TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM

PHẠM VĂN VIỆT - TRƯƠNG LẬP VĨ

TÌM HIỂU NGÔN NGỮ C# VÀ VIẾT MỘT ỨNG DỤNG MINH HỌA

ĐỒ ÁN TỐT NGIỆP

GIÁO VIÊN HƯỚNG DẪN
NGUYỄN TẤN TRẦN MINH KHANG

TP. HCM 2002

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM

PHẠM VĂN VIỆT - TRƯƠNG LẬP VĨ

TÌM HIỂU NGÔN NGỮ C# VÀ VIẾT MỘT ỨNG DỤNG MINH HỌA

GIÁO VIÊN HƯỚNG DẪN
NGUYỄN TẤN TRẦN MINH KHANG

TP. HCM 2002

Lời cám ơn

Để có thể hoàn tất được bài đồ án này, trước tiên phải kể đến công sức của thầy Nguyễn Tấn Trần Minh Khang. Chúng em kính lời cảm ơn đến thầy đã tận tình hướng dẫn và giúp đỡ trong thời gian thực hiện đồ án này.

Chúng em xin tỏ lòng biết ơn sâu sắc đối với gia đình đã động viên, tạo điều kiện để thực hiện tốt bài đồ án. Xin cám ơn cha, mẹ, anh, chị, em!

Chúng em cũng xin chân thành cảm ơn đến các thầy cô khoa Công nghệ thông tin trường Đại học Khoa học Tự nhiên Thành phố Hồ Chí Minh đã truyền đạt những kiến thức, kinh nghiệm quí báu cho chúng em trong quá trình học tập tại trường.

Chúng em cũng xin chân thành cảm ơn đến các bạn bè đã giúp đỡ tài liệu, trao đổi học thuật mới có thể thực hiện đồ án này. Xin gởi lời cảm ơn đến các bạn Hồ Ngọc Huy, Trần Thế Anh, Bùi Thanh Tuấn...

Thành phố Hồ Chí Minh, ngày 03 tháng 8 năm 2002

Sinh viên

Phạm Văn Việt

Trương Lập Vĩ

Mục lục

Tóm tắt

Đề tài này tập trung tìm hiểu toàn bộ các khái niệm liên quan đến ngôn ngữ C#. Bởi vì C# được Microsoft phát triển như là một thành phần của khung ứng dụng .NET Framework và hướng Internet nên đề tài này bao gồm hai phần sau:

Phần 1: Tìm hiểu về ngôn ngữ C#

Việc tìm hiểu bao gồm cả các kiến thức nền tảng về công nghệ .NET Framework, chuẩn bị cho các khái niệm liên quan giữa C# và .NET Framework. Sau đó tìm hiểu về bộ cú pháp của ngôn ngữ này, bao gồm toàn bộ tập lệnh, từ khóa, khái niệm về lập trình hướng đối tượng theo C#, các hỗ trợ lập trình hướng component ... Sau cùng là cách lập trình C# với ứng dụng Window cho máy để bàn và C# với các công nghệ hiện đại như ASP.NET. ADO.NET, XML cho lập trình Web.

Phần 2: Xây dựng một ứng dụng

Phần này là báo cáo về ứng dụng minh họa cho việc tìm hiểu ở trên. Tên ứng dụng là Xây dựng một Website dạy học C#. Đây là ứng dụng Web cài đặt bằng ngôn ngữ C# và ASP.NET. Trong đó ASP.NET được dùng để xây dựng giao diện tương tác với người dùng; còn C# là ngôn ngữ lập trình bên dưới. Ứng dụng có thao tác cơ sở dữ liệu (Microsoft SQL Server) thông quan mô hình ADO.NET.

Phần 1

Tìm hiểu ngôn ngữ C#

Chương 1 C# và .Net Framework

Mục tiêu của C# là cung cấp một ngôn ngữ lập trình đơn giản, an toàn, hiện đại, hướng đối tượng, đặt trọng tâm vào Internet, có khả năng thực thi cao cho môi trường .NET. C# là một ngôn ngữ mới, nhưng tích hợp trong nó những tinh hoa của ba thập kỷ phát triển của ngôn ngữ lập trình. Ta có thể dễ dàng thấy trong C# có những đặc trưng quen thuộc của Java, C++, Visual Basic, …

Đề tài này đặt trọng tâm giới thiệu ngôn ngữ C# và cách dùng nó như là một công cụ lập trình trên nền tảng .NET. Với ngôn ngữ C++, khi học nó ta không cần quan tâm đến môi trường thực thi. Với ngôn ngữ C#, ta học để tạo một ứng dụng .NET, nếu lơ là ý này có thể bỏ lỡ quan điểm chính của ngôn ngữ này. Do đó, trong đề tài này xét C# tập trung trong ngữ cảnh cụ thể là nền tảng .NET của Microsoft và trong các ứng dụng máy tính để bàn và ứng dụng Internet.

Chương này trình bày chung về hai phần là ngôn ngữ C# và nền tảng .NET, bao gồm cả khung ứng dụng .NET (.NET Framework)

1.1 Nền tảng của .NET

Khi Microsoft công bố C# vào tháng 7 năm 2000, việc khánh thành nó chỉ là một phần trong số rất nhiều sự kiện mà nền tảng .Net được công công bố. Nền tảng .Net là bô khung phát triển ứng dụng mới, nó cung cấp một giao diện lập trình ứng dụng (Application Programming Interface - API) mới mẽ cho các dịch vụ và hệ điều hành Windows, cụ thể là Windows 2000, nó cũng mang lại nhiều kỹ thuật khác nổi bật của Microsoft suốt từ những năm 90. Trong số đó có các dịch vụ COM+, công nghệ ASP, XML và thiết kế hướng đối tượng, hỗ trợ các giao thức dịch vụ web mới như SOAP, WSDL và UDDL với trọng tâm là Internet, tất cả được tích hợp trong kiến trúc DNA.

Nền tảng .NET bao gồm bốn nhóm sau:

1. Một tập các ngôn ngữ, bao gồm C# và Visual Basic .Net; một tập các công cụ phát triển bao gồm Visual Studio .Net; một tập đầy đủ các thư viện phục vụ cho việc xây dựng các ứng dụng web, các dịch vụ web và các ứng dụng Windows; còn có CLR - Common Language Runtime: (ngôn ngữ thực thi dùng chung) để thực thi các đối tượng được xây dựng trên bô khung này.
2. Một tập các Server Xí nghiệp .Net như SQL Server 2000. Exchange 2000, BizTalk 2000, … chúng cung cấp các chức năng cho việc lưu trữ dữ liệu quan hệ, thư điện tử, thương mại điện tử B2B, …

3. Các dịch vụ web thương mại miễn phí, vừa được công bố gần đây như là dự án Hailstorm; nhà phát triển có thể dùng các dịch vụ này để xây dựng các ứng dụng đòi hỏi tri thức về định danh người dùng…
4. .NET cho các thiết bị không phải PC như điện thoại (cell phone), thiết bị game

1.2 .NET Framework

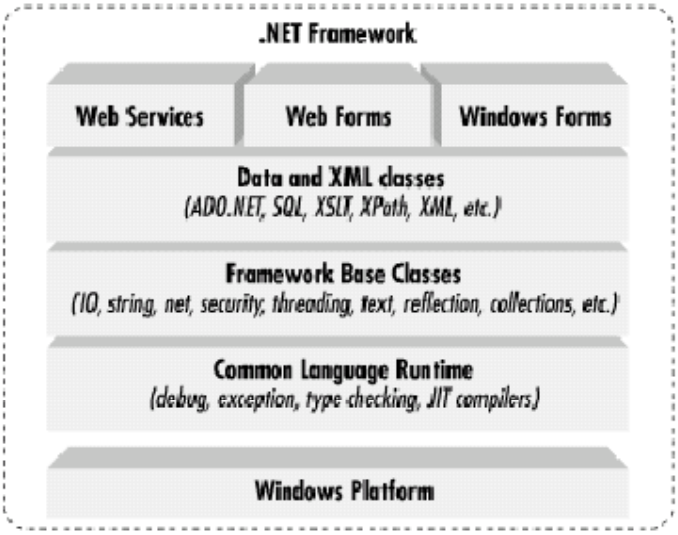
.Net hỗ trợ tích hợp ngôn ngữ, tức là ta có thể kế thừa các lớp, bắt các biệt lệ, đa hình thông qua nhiều ngôn ngữ. .NET Framework thực hiện được việc này nhờ vào đặc tả Common Type System - CTS (hệ thống kiểu chung) mà tất cả các thành phần .Net đều tuân theo. Ví dụ, mọi thứ trong .Net đều là đối tượng, thừa kế từ lớp gốc System.Object.

Ngoài ra .Net còn bao gồm Common Language Specification - CLS (đặc tả ngôn ngữ chung). Nó cung cấp các qui tắc cơ bản mà ngôn ngữ muốn tích hợp phải thỏa mãn. CLS chỉ ra các yêu cầu tối thiểu của ngôn ngữ hỗ trợ .Net. Trình biên dịch tuân theo CLS sẽ tạo các đối tượng có thể tương hợp với các đối tượng khác. Bộ thư viện lớp của khung ứng dụng (Framework Class Library - FCL) có thể được dùng bởi bất kỳ ngôn ngữ nào tuân theo CLS.

.NET Framework nằm ở tầng trên của hệ điều hành (bất kỳ hệ điều hành nào không chỉ là Windows). .NET Framework bao bao gồm:

- Bốn ngôn ngữ chính thức: C#, VB.Net, C++, và Jscript.NET
- Common Language Runtime - CLR, nền tảng hướng đối tượng cho phát triển ứng dụng Windows và web mà các ngôn ngữ có thể chia sẻ sử dụng.
- Bộ thư viện Framework Class Library - FCL.

Hình 1-1 Kiến trúc khung ứng dụng .Net



Thành phần quan trọng nhất của .NET Framework là CLR, nó cung cấp môi trường cho ứng dụng thực thi, CLR là một máy ảo, tương tự máy ảo Java. CLR kích hoạt đối tượng, thực hiện kiểm tra bảo mật, cấp phát bộ nhớ, thực thi và thu dọn chúng.

