lặp và nhảy về chỉ thị tại IL_003a. Các chỉ thị từ IL_003a đến IL_0040 sẽ triệu gọi hàm hành sự **GetEnumerator** của đối tượng **SortLists**. Trị trả về là một đối tượng **System.Collections.IDictionary Enumerator**, được để lại trên stack để trở thành trị trả về **GetWordsBy OccurrenceEnumerator**.

14.7.2 Sử dụng Windows Forms Class Viewer (Wincv.exe)



Hình 14-53: WinCV.exe đang hoạt động

WinCV là một trình tiện ích mà Microsoft cung cấp cho phép bạn khảo sát các lớp cơ bản và xem các hàm hành sự nào có sẵn. Nó cũng tương tự như Object Browser của Visual Studio .NET, ngoại trừ nó là một ứng dụng hoàn toàn độc lập, và nó cho thấy tất

cả các lớp cơ bản, trong khi Object Browser chỉ cho thấy những lớp trong assembly mà dự án của bạn qui chiếu về.

Sử dụng WinCV cũng khá dễ dàng. Bạn ra lệnh **Start | Run,** rồi khỏ lệnh **cmd.exe** để cho cửa sổ command line hiện lên. Tiếp theo, bạn khỏ vào WinCV.exe và chờ cho cửa sổ của WinCV hiện lên. Hình 14-53 (trang trước). Trên khung đối thoại này, bạn khỏ vào một phần tên lớp nào bạn muốn xem vào ô text box trên đầu khung đối thoại.

Ô list box Class Name phía tay trái trên khung đối thoại cho liệt kê một danh sách các lớp mà Wincv.exe tìm thấy dựa trên tên bạn khò vào. Namespace **System** được xem như là hiểu ngầm đối với tên các lớp; do đó khi bạn khỏ kiểu dữ liệu "Object" được xem như là bạn khỏ vào "System.Object". Khi bạn chọn một kiểu dữ liệu trên danh sách phía tay trái thì định nghĩa lớp sẽ hiện lên trên khung cửa phía tay phải. Định nghĩa lớp sẽ được hiển thị sử dụng cú pháp C#.

Thí dụ lệnh sau đây cho chạy WinCV.exe và nạp myApp.exe và các assembly mặc nhiên cho phép bạn rảo xem:

```
wincv /r:myApp.exe
```

Còn lệnh sử dụng cho chạy WinCV.exe và chỉ nạp myApp.exe mà thôi để rảo xem, các assembly mặc nhiên không được nạp vào:

```
wincv /r:myApp.exe /nostdlib+
```

Bản chỉ mục