Trong Hình 1-1 tầng trên của CLR bao gồm:

- Các lớp cơ sở
- Các lớp dữ liệu và XML
- Các lớp cho dịch vụ web, web form, và Windows form.

Các lớp này được gọi chung là FCL, Framework Class Library, cung cấp API hướng đối tượng cho tất cả các chức năng của .NET Framework (hơn 5000 lớp).

Các lớp cơ sở tương tự với các lớp trong Java. Các lớp này hỗ trợ các thao tác nhập xuất, thao tác chuỗi, văn bản, quản lý bảo mật, truyền thông mạng, quản lý tiểu trình và các chức năng tổng hợp khác …

Trên mức này là lớp dữ liệu và XML. Lớp dữ liệu hỗ trợ việc thao tác các dữ liệu trên cơ sở dữ liệu. Các lớp này bao gồm các lớp SQL (Structure Query Language: ngôn ngữ truy vấn có cấu trúc) cho phép ta thao tác dữ liệu thông qua một giao tiếp SQL chuẩn. Ngoài ra còn một tập các lớp gọi là ADO.Net cũng cho phép thao tác dữ liệu. Lớp XML hỗ trợ thao tác dữ liệu XML, tìm kiếm và diễn dịch XML.

Trên lớp dữ liệu và XML là lớp hỗ trợ xây dựng các ứng dụng Windows (Windows forms), ứng dụng Web (Web forms) và dịch vụ Web (Web services).

1.3 Biên dịch và ngôn ngữ trung gian (MSIL)

Với .NET chương trình không biên dịch thành tập tin thực thi, mà biên dịch thành ngôn ngữ trung gian (MSIL - Microsoft Intermediate Language, viết tắt là IL), sau đó chúng được CLR thực thi. Các tập tin IL biên dịch từ C# đồng nhất với các tập tin IL biên dịch từ ngôn ngữ .Net khác.

Khi biên dịch dự án, mã nguồn C# được chuyển thành tập tin IL lưu trên đĩa. Khi chạy chương trình thì IL được biên dịch (hay thông dịch) một lần nữa bằng trình *Just In Time* - JIT, khi này kết quả là mã máy và bộ xử lý sẽ thực thi.

Trình biên dịch JIT chỉ chạy khi có yêu cầu. Khi một phương thức được gọi, JIT phân tích IL và sinh ra mã máy tối ưu cho từng loại máy. JIT có thể nhận biết mã nguồn đã được biên dịch chưa, để có thể chạy ngay ứng dụng hay phải biên dịch lại.

CLS có nghĩa là các ngôn ngữ .Net cùng sinh ra mã IL. Các đối tượng được tạo theo một ngôn ngữ nào đó sẽ được truy cập và thừa kế bởi các đối tượng của ngôn ngữ khác. Vì vậy ta có thể tạo được một lớp cơ sở trong VB.Net và thừa kế nó từ C#.

1.4 Ngôn ngữ C#

C# là một ngôn ngữ rất đơn giản, với khoảng 80 từ khoá và hơn mười kiểu dữ liệu dựng sẵn, nhưng C# có tính diễn đạt cao. C# hỗ trợ lập trình có cấu trúc, hướng đối tượng, hướng thành phần (component oriented).

Trọng tâm của ngôn ngữ hướng đối tượng là lớp. Lớp định nghĩa kiểu dữ liệu mới, cho phép mở rộng ngôn ngữ theo hướng cần giải quyết. C# có những từ khoá dành cho việc khai báo lớp, phương thức, thuộc tính (property) mới. C# hỗ trợ đầy đủ khái niệm trụ cột trong lập trình hướng đối tượng: đóng gói, thừa kế, đa hình.

Định nghĩa lớp trong C# không đòi hỏi tách rời tập tin tiêu đề với tập tin cài đặt như C++. Hơn thế, C# hỗ trợ kiểu sưu liệu mới, cho phép sưu liệu trực tiếp trong tập tin mã nguồn. Đến khi biên dịch sẽ tạo tập tin sưu liệu theo định dạng XML.

C# hỗ trợ khái niệm giao diện, *interfaces* (tương tự Java). Một lớp chỉ có thể kế thừa duy nhất một lớp cha nhưng có thế cài đặt nhiều giao diện.

C# có kiểu *cấu trúc,* struct (không giống C++). Cấu trúc là kiểu hạng nhẹ và bị giới hạn.Cấu trúc không thể thừa kế lớp hay được kế thừa nhưng có thể cài đặt giao diện.

C# cung cấp những đặc trưng lập trình hướng thành phần như property, sự kiện và dẫn hướng khai báo (được gọi là *attribute*). Lập trình hướng component được hỗ trợ bởi CLR thông qua siêu dữ liệu (metadata). Siêu dữ liệu mô tả các lớp bao gồm các phương thức và thuộc tính, các thông tin bảo mật ….

Assembly là một tập hợp các tập tin mà theo cách nhìn của lập trình viên là các thư viện liên kết động (DLL) hay tập tin thực thi (EXE). Trong .NET một assembly là một đơn vị của việc tái sử dụng, xác định phiên bản, bảo mật, và phân phối. CLR cung cấp một số các lớp để thao tác với assembly.

C# cũng cho truy cập trực tiếp bộ nhớ dùng con trỏ kiểu C++, nhưng vùng mã đó được xem như không an toàn. CLR sẽ không thực thi việc thu dọn rác tự động các đối tượng được tham chiếu bởi con trỏ cho đến khi lập trình viên tự giải phóng.

Chương 2 Khởi đầu

Chương này ta sẽ tạo, biên dịch và chạy chương trình "Hello World" bằng ngôn ngữ C#. Phân tích ngắn gọn chương trình để giới thiệu các đặc trưng chính yếu trong ngôn ngữ C#.

Ví dụ 2-1 Chương trình Hello World

```
class HelloWorld
{
   static void Main( )
   {
      // sử dụng đối tượng console của hệ thống
      System.Console.WriteLine("Hello World");
   }
}
```

Sau khi biên dịch và chạy HelloWorld, kết quả là dòng chữ "Hello World" hiển thị trên màn hình.

2.1 Lớp, đối tượng và kiểu

Bản chất của lập trình hướng đối tượng là tạo ra các kiểu mới. Một *kiểu* biểu diễn một vật gì đó. Giống với các ngôn ngữ lập trình hướng đối tượng khác, một kiểu trong C# cũng định nghĩa bằng từ khoá *class* (và được gọi là lớp) còn thể hiện của lớp được gọi là *đối tượng*.

Xem Ví dụ 2-1 ta thấy cách khai báo một lớp HelloWorld. Ta thấy ngay là cách khai báo và nội dung của một lớp hoàn toàn giống với ngôn ngữ Java và C++, chỉ có khác là cuối khai báo lớp không cần dấu ";"

2.1.1 Phương thức

Các hành vi của một lớp được gọi là các phương thức thành viên (gọi tắt là phương thức) của lớp đó. Một *phương thức* là một *hàm* (phương thức thành viên còn gọi là hàm thành viên). Các phương thức định nghĩa những gì mà một lớp có thể làm.

Cách khai báo, nội dung và cách sử dụng các phương thức giống hoàn toàn với Java và C++. Trong ví dụ trên có một phương thức đặc biệt là phương thức Main() (như hàm main() trong C++) là phương thức bắt đầu của một ứng dụng C#, có thể trả về kiểu void hay int. Mỗi một chương trình (assembly) có thể có nhiều phương thức Main nhưng khi đó phải chỉ định phương thức Main() nào sẽ bắt đầu chương trình.

2.1.2 Các ghi chú

C# có ba kiểu ghi chú trong đó có hai kiểu rất quen thuộc của C++ là dùng: "//" và "/* … */". Ngoài ra còn một kiểu ghi chú nữa sẽ trình bày ở các chương kế.

Ví dụ 2-2 Hai hình thức ghi chú trong C#

```
class HelloWorld
{
  static void Main( ) // Đây là ghi trên một dòng
  {
    /* Bắt đầu ghi chú nhiều dòng
       Vẫn còn trong ghi chú
       Kết thúc ghi chú bằng */
    System.Console.WriteLine("Hello World");
  }
}
```

2.1.3 Ứng dụng dạng console

"Hello World" là một ứng dụng console. Các ứng dụng dạng này thường không có giao diện người dùng đồ họa Các nhập xuất đều thông qua các console chuẩn (dạng dòng lệnh như DOS).

Trong ví dụ trên, phương thức Main() viết ra màn hình dòng "Hello World". Do màn hình quản lý một đối tượng Console, đối tượng này có phương thức WriteLine() cho phép đặt một dòng chữ lên màn hình. Để gọi phương thức này ta dùng toán tử ".", như sau: Console.WriteLine(…).

2.1.4 Namespaces - Vùng tên

Console là một trong rất nhiều (cả ngàn) lớp trong bộ thư viện .NET. Mỗi lớp đều có tên và như vậy có hàng ngàn tên mà lập trình viên phải nhớ hoặc phải tra cứu mỗi khi sử dụng. Vấn đề là phải làm sao giảm bớt lượng tên phải nhớ.

Ngoài vấn đề phải nhớ quá nhiều tên ra, còn một nhận xét sau: một số lớp có mối liên hệ nào đó về mặt ngữ nghĩa, ví dụ như lớp Stack, Queue, Hashtable … là các lớp cài đặt cấu trúc dữ liệu túi chứa. Như vậy có thể nhóm những lớp này thành một nhóm và thay vì phải nhớ tên các lớp thì lập trình viên chỉ cần nhớ tên nhóm, sau đó có thể thực hiện việc tra cứu tên lớp trong nhóm nhanh chóng hơn. Nhóm là một *vùng tên* trong C#.

Một vùng tên có thể có nhiều lớp và vùng tên khác. Nếu vùng tên A nằm trong vùng tên B, ta nói vùng tên A là vùng tên con của vùng tên B. Khi đó các lớp trong vùng tên A được ghi như sau: `B.A.Tên_lớp_trong_vùng_tên_A`

System là vùng tên chứa nhiều lớp hữu ích cho việc giao tiếp với hệ thống hoặc các lớp công dụng chung như lớp Console, Math, Exception….Trong ví dụ HelloWorld trên, đối tượng Console được dùng như sau:

```
System.Console.WriteLine("Hello World");
```

2.1.5 Toán tử chấm "."

Như trong Ví dụ 2-1 toán tử chấm được dùng để truy suất dữ liệu và phương thức một lớp (như Console.WriteLine()), đồng thời cũng dùng để chỉ định tên lớp trong một vùng tên (như System.Console).

Toán tử dấu chấm cũng được dùng để truy xuất các vùng tên con của một vùng tên

```
Vùng_tên.Vùng_tên_con.Vùng_tên_con_con
```

2.1.6 Từ khoá using

Nếu chương trình sử dụng nhiều lần phương thức Console.WriteLine, từ System sẽ phải viết nhiều lần. Điều này có thể khiến lập trình viên nhàm chán. Ta sẽ khai báo rằng chương trình có sử dụng vùng tên System, sau đó ta dùng các lớp trong vùng tên System mà không cần phải có từ System đi trước.

Ví dụ 2-3 Từ khóa using

```
// Khai báo chương trình có sử dụng vùng tên System
using System;

class HelloWorld
{
  static void Main( )
  {
    // Console thuộc vùng tên System
    Console.WriteLine("Hello World");
  }
}
```

2.1.7 Phân biệt hoa thường

Ngôn ngữ C# cũng phân biệt chữ hoa thường giống như Java hay C++ (không như VB). Ví dụ như WriteLine khác với writeLine và cả hai cùng khác với WRITELINE. Tên biến, hàm, hằng … đều phân biệt chữ hoa chữ thường.

2.1.8 Từ khoá static

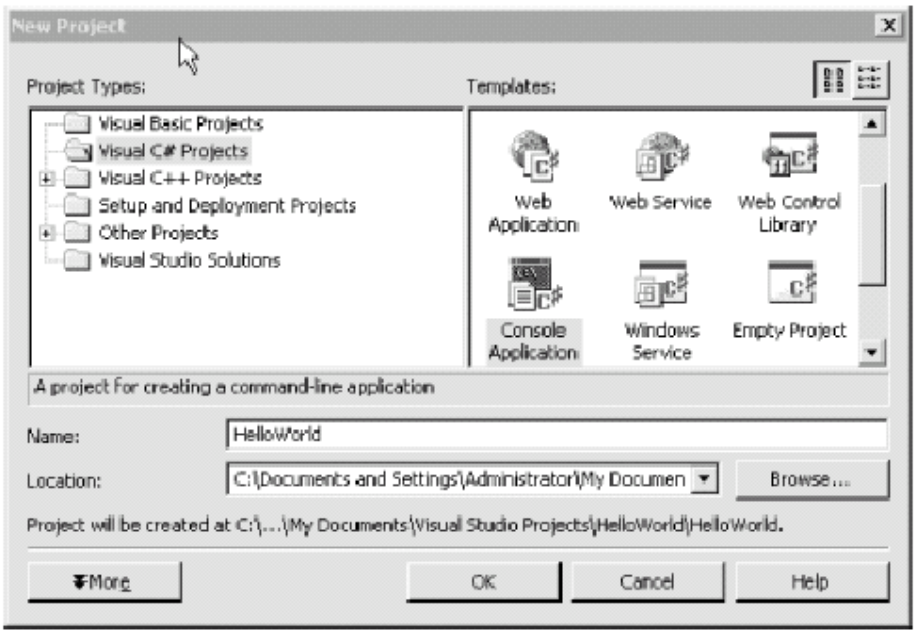
Trong Ví dụ 2-1 phương thức Main() được khai báo kiểu trả về là void và dùng từ khoá **static**. Từ khoá static cho biết là ta có thể gọi phương thức Main() mà không cần tạo một đối tượng kiểu HelloWorld.

2.2 Phát triển "Hello World"

Có hai cách để viết, biên dịch và chạy chương trình HelloWorld là dùng môi trường phát triển tích hợp (IDE) Visual Studio .Net hay viết bằng trình soạn thảo văn bản và biên dịch bằng dòng lệnh. IDE Vs.Net dễ dùng hơn. Do đó, trong đề tài này chỉ trình bày theo hướng làm việc trên IDE Visual Studio .Net.

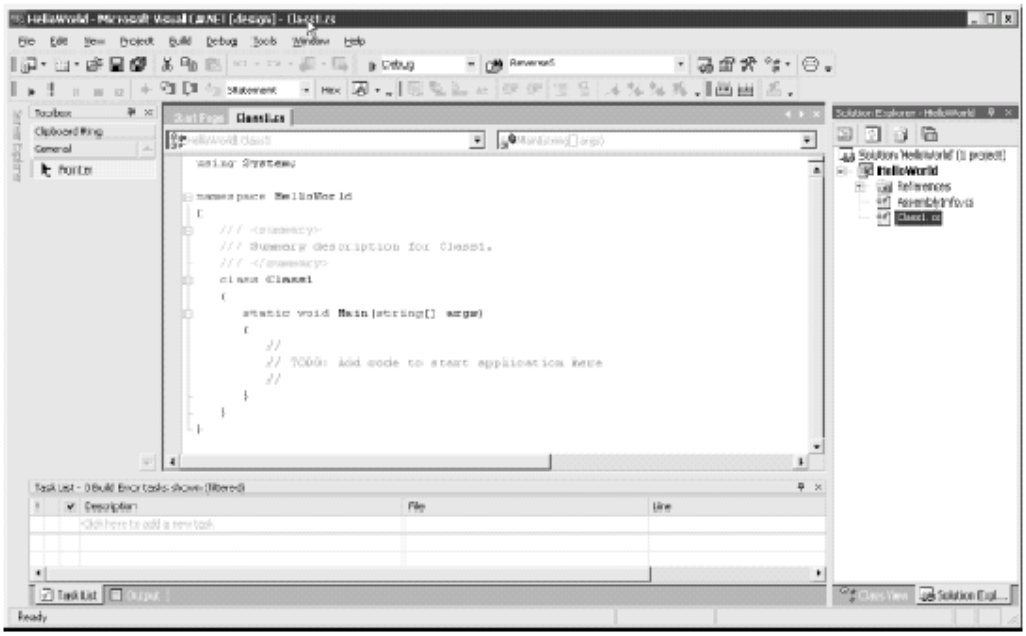
2.2.1 Soạn thảo "Hello World"

Để tạo chương trình "Hello World" trong IDE, ta chọn Visual Studio .Net từ thanh thực đơn. Tiếp theo trên màn hình của IDE chọn *File > New > Project* từ thanh thực đơn, theo đó xuất hiện một cửa sổ như sau:



Hình 2-1 Tạo một ứng dụng console trong VS.Net

Để tạo chương trình "Hello World" ta chọn Visual *C# Project > Console Application*, điền *HelloWorld* trong ô Name, chọn đường dẫn và nhấn OK. Một cửa sổ soạn thảo xuất hiện.



Hình 2-2 Cửa sổ soạn thảo nội dung mã nguồn

Vs.Net tự tạo một số mã, ta cần chỉnh sửa cho phù hợp với chương trình của mình.

2.2.2 Biên dịch và chạy "Hello World"

Sau khi đã đầy đủ mã nguồn ta tiến hành biên dịch chương trình: nhấn "Ctrl–Shift–B" hay chọn *Build > Build Solution.* Kiểm tra xem chương trình có lỗi không ở của sổ *Output* cuối màn hình. Khi biên dịch chương trình nó sẽ lưu lại thành tập tin *.cs*.

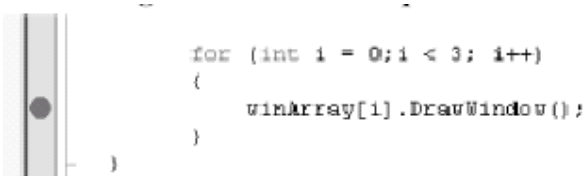
Chạy chương trình bằng "Ctrl–F5" hay chọn *Debug > Start Without Debugging*.

2.2.3 Trình gở rối của Visual Studio .Net

Trình gỡ rối của VS.Net rất mạnh hữu ích. Ba kỹ năng chính yếu để sử dụng của trình gở rối là:

- Cách đặt điểm ngắt (breakpoint) và làm sao chạy cho đến điểm ngắt
- Làm thế nào chạy từng bước và chạy vượt qua một phương thức.
- Làm sao để quan sát và hiệu chỉnh giá trị của biến, dữ liệu thành viên, …

Cách đơn giản nhất để đặt điểm ngắt là bấm chuột trái vào phía lề trái, tại đó sẽ hiện lên một chấm đỏ.



Hình 2-3 Minh họa một điểm ngắt

Cách dùng trình gở rối hoàn toàn giống với trình gở rối trong VS 6.0. Nó cho phép ta dừng lại ở một vị trí bất kỳ, cho ta kiểm tra giá trị tức thời bằng cách di chuyển chuột đến vị trị biến. Ngoài ra, khi gở rối ta cũng có thể xem giá trị các biến thông qua cửa sổ *Watch* và *Local*.

Để chạy trong chế độ gở rối ta chọn *Debug* → *Start* hay nhấn **F5**, muốn chạy từng bước ta bấm **F11** và chạy vượt qua một phương thức ta bấm **F10**.

Chương 3 Những cơ sở của ngôn ngữ C#

Trong chương này sẽ trình bày về hệ thống kiểu trong C#; phân biệt kiểu dựng sẵn (*int, long, bool, ...*) với các kiểu do người dùng định nghĩa. Ngoài ra, chương này cũng sẽ trình bày cách tạo và dùng biến, hằng; giới thiệu kiểu liệt kê, chuỗi, kiểu định danh, biểu thức, và câu lệnh. Phần hai của chương trình bày về các cấu trúc điều kiện và các toán tử logic, quan hệ, toán học, …

3.1 Các kiểu

C# buộc phải khai báo kiểu của đối tượng được tạo. Khi kiểu được khai báo rõ ràng, trình biên dịch sẽ giúp ngăn ngừa lỗi bằng cách kiểm tra dữ liệu được gán cho đối tượng có hợp lệ không, đồng thời cấp phát **đúng** kích thước bộ nhớ cho đối tượng.

C# phân thành hai loại: loai dữ liệu dựng sẵn và loại do người dùng định nghĩa.

C# cũng chia tập dữ liệu thành hai kiểu: *giá trị* và *tham chiếu*. Biến kiểu giá trị được lưu trong vùng nhớ stack, còn biến kiểu tham chiếu được lưu trong vùng nhớ heap.

C# cũng hỗ trợ kiểu con trỏ của C++, nhưng ít khi được sử dụng. Thông thường con trỏ chỉ được sử dụng khi làm việc trực tiếp với Win API hay các đối tượng COM.

3.1.1 Loại dữ liệu định sẵn

C# có nhiểu kiểu dữ liệu định sẵn, mỗi kiểu ánh xạ đến một kiểu được hổ trợ bởi CLS (Commom Language Specification), ánh xạ để đảm bảo rằng đối tượng được tạo trong C# không khác gì đối tượng được tạo trong các ngôn ngữ .NET khác Mỗi kiểu có một kích thước cố định được liệt kê trong bảng sau

Bảng 3-1 Các kiểu dựng sẵn

Kiểu	Kích thước (byte)	Kiểu .Net	Mô tả - giá trị
byte	1	Byte	Không dấu (0..255)
char	1	Char	Mã ký thự Unicode
bool	1	Boolean	true hay false
sbyte	1	Sbyte	Có dấu (-128 .. 127)
short	2	Int16	Có dấu (-32768 .. 32767)
ushort	2	Uint16	Không dấu (0 .. 65535)
int	4	Int32	Có dấu (-2147483647 .. 2147483647)

uint	4	Uint32	Không dấu (0 .. 4294967295)
float	4	Single	Số thực (≈ ±1.5*10-45 .. ≈ ±3.4*1038)
double	8	Double	Số thực (≈ ±5.0*10-324 .. ≈ ±1.7*10308)
decimal	8	Decimal	số có dấu chấm tĩnh với 28 ký số và dấu chấm
long	8	Int64	Số nguyên có dấu (- 9223372036854775808 .. 9223372036854775807)
ulong	8	Uint64	Số nguyên không dấu (0 .. 0xffffffffffffffff.)

3.1.1.1 Chọn một kiểu định sẵn

Tuỳ vào từng giá trị muốn lưu trữ mà ta chọn kiểu cho phù hợp. Nếu chọn kiểu quá lớn so với các giá trị cần lưu sẽ làm cho chương trình đòi hỏi nhiều bộ nhớ và chạy chậm. Trong khi nếu giá trị cần lưu lớn hơn kiểu thực lưu sẽ làm cho giá trị các biến bị sai và chương trình cho kết quả sai.

Kiểu *char* biểu diễn một ký tự Unicode. Ví dụ "\u0041" là ký tự "A" trên bảng Unicode. Một số ký tự đặc biệt được biểu diễn bằng dấu "\" trước một ký tự khác.

Bảng 3-2 Các ký tự đặc biệt thông dụng

Ký tự	Nghĩa
\'	**dầu nháy đơn**
\"	**dấu nháy đôi**
\\	**dấu chéo ngược "\"**
\0	**Null**
\a	**Alert**
\b	**lùi về sau**
\f	**Form feed**
\n	**xuống dòng**
\r	**về đầu dòng**
\t	**Tab ngang**
\v	**Tab dọc**

3.1.1.2 Chuyển đổi kiểu định sẵn

Một đối tượng có thể chuyển từ kiểu này sang kiểu kia theo hai hình thức: ngầm hoặc tường minh. Hình thức ngầm được chuyển tự động còn hình thức tường minh cần sự can thiệp trực tiếp của người lập trình (giống với C++ và Java).

```
short x = 5;
int y ;
y = x; // chuyển kiểu ngầm định - tự động
x = y; // lỗi, không biên dịch được
x = (short) y; // OK
```

3.2 Biến và hằng

Biến dùng để lưu trữ dữ liệu. Mỗi biến thuộc về một kiểu dữ liệu nào đó.

3.2.1 Khởi tạo trước khi dùng

Trong C#, trước khi dùng một biến thì biến đó phải được khởi tạo nếu không trình biên dịch sẽ báo lỗi khi biên dịch. Ta có thể khai báo biến trước, sau đó khởi tạo và sử dụng; hay khai báo biến và khởi gán trong lúc khai báo.

```
int x; // khai báo biến trước
x = 5; // sau đó khởi gán giá trị và sử dụng

int y = x; // khai báo và khởi gán cùng lúc
```

3.2.2 Hằng

Hằng là một biến nhưng giá trị không thay đổi theo thời gian. Khi cần thao tác trên một giá trị xác định ta dùng hằng. Khai báo hằng tương tự khai báo biến và có thêm từ khóa const ở trước. Hằng một khi khởi động xong không thể thay đổi được nữa.

```
const int HANG_SO = 100;
```

3.2.3 Kiểu liệt kê

Enum là một cách thức để đặt tên cho các trị nguyên (các trị kiểu số nguyên, theo nghĩa nào đó tương tự như tập các hằng), làm cho chương trình rõ ràng, dễ hiểu hơn. Enum không có hàm thành viên. Ví dụ tạo một enum tên là Ngay như sau:

```
enum Ngay {Hai, Ba, Tu, Nam, Sau, Bay, ChuNhat};
```

Theo cách khai báo này enum ngày có bảy giá trị nguyên đi từ 0 = Hai, 1 = Ba, 2 = Tư … 7 = ChuNhat.

Ví dụ 3-1 Sử dụng enum Ngay

```
using System;
public class EnumTest
{
   enum Ngay {Hai, Ba, Tu, Nam, Sau, Bay, ChuNhat };

   public static void Main()
   {
      int x = (int) Ngay.Hai;
      int y = (int) Ngay.Bay;
      Console.WriteLine("Thu Hai = {0}", x);
      Console.WriteLine("Thu Bay = {0}", y);
   }
}

Kết quả
Thu Hai = 0
Thu Bay = 5
```

Mặc định enum gán giá trị đầu tiên là 0 các trị sau lớn hơn giá trị trước một đơn vị, và các trị này thuộc kiểu int. Nếu muốn thay đổi trị mặc định này ta phải gán trị mong muốn.

Ví dụ 3-2 Sử dụng enum Ngay (2)

```
using System;
namespace ConsoleApplication
{
  enum Ngay: byte { Hai=2,Ba,Tu,Nam,Sau,Bay,ChuNhat=10 };
  class EnumTest
  {
    static void Main(string[] args)
    {
      byte x = (byte)Ngay.Ba;
      byte y = (byte)Ngay.ChuNhat;
      Console.WriteLine("Thu Ba = {0}", x);
      Console.WriteLine("Chu Nhat = {0}", y);
      Console.Read();
    }
  }
}
Kết quả:
Thu Ba = 3
Chu Nhat = 10
```

Kiểu enum ngày được viết lại với một số thay đổi, giá trị cho Hai là 2, giá trị cho Ba là 3 (Hai + 1) …, giá trị cho ChuNhat là 10, và các giá trị này sẽ là kiểu byte.

Cú pháp chung cho khai báo một kiểu enum như sau

```
[attributes] [modifiers] enum identifier [:base-type]
{
  enumerator-list
};
attributes (tùy chọn): các thông tin thêm (đề cập sau)
modifiers (tùy chọn): public, protected, internal, private
                      (các bổ từ xác định phạm vi truy xuất)
identifer: tên của enum
base_type (tùy chọn): kiểu số, ngoại trừ char
enumerator-list: danh sách các thành viên.
```

3.2.4 Chuỗi

Chuỗi là kiểu dựng sẵn trong C#, nó là một chuỗi các ký tự đơn lẻ. Khi khai báo một biến chuỗi ta dùng từ khoá *string*. Ví dụ khai báo một biến string lưu chuỗi "Hello World"

```
string myString = "Hello World";
```

3.2.5 Định danh

Định danh là tên mà người lập trình chọn đại diện một kiểu, phương thức, biến, hằng, đối tượng… của họ. Định danh **phải** bắt đầu bằng một ký tự hay dấu "_". Định danh không được trùng với từ khoá C# và phân biệt hoa thường.

3.3 Biểu thức

Bất kỳ câu lệnh định lượng giá trị được gọi là một biểu thức (*expression)*. Phép gán sau cũng được gọi là một biểu thức vì nó định lượng giá trị được gán (là 32)

```
x = 32;
```

vì vậy phép gán trên có thể được gán một lần nữa như sau

```
y = x = 32;
```

Sau lệnh này y có giá trị của biểu thức x = 32 và vì vậy y = 32.

3.4 Khoảng trắng

Trong C#, khoảng trống, dấu tab, dấu xuống dòng đều được xem là khoảng trắng (*whitespace*). Do đó, dấu cách dù lớn hay nhỏ đều như nhau nên ta có:

```
x = 32;
```

cũng như

```
x     =                    32;
```

Ngoại trừ khoảng trắng trong chuỗi ký tự thì có ý nghĩa riêng của nó.

3.5 Câu lệnh

Cũng như trong C++ và Java một chỉ thị hoàn chỉnh thì được gọi là một câu lệnh (*statement)*. Chương trình gồm nhiều câu lệnh, mỗi câu lệnh kết thúc bằng dấu ";". Ví dụ:

```
int x; // là một câu lệnh
x = 23; // một câu lệnh khác
```

Ngoài các câu lệnh bình thường như trên, có các câu lệnh khác là: lệnh rẽ nhánh không điều kiện, rẽ nhánh có điều kiện và lệnh lặp.

3.5.1 Các lệnh rẽ nhánh không điều kiện

Có hai loại câu lệnh rẽ nhánh không điều kiện. Một là lệnh gọi phương thức: khi trình biên dịch thấy có lời gọi phương thức nó sẽ tạm dừng phương thức hiện hành và nhảy đến phương thức được gọi cho đến hết phương thức này sẽ trở về phương thức cũ.

Ví dụ 3-3 Gọi một phương thức

```
using System;
class Functions
{
  static void Main( )
  {
    Console.WriteLine("In Main! Calling SomeMethod( )...");
    SomeMethod( );
    Console.WriteLine("Back in Main( ).");
  }
  static void SomeMethod( )
  {
```

```
      Console.WriteLine("Greetings from SomeMethod!");
    }
}
Kết quả:
In Main! Calling SomeMethod( )...
Greetings from SomeMethod!
Back in Main( ).
```

Cách thứ hai để tạo các câu lệnh rẽ nhánh không điều kiện là dùng từ khoá: *goto, break, continue, return,* hay *throw*. Cách từ khóa này sẽ được giới thiệu trong các phần sau.

3.5.2 Lệnh rẽ nhánh có điều kiện

Các từ khóa if-else, while, do-while, for, switch-case, dùng để điều khiển dòng chảy chương trình. C# giữ lại tất cả các cú pháp của C++, ngoại trừ switch có vài cải tiến.

3.5.2.1 Lệnh If .. else …

Cú pháp:

```
if ( biểu thức logic )
  khối lệnh;
```

hoặc

```
if ( biểu thức logic )
  khối lệnh 1;
else
  khối lệnh 2;
```

Ghi chú: Khối lệnh *là một tập các câu lện trong cặp dấu "{...}". Bất kỳ nơi đâu có câu lệnh thì ở đó có thể viết bằng một khối lệnh.*

Biểu thức logic là biểu thức cho giá trị đúng hoặc sai (true hoặc false). Nếu "biểu thức logic" cho giá trị đúng thì "khối lệnh" hay "khối lệnh 1" sẽ được thực thi, ngược lại "khối lệnh 2" sẽ thực thi. Một điểm khác biệt với C++ là biểu thức trong câu lệnh if phải là biểu thức logic, không thể là biểu thức số.

3.5.2.2 Lệnh switch

Cú pháp:

```
switch ( biểu_thức_lựa_chọn )
{
  case biểu_thức_hằng :
    khối lệnh;
    lệnh nhảy;
  [ default :
    khối lệnh;
    lệnh nhảy; ]
}
```

Biểu thức lựa chọn là biểu thức sinh ra trị nguyên hay chuỗi. Switch sẽ so sánh biểu_thức_lựa_chọn với các biểu_thức_hằng để biết phải thực hiện với khối lệnh nào. Lệnh nhảy như break, goto…để thoát khỏi câu switch và bắt buộc phải có.

```
int nQuyen = 0;
switch ( sQuyenTruyCap )
{
  case "Administrator":
    nQuyen = 1;
    break;
  case "Admin":
    goto case "Administrator";
  default:
    nQuyen = 2;
  break;
}
```

3.5.3 Lệnh lặp

C# cung cấp các lệnh lặp giống C++ như *for, while, do-while* và lệnh lặp mới *foreach*. Nó cũng hỗ trợ các câu lệnh nhảy như: *goto, break, continue* và *return*.

3.5.3.1 Lệnh goto

Lệnh *goto* có thể dùng để tạo lệnh nhảy nhưng nhiều nhà lập trình chuyên nghiệp khuyên không nên dùng câu lệnh này vì nó phá vỡ tính cấu trúc của chương trình. Cách dùng câu lệnh này như sau: (giống như trong C++)

1. Tạo một nhãn
2. goto đến nhãn đó.

3.5.3.2 Vòng lặp while

Cú pháp:

```
while ( biểu_thức_logic )
  khối_lệnh;
```

Khối_lệnh sẽ được thực hiện cho đến khi nào biểu thức còn đúng. Nếu ngay từ đầu biểu thức sai, khối lệnh sẽ không được thực thi.

3.5.3.3 Vòng lặp do … while

Cú pháp:

```
do
  khối_lệnh
while ( biểu_thức_logic )
```

Khác với while khối lệnh sẽ được thực hiện trước, sau đó biệu thức được kiểm tra. Nếu biểu thức đúng khối lệnh lại được thực hiện.

3.5.3.4 Vòng lặp for

Cú pháp:

```
for ( [khởi_tạo_biến_đếm]; [biểu_thức]; [gia_tăng_biến_đếm] )
  khối lệnh;
```

Ví dụ 3-4 Tính tổng các số nguyên từ a đến b

```
int a = 10; int b = 100; int nTong = 0;
```

```
for ( int i = a; i <= b; i++ )
{
  nTong += i;
}
```

Câu lệnh lặp *foreach* sẽ được trình bày ở các chương sau.

3.5.3.5 Câu lệnh break, continue, và return

Cả ba câu lệnh *break, continue,* và *return* rất quen thuộc trong C++ và Java, trong C#, ý nghĩa và cách sử dụng chúng hoàn toàn giống với hai ngôn ngữ này.

3.6 Toán tử

Các phép toán +, -, *, / là một ví dụ về toán tử. Áp dụng các toán tử này lên các biến kiểu số ta có kết quả như việc thực hiện các phép toán thông thường.

```
int a = 10;
int b = 20;
int c = a + b; // c = 10 + 20 = 30
```

C# cung cấp cấp nhiều loại toán tử khác nhau để thao tác trên các kiểu biến dữ liệu, được liệt kê trong bảng sau theo từng nhóm ngữ nghĩa.

Bảng 3-3 Các nhóm toán tử trong C#

Nhóm toán tử	Toán tử	Ý nghĩa
Toán học	+ - * / %	cộng , trừ, nhân chia, lấy phần dư
Logic	& | ^ ! ~ && || true false	phép toán logic và thao tác trên bit
Ghép chuỗi	+	ghép nối 2 chuỗi
Tăng, giảm	++, --	tăng / giảm toán hạng lên / xuống 1. Đứng trước hoặc sau toán hạng.
Dịch bit	<< >>	dịch trái, dịch phải
Quan hệ	== != < > <= >=	bằng, khác, nhỏ/lớn hơn, nhỏ/lớn hơn hoặc bằng
Gán	= += -= *= /= %= &= |= ^= <<= >>=	phép gán
Chỉ số	[]	cách truy xuất phần tử của mảng
Ép kiểu	()	
Indirection và Address	* -> [] &	dùng cho con trỏ

3.6.1 Toán tử gán (=)

Toán tử này cho phép thay đổi các giá trị của biến bên phải toán tử bằng giá trị bên trái toán tử.

3.6.2 Nhóm toán tử toán học

C# dùng các toàn tử số học với ý nghĩa theo đúng tên của chúng như: + (cộng), – (trừ) , * (nhân) và / (chia). Tùy theo kiểu của hai toán hạng mà toán tử trả về kiểu tương ứng. Ngoài ra, còn có toán tử *%* (lấy phần dư) được sử dụng trong các kiểu số nguyên.

3.6.3 Các toán tử tăng và giảm

C# cũng kế thừa từ C++ và Java các toán tử: `+=,-=, *=, /= , %=` nhằm làm đơn giản hoá. Nó còn kế thừa các toán tử *tiền tố* và *hậu tố* (như *biến*++, hay ++*biến*) để giảm bớt sự cồng kềnh trong các toán tử cổ điển.

3.6.4 Các toán tử quan hệ

Các toán tử quan hệ được dùng để so sánh hai giá trị với nhau và kết quả trả về có kiểu Boolean. Toán tử quan hệ gồm có: == (so sánh bằng), != (so sánh khác), > (so sánh lớn hơn), >= (lớn hơn hay bằng), < (so sánh nhỏ hơn), <= (nhỏ hơn hay bằng).

3.6.5 Các toán tử logic

Các toán tử logic gồm có: *&&* (*và*), || (*hoặc*), ! (*phủ định*). Các toán tử này được dùng trong các biểu thức điều kiện để kết hợp các toán tử quan hệ theo một ý nghĩa nhất định.

3.6.6 Thứ tự các toán tử

Đối với các biểu thức toán, thứ tự ưu tiên là thứ tự được qui định trong toán học. Còn thứ tự ưu tiên thực hiện của các nhóm toán tử được liệt kê theo bảng dưới đây

Bảng 3-4 Thứ tự ưu tiên của các nhóm toán tử (chiều ưu tiên từ trên xuống)

Nhóm toán tử	`Toán tử`	Ý nghĩa
Primary (chính)	`{x} x.y f(x) a[x] x++ x--`	
Unary	`+ - ! ~ ++x –x (T)x`	
Nhân	`* / %`	Nhân, chia, lấy phần dư
Cộng	`+ -`	cộng, trù
Dịch bít	`<< >>`	Dịch trái, dịch phải
Quan hệ	`< > <= >= is`	nhỏ hơn, lớn hơn, nhỏ hơn hay bằng, lớn hơn hay bằng và *là*
Bằng	`== !=`	bằng, khác
Logic trên bit AND	`&`	*Và* trên bit.
XOR	`^`	Xor trên bit
OR	`|`	*hoặc* trên bit

Điều kiện AND	`&&`	*Và* trên biểu thức điều kiện
Điều kiện OR	`||`	*Hoặc* trên biểu thức điều kiện
Điều kiện	`?:`	*điều kiện* tương tự *if*
Assignment	`= *= /= %= += -= <<= =>> &= ^= |=`	

3.6.7 Toán tử tam phân

Cú pháp:

```
<biểu thức điều kiện>? <biểu thức 1>: <biểu thức 2>;
```

Ý nghĩa:

```
Nếu biểu thức điều kiện đúng thì thực hiện biểu thức 1.
Nếu sai thì thực hiện biểu thức 2.
```

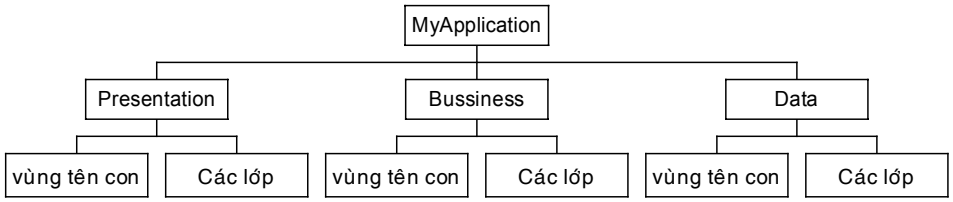
3.7 Tạo vùng tên

Như đã có giải thích trong phân tích ví dụ HelloWorld, vùng tên là một cách tổ chức mã nguồn thành các nhóm có ngữ nghĩa liên quan. Ví dụ:

Trong mô hình kiến trúc 3 lớp (3 tầng, tiếng Anh là 3 – tier Architecture) chia một ứng dụng ra thành 3 tầng: tầng giao diện, tầng nghiệp vụ và tầng dữ liệu (Presentation, Bussiness và Data). Ta có thể chia dự án thành 3 vùng tên tương ứng: Presentation, Bussiness và Data. Các vùng tên này chứa các lớp thuộc về tầng của mình.

Một vùng tên chứa các lớp và các vùng tên con khác. Vậy trong ví dụ trên ta sẽ tạo một vùng tên chung cho ứng dụng là MyApplication và ba vùng tên kia sẽ là ba vùng tên con của vùng tên MyApplication. Cách này giải quyết được trường hợp nếu ta có nhiều dự án mà chỉ có 3 vùng tên và dẫn đến việc không biết một lớp thuộc vùng tên Data nhưng không biết thuộc dự án nào.

Sơ đồ cây vùng tên



Vùng tên con được truy xuất thông qua tên vùng tên cha cách nhau bằng dấu chấm.

Để khai báo vùng tên ta sử dụng từ khóa namespace. Ví dụ dưới đây là 2 cách khai báo các vùng tên trong ví dụ ở trên.

Cách 1

```
namespace MyApplication
{
  namespace Presentation
  {
    // khai báo lớp
    // khai báo vùng tên con
  }
  namespace Bussiness
  {
    // khai báo lớp
    // khai báo vùng tên con
  }
  namespace Data
  {
    // khai báo lớp
    // khai báo vùng tên con
  }
}
```

Cách 2

```
namespace MyApplication.Presentation
{
  // khai báo lớp
  // khai báo vùng tên con
}
namespace MyApplication.Bussiness
{
  // khai báo lớp
  // khai báo vùng tên con
}
namespace MyApplication.Data
{
  // khai báo lớp
  // khai báo vùng tên con
}
```

Cách khai báo vùng tên thứ nhất chỉ tiện nếu các vùng tên nằm trên cùng một tập tin. Cách thứ hai tiện lợi hơn khi các vùng tên nằm trên nhiều tập tin khác nhau.

3.8 Chỉ thị tiền xử lý

Không phải mọi câu lệnh đều được biên dịch cùng lúc mà có một số trong chúng được biên dịch trước một số khác. Các câu lệnh như thế này gọi là các *chỉ thị tiền xử lý*. Các chỉ thị tiền xử lý được đặt sau dấu #.

3.8.1 Định nghĩa các định danh

`#define DEBUG` định nghĩa một *định danh tiền xử lý* (preprocessor identifier) `DEBUG`. Mặc dù các chỉ thị tiền xử lý có thể định nghĩa ở đâu tuỳ thích nhưng định danh tiền xử lý bắt buộc phải định nghĩa ở đầu của chương trình, trước cả từ khóa using. Do đó, ta cần trình bày như sau:

```
#define DEBUG
//... mã nguồn bình thường - không ảnh hưởng bởi bộ tiền xử lý
```

```
#if DEBUG
// mã nguồn được bao gồm trong chương trình
// khi chạy dưới chế độ debug
#else
// mã nguồn được bao gồm trong chương trình
// khi chạy dưới chế độ không debug
#endif
//... các đoạn mã nguồn không ảnh hưởng tiền xử lý
```

Trình biên dịch nhảy đến các đoạn thoả điều kiện tiền biên dịch để biên dịch trước.

3.8.2 Hủy một định danh

Ta hủy một định danh bằng cách dùng `#undef`. Bộ tiền xử lý duyệt mã nguồn từ trên xuống dưới, nên định danh được định nghĩa từ `#define`, hủy khi gặp `#undef` hay đến hết chương trình. Ta sẽ viết là:

```
#define DEBUG
  #if DEBUG
  // mã nguồn được biên dịch
  #endif
#undef DEBUG

#if DEBUG
  // mã nguồn sẽ không được biên dịch
#endif
```

3.8.3 #if, #elif, #else và #endif

Đây là các chỉ thị để chọn lựa xem có tiền biên dịch hay không. Các chỉ thị trên có ý nghĩa tương tự như câu lệnh điều kiện if - else. Quan sát ví dụ sau:

```
#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 DEBUG lẫn TEST
// không được định nghĩa
#endif
```

3.8.4 Chỉ thị #region và #endregion

Chỉ thị phục vụ cho các công cụ IDE như VS.NET cho phép mở/đóng các ghi chú.

```
#region Đóng mở một đoạn mã
// mã nguồn
#endregion
```

khi này VS.NET cho phép đóng hoặc mở vùng mã này. Ví dụ trên đang ở trạng thái mở. Khi ở trạng thái đóng nó vhư sau

```
Đóng mở một đoạn mã
```

Chương 4 Lớp và đối tượng

Đối tượng là một trị có thể được tạo ra, lưu giữ và sử dụng. Trong C# tất cả các biến đều là đối tượng. Các biến kiểu số, kiểu chuỗi … đều là đối tượng. Mỗi một đối tượng đều có các biến thành viên để lưu giữ dữ liệu và có các phương thức (hàm) để tác động lên biến thành viên. Mỗi đối tượng thuộc về một lớp đối tương nào đó. Các đối tượng có cùng lớp thì có cùng các biến thành viên và phương thức.

4.1 Định nghĩa lớp

Định nghĩa một lớp mới với cú pháp như sau:

```
[attribute][bổ từ truy xuất] class định danh [:lớp cơ sở]
{
  thân lớp
}
```

Ví dụ 4-1 Khai báo một lớp

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

Khi khai báo một lớp ta định nghĩa các đặc tính chung của tất cả các đối tượng của lớp và các hành vi của chúng.

Ví dụ 4-2 Khai báo, tạo và sử dụng một lớp

```
using System;
public class Time
{
  // phương thức public
  public void DisplayCurrentTime( )
  {
    Console.WriteLine( "stub for DisplayCurrentTime" );
  }
  // các biến private
  int Year; int Month; int Date;
  int Hour; int Minute; int Second;
}
public class Tester
{
  static void Main( )
  {
    Time t = new Time( );
    t.DisplayCurrentTime( );
  }
```

```
}
```

4.1.1 Bổ từ truy xuất

Bổ từ truy xuất xác định thành viên (nói tắt của biến thành viên và phương thức thành viên) nào của lớp được truy xuất từ lớp khác. Có các loại kiểu truy xuất sau:

Bảng 4-1 Các bổ từ truy xuất

Từ khóa	Giải thích
public	Truy xuất mọi nơi
protected	Truy xuất trong nội bộ lớp hoặc trong các lớp con
internal	Truy xuất nội trong chương trình (assembly)
protected internal	Truy xuất nội trong chương trình (assembly) và trong các lớp con
private (mặc định)	Chỉ được truy xuất trong nội bộ lớp

4.1.2 Các tham số của phương thức

Mỗi phương thức có thể không có tham số mà cũng có thể có nhiều tham số. Các tham số theo sau tên phương thức và đặt trong cặp ngoặc đơn. Ví dụ như phương thức *SomeMethod* sau:

Ví dụ 4-3 Các tham số và cách dùng chúng trong phương thức

```
using System;
public class MyClass
{
  public void SomeMethod(int firstParam, float secondParam)
  {
    Console.WriteLine("Here are the parameters received: {0}, {1}",
                firstParam, secondParam);
  }
}
public class Tester
{
  static void Main( )
  {
    int howManyPeople = 5;
    float pi = 3.14f;
    MyClass mc = new MyClass( );
    mc.SomeMethod(howManyPeople, pi);
  }
}
```

4.2 Tạo đối tượng

Tạo một đối tượng bằng cách khai báo kiểu và sau đó dùng từ khoá *new* để tạo như trong Java và C++.

4.2.1 Hàm dựng - Constructor

Hàm dựng là phương thức đầu tiên được triệu gọi và chỉ gọi một lần khi khởi tạo đối tượng, nó nhằm thiết lập các tham số đầu tiên cho đối tượng. Tên hàm dựng trùng tên lớp; còn các mặt khác như phương thức bình thường.

Nếu lớp không định nghĩa hàm dựng, trình biên dịch tự động tạo một hàm dựng mặc định. Khi đó các biến thành viên sẽ được khởi tạo theo các giá trị mặc định:

Bảng 4-2 Kiểu cơ sở và giá trị mặc định

Kiểu	Giá trị mặc định
số (int, long, …)	0
bool	false
char	'\0' (null)
enum	0
Tham chiếu	null

Ví dụ 4-4 Cách tạo hàm dựng

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime( )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }
   // constructor
   public Time(System.DateTime dt)
   {
      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
      Hour = dt.Hour;
      Minute = dt.Minute;
      Second = dt.Second;
   }
   // private member variables
   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( );
```

```
    }
}
Kết quả:
11/16/2000 16:21:40
```

4.2.2 Khởi tạo

Ta có thể khởi tạo giá tri các biến thành viên theo ý muốn bằng cách khởi tạo nó trong constructor của lớp hay có thể gán vào trực tiếp lúc khai báo. Với giá trị khởi tạo này thì khi một đối tượng khai báo kiểu của lớp này thì giá trị ban đầu là các giá trị khởi tạo chứ không phải là giá trị mặc định.

4.2.3 Hàm dựng sao chép

Hàm dựng sao chép (copy constructor) là sao chép toàn bộ nội dung các biến từ đối tượng đã tồn tại sang đối tượng mới khởi tạo.

Ví dụ 4-5 Một hàm dựng sao chép

```
public Time(Time existingTimeObject)
{
   Year = existingTimeObject.Year;
   Month = existingTimeObject.Month;
   Date = existingTimeObject.Date;
   Hour = existingTimeObject.Hour;
   Minute = existingTimeObject.Minute;
   Second = existingTimeObject.Second;
}
```

4.2.4 Từ khoá this

Từ khoá *this* được dùng để tham chiếu đến chính bản thân của đối tượng đó. Ví dụ:

```
public void SomeMethod (int hour)
{
   this.hour = hour;
}
```

4.3 Sử dụng các thành viên tĩnh

Các đặc tính và phương thức của một lớp có thể là *thành viên thể hiện (instance member)* hay *thành viên tĩnh*. Thành viên thể hiện thì kết hợp với thể hiện của một kiểu, trong khi các thành viên của static nó lại là một phần của lớp. Ta có thể truy cập các thành viên static thông qua tên của lớp mà không cần tạo một thể hiện lớp.

4.3.1 Cách gọi một thành viên tĩnh

Phương thức tĩnh (static) được nói là hoạt động trong lớp. Do đó, nó không thể được tham chiếu *this* chỉ tới. Phương thức static cũng không truy cập trực tiếp vào các phương thức không static được mà phải dùng qua thể hiện của đối tượng.

Ví dụ 4-6 Cách sử dụng phương thức tĩnh

```
using System;
```

```
public class MyClass
{
  public void SomeMethod(int firstParam, float secondParam)
  {
    Console.WriteLine(
      "Here are the parameters received: {0}, {1}",
      firstParam, secondParam);
  }
}
public class Tester
{
  static void Main( )
  {
    int howManyPeople = 5;
    float pi = 3.14f;
    MyClass mc = new MyClass( );
    mc.SomeMethod(howManyPeople, pi);
  }
}
```

Trong ví dụ trên phương thức *Main()* là tĩnh và phương thức *SomeMethod()* không là tĩnh.

4.3.2 Sử dụng hàm dựng tĩnh

Hàm dựng tĩnh (static constructor) sẽ được chạy trước khi bất kỳ đối tượng nào tạo ra.Ví dụ:

```
static Time( )
{
  Name = "Time";
}
```

Khi dùng hàm dựng tĩnh phải khá thận trọng vì nó có thể có kết quả khó lường.

4.3.3 Hàm dựng private

Khi muốn tạo một lớp mà không cho phép tạo bất kỷ một thể hiện nào của lớp thì ta dùng hàm dựng *private.*

4.3.4 Sử dụng các trường tĩnh

Cách dùng chung các biến thành viên tĩnh là giữ vết của một số các thể hiện mà hiện tại nó đang tồn tại trong lớp đó.

Ví dụ 4-7 Cách dùng trường tĩnh

```
using System;
public class Cat
{
  public Cat( )
  {
       instances++;
  }
  public static void HowManyCats( )
  {
```

```
      Console.WriteLine("{0} cats adopted",
        instances);
    }
    private static int instances = 0;
}
public class Tester
{
    static void Main( )
    {
      Cat.HowManyCats( );
      Cat frisky = new Cat( );
      Cat.HowManyCats( );
      Cat whiskers = new Cat( );
      Cat.HowManyCats( );
    }
}
Kết quả:
0 cats adopted
1 cats adopted
2 cats adopted
```

Ta có thể thấy được rằng phương thức static có thể truy cập vào biến static.

4.4 Hủy đối tượng

Giống với Java, C# cũng cung cấp bộ thu dọn rác tự động nó sẽ ngầm hủy các biến khi không dùng. Tuy nhiên trong một số trường hợp ta cũng cần hủy tường minh, khi đó chỉ việc cài đặt phương thức *Finalize(),* phương thức này sẽ được gọi bởi bộ thu dọn rác. Ta không cần phải gọi phương thức này.

4.4.1 Hủy tử của C#

Hủy tử của C# cũng giống như hủy tử trong C++. Khai báo một hủy tử theo cú pháp:

```
~<định danh>() {}
```

trong đó, định danh của hủy tử trùng với dịnh danh của lớp. Để hủy tường minh ta gọi phương thức *Finalize()* của lớp cơ sở trong nội dung của hủy tử này.

4.4.2 Finalize hay Dispose

Finalize không được pháp gọi tường minh; tuy nhiên trong trường hợp ta đang giữ môt tài nguyên hệ thống và hàm gọi có khả năng giải phóng tài nguyên này, ta sẽ cài đặt giao diện IDisposable (chí có một phương thức Dispose). Giao diện sẽ được đề cậpp ở chương sau.

4.4.3 Câu lệnh `using`

Bởi vì ta không thể chắc rằng *Dispose()* sẽ được gọi và vì việc giải phóng tài nguyên không thể xác định được, C# cung cấp cho ta lệnh using để đảm bảo rằng *Dispose()* sẽ được gọi trong thời gian sớm nhất. Ví dụ sau minh hoạ vấn đề này:

Ví dụ 4-8 Sử dụng using

```
using System.Drawing;
class Tester
{
  public static void Main( )
  {
    using (Font theFont = new Font("Arial", 10.0f))
    {
      // sử dụng theFont
    } // phương thức Dispose của theFont được gọi
    Font anotherFont = new Font("Courier",12.0f);
    using (anotherFont)
    {
      // sử dụng anotherFont
    } // phương thức Dispose của anotherFont được gọi
  }
}
```

4.5 Truyền tham số

C# cung cấp các tham số *ref* để h iệu chỉnh giá trị của những đối tượng bằng các tham chiếu.

4.5.1 Truyền bằng tham chiếu

Một hàm chỉ có thể trả về một giá trị. Trong trường hợp muốn nhận về nhiều kết quả, ta sử dụng chính các tham số truyền cho hàm như các tham số có đầu ra (chứa trị trả về). Ta gọi tham số truyền theo kiểu này là tham chiếu.

Trong C#, tất cả các biến có kiểu tham chiếu sẽ mặc định là tham chiếu khi các biến này được truyền cho hàm. Các biến kiểu giá trị để khai báo tham chiếu, sử dụng từ khóa ref.

Ví dụ 4-9 Trị trả về trong tham số

```
public class Time
{
  // một phương thức public
  public void DisplayCurrentTime( )
  {
    System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
      Month, Date, Year, Hour, Minute, Second);
  }
  public int GetHour( )
  {
    return Hour;
  }
  public void GetTime(ref int h, ref int m, ref int s)
  {
    h = Hour;
    m = Minute;
    s = Second;
  }
  // hàm dựng
  public Time(System.DateTime dt)
```

```
    {
      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
      Hour = dt.Hour;
      Minute = dt.Minute;
      Second = dt.Second;
    }
    // biến thành viên private
    private int Year;
    private int Month;
    private int Date;
    private int Hour;
    private int Minute;
    private int Second;
}
public class Tester
{
    static void Main( )
    {
      System.DateTime currentTime = System.DateTime.Now;
      Time t = new Time(currentTime);
      t.DisplayCurrentTime( );
      int theHour = 0;
      int theMinute = 0;
      int theSecond = 0;
      t.GetTime(ref theHour, ref theMinute, ref theSecond);
      System.Console.WriteLine("Current time: {0}:{1}:{2}",
        theHour, theMinute, theSecond);
    }
}
Kết quả:
11/17/2000 13:41:18
Current time: 13:41:18
```

4.5.2 Truyền tham số đầu ra (out parameter)

Như đã có đề ập ở các chương trước, dể sử dụng được, một biến phải được khai báo và khởi tạo giá trị ban đầu. Như trong Ví dụ 4-9 các biến theHour, theMinute, theSecond phải được khởi tạo giá trị 0 trước khi truyền cho hàm GetTime. Sau lời gọi hàm thì giá trị các biến sẽ thay đổi ngay, vì vậy C# cung cấp từ khóa out để không cần phải kho8\73i tạo tham số trước khi dùng. Ta sửa khai báo hàm GetTime trong ví dụ trên như sau:

```
public void GetTime(out int h, out int m, out int s)
```

Hàm Main() không cần khởi tạo trước tham số

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

Vì các tham số không được khởi gán trước nên trong thân hàm (như trường hợp này là GetTime) không thể sử dung các tham số (thực hiện phép lấy giá trị tham số) này trước khi khởi gán lại trong thân hàm. Ví dụ

```
public void GetTime(out int h, out int m, out int s)
{
```

```
    int nKhong_y_nghia = h; // lỗi, h chưa khởi gán
}
```

4.6 Nạp chồng phương thức và hàm dựng

Ta muốn có nhiều phương thức cùng tên mà mỗi phương thức lại có các tham số khác nhau, số lượng tham số cũng có thể khác nhau. Như vậy ý nghĩa của các phương thức được trong sáng hơn và các phương thức linh động hơn trong nhiều trường hợp. Nạp chồng cho phép ta làm được việc này.

Ví dụ 4-10 Nạp chồng hàm dựng

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime( )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }
   // constructors
   public Time(System.DateTime dt)
   {
      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
      Hour = dt.Hour;
      Minute = dt.Minute;
      Second = dt.Second;
   }
   public Time(int Year, int Month, int Date,
      int Hour, int Minute, int Second)
   {
      this.Year = Year;
      this.Month = Month;
      this.Date = Date;
      this.Hour = Hour;
      this.Minute = Minute;
      this.Second = Second;
   }
   // private member variables
   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( );
         Time t2 = new Time(2000,11,18,11,03,30);
      t2.DisplayCurrentTime( );
```

```
    }
}
```

4.7 Đóng gói dữ liệu với property

Trong lập trình C++, thông thường để đọc hoặc gán giá trị cho biến thành viên, lập trình viên thường viết hai hàm get và set tương ứng cho biến. C# cung cấp khai báo hàm chung gọi là property cho hàm get và set.

Ví dụ: trong lớp DocGia có biến thành viên m_sHoTen, cài đặt Property cho biến thành viên này như sau:

```
public string HoTen
{
  get {  return m_sHoTen; }
  set { m_sHoTen = value; }
}
```

Property có một vài khác biệt so với hàm thành viên. Thứ nhất khai báo Property không có tham số và cặp ngoặc. Trong thân property dùng hai từ khóa get/set tương ứng cho hai hành động lấy/thiết đặt giá trị thuộc tính. Trong thân set, có biến mặc định là value, biến này sẽ mang kiểu đã được khai báo property, như trong trường hợp trên là string. Biến value sẽ nhận giá trị được gán cho Property. Cách sử dụng một Property như sau:

```
1   // trong thân của một hàm
2   DocGia dgMoi = new DocGia();
3
4   // sử dung property set
5   dgMoi.HoTen = "Nguyễn Văn A";
6
7   // sử dụng property get
8   string ten = dgMoi.HoTen; //ten có giá trị "Nguyễn Văn A"
```

Ở dòng mã thứ 5, khối set trong property HoTen sẽ được gọi, biến value sẽ có giá trị của biến nằm sau phép gán (trong trường hợp này là "Nguyễn Van A").

Nếu trong thân hàm không cài đặt hàm set, property sẽ có tính chỉ đọc, phép gán sẽ bị cấm. Ngược lại nếu không cài đặt hàm get, property sẽ có tính chỉ ghi.

Ví dụ 4-11 Minh họa dùng một property

```
public class Time
{
  // public accessor methods
  public void DisplayCurrentTime( )
  {
    System.Console.WriteLine("Time\t: {0}/{1}/{2} {3}:{4}:{5}",
      month, date, year, hour, minute, second);
  }
  // constructors
  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 đặc tính
   public int Hour
   {
      get { return hour; }
      set { hour = value; }
   }
   // các biến thành viên kiểu 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("\nRetrieved the hour: {0}\n",
                               theHour);
      theHour++;
      t.Hour = theHour;
      System.Console.WriteLine("Updated the hour: {0}\n",
                                theHour);
   }
}
```

4.7.1 Phương thức get

Thân của phương thức *truy cập get* cũng giống như các phương thức khác nhưng phương thức này trả về một đối tượng kiểu là một đặc tính của lớp. Ví dụ muốn lấy *Hour* như sau:

```
get { return hour; }
```

4.7.2 Phương thức set

Phương thức *set* thiết lập giá trị một property của đối tượng và có trị trả về là *void*. Phương thức *set* có thể ghi vào cơ sở dữ liệu hay cập nhật biến thành viên khi cần. Ví dụ:

```
set { hour = value; }
```

4.7.3 Các trường chỉ đọc

C# cung cấp từ khoá *readonly* để khai báo các biến thành viên. Các biến khai báo kiểu này chỉ cho phép gán giá trị cho biến một lần vào lúc khởi tạo qua *constructor*.

Chương 5 Thừa kế và Đa hình

Thừa kế là cách tạo mới một lớp từ những lớp có sẵn. Tức là nó cho phép tái sử dụng lại mã nguồn đã viết trong lớp có sẵn. Thừa kế nói đơn giản là việc tạo một đối tượng khác B thừa hưởng tất cả các đặc tính của lớp A. Cách này gọi là đơn thừa kế. Nếu lớp B muốn có đặc tính của nhiều lớp A1, A2 … thì gọi là đa thừa kế. Đa thừa kế là khái niệm rất khó cài đặt cho các trình biên dịch. C# cũng như nhiều ngôn ngữ khác tìm cách tránh né khái niệm này.

Đa hình là việc lớp B thừa kế các đặc tính từ lớp A nhưng có thêm một số cài đặt riêng.

5.1 Đặc biệt hoá và tổng quát hoá

Sự đặc biệt và tổng quát hoá có mối quan hệ tương hổ và phân cấp. Khi ta nói *ListBox* và *Button* là những cửa sổ (Window), có nghĩa rằng ta tìm thấy được đầy đủ các đặc tính và hành vi của *Window* đều tồn tại trong hai loại trên. Ta nói rằng Window là tổng quát hoá của ListBox và Button; ngược lại ListBox và Button là hai đặc biệt hoá của Window

5.2 Sự kế thừa

Trong C#, mối quan hệ chi tiết hoá là một kiểu kế thừa. Sự kế thừa không cho mang ý nghĩa chi tiết hoá mà còn mang ý nghĩa chung của tự nhiên về mối quan hệ này.

Khi ta nói rằng *ListBox* kế thửa từ *Window* có nghĩa là nó chi tiết hoá *Window*. *Window* được xem như là lớp *cơ sở* (base class) và *ListBox* được xem là lớp *kế thừa* (derived class). Lớp *ListBox* này nhận tất cả các đặc tính và hành vi của *Window* và chi tiết hoá nó bằng một số thuộc tính và phương thức của nó cần.

5.2.1 Thực hiện kế thừa

Trong C#, khi ta tạo một lớp kế thừa bằng cách công một thêm dấu "**:**" và sau tên của lớp kế thừa và theo sau đó là lớp cơ sở như sau:

```
public class ListBox : Window
```

có nghĩa là ta khai báo một lớp mới *ListBox* kế thừa từ lớp *Window*.

Lớp kế thừa sẽ thừa hưởng được tất các phương thức và biến thành viên của lớp cơ sở, thậm chí còn thừa hưởng cả các thành viên mà cơ sở đã thừa hưởng.

Ví dụ 5-1 Minh hoạ cách dùng lớp kế thừa

```
public class Window
{
```

```
    // constructor takes two integers to
    // fix location on the console
    public Window(int top, int left)
    {
      this.top = top;
      this.left = left;
    }
    // simulates drawing the window
    public void DrawWindow( )
    {
      System.Console.WriteLine("Drawing Window at {0}, {1}",
        top, left);
    }
    // these members are private and thus invisible
    // to derived class methods; we'll examine this
    // later in the chapter
    private int top;
    private int left;
}
// ListBox kế thừa từ Window
public class ListBox : Window
{
    // thêm tham số vào constructor
    public ListBox(
      int top,
      int left,
      string theContents):
      base(top, left) // gọi constructor cơ sở
    {
      mListBoxContents = theContents;
    }
    // tạo một phương thức mới bởi vì trong
    // phương thức kế thừa có sự thay đổi hành vi
    public new void DrawWindow( )
    {
      base.DrawWindow( ); // gọi phương thức cơ sở
      System.Console.WriteLine ("Writing string to the listbox:
                                  {0}", mListBoxContents);
    }
    private string mListBoxContents; // biến thành viên mới
}
public class Tester
{
    public static void Main( )
    {
      // tạo một thể hiện cơ sở
      Window w = new Window(5,10);
      w.DrawWindow( );
      // tạo một thể hiện kế thừa
      ListBox lb = new ListBox(20,30,"Hello world");
      lb.DrawWindow( );
    }
}
Kết quả:
Drawing Window at 5, 10
Drawing Window at 20, 30
Writing string to the listbox: Hello world
```

5.2.2 Gọi hàm dựng lớp cơ sở

Trong Ví dụ 5-1 lớp *ListBox* thừa kế từ *Window* và có hàm dựng ba tham số. Trong hàm dựng của *ListBox* có lời gọi đến hàm dựng của Window thông qua từ khoá *base* như sau:

```
public ListBox( int top, int left, string theContents):
  base(top, left) // gọi constructor cơ sở
```

Bởi vì các hàm dựng không được thừa kế nên lớp kế thừa phải thực hiện hàm dựng của riêng nó và chỉ có thể dùng hàm dựng cơ sở thông qua lời gọi tường minh. Nếu lớp cơ sở có hàm dựng mặc định thì hàm dựng lớp kế thừa không cần thiết phải gọi hàm dựng cơ sở một cách tường minh (mặc định được gọi ngầm).

5.2.3 Gọi các phương thức của lớp cơ sở

Để gọi các phương thức của lớp cơ sở C# cho phép ta dùng từ khoá *base* để gọi đến các phương thức của lớp cơ sở hiện hành.

```
base.DrawWindow( ); // gọi phương thức cơ sở
```

5.2.4 Cách điều khiển truy cập

Cách truy cập vào các thành viên của lớp được giới hạn thông qua cách dùng các từ khoá khai báo kiểu truy cập và hiệu chỉnh (như trong chương 4.1). Xem Bảng 4-1 Các bổ từ truy xuất

5.3 Đa hình

Đa hình là việc lớp B thừa kế các đặc tính từ lớp A nhưng có thêm một số cài đặt riêng. Đa hình cũng là cách có thể dùng nhiều dạng của một kiểu mà không quan tâm đến chi tiết.

5.3.1 Tạo kiểu đa hình

ListBox và *Button* đều là một *Window*, ta muốn có một form để giữ tập hợp tất cả các thể hiện của *Window* để khi một thể hiện nào được mở thì nó có thể bắt *Window* của nó vẽ lên. Ngắn gọn, form này muốn quản lý mọi cư xử của tất cả các đối tượng đa hình của *Window*.

5.3.2 Tạo phương thức đa hình

Tạo phương thức đa hình, ta cần đặt từ khoá *virtual* trong phương thức của lớp cơ sở. Ví dụ như:

```
public virtual void DrawWindow( )
```

Trong lớp kế thừa để nạp chồng lại mã nguồn của lớp cơ sở ta dùng từ khoá *override* khi khai báo phương thức và nội dung bên trong viết bình thường. Ví dụ về nạp chồng phương thức *DrawWindow*:

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

```
        base.DrawWindow( ); // gọi phương thức của lớp co sở
        Console.WriteLine ("Writing string to the listbox: {0}",
          listBoxContents);
    }
```

Dùng hình thức đa hình phương thức này thì tuỳ kiểu khai báo của đối tượng nào thì nó dùng phương thức của lớp đó.

5.3.3 Tạo phiên bản với từ khoá new và override

Khi cần viết lại một phương thức trong lớp kế thừa mà đã có trong lớp cơ sở nhưng ta không muốn nạp chồng lại phương thức *virtual* trong lớp cơ sở ta dùng từ khoá *new* đánh dấu trước khi từ khoá *virtual* trong lớp kế thừa.

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

5.4 Lớp trừu tượng

Phương thức trừu tượng là phương thức chỉ có tên thôi và nó phải được cài đặt lại ở tất các các lớp kế thừa. Lớp trừu tượng chỉ thiết lập một cơ sở cho các lớp kế thừa mà nó không thể có bất kỳ một thể hiện nào tồn tại.

Ví dụ 5-2 Minh hoạ phương thức và lớp trừu tượng

```
using System;
abstract public class Window
{
  // constructor takes two integers to
  // fix location on the console
  public Window(int top, int left)
  {
    this.top = top;
    this.left = left;
  }
  // simulates drawing the window
  // notice: no implementation
  abstract public void DrawWindow( );
  // these members are private and thus invisible
  // to derived class methods. We'll examine this
  // later in the chapter
  protected int top;
  protected int left;
}
// ListBox derives from Window
public class ListBox : Window
{
  // constructor adds a parameter
  public ListBox(int top, int left, string contents):
    base(top, left) // call base constructor
  {
    listBoxContents = contents;
  }
  // an overridden version implementing the
  // abstract method
```

```
    public override void DrawWindow( )
    {
         Console.WriteLine("Writing string to the listbox: {0}",
           listBoxContents);
    }
    private string listBoxContents; // new member variable
}
public class Button : Window
{
    public Button(int top, int left): base(top, left)
    {
    }
    // implement the abstract method
    public override void DrawWindow( )
    {
      Console.WriteLine("Drawing a button at {0}, {1}\n", top, left);
    }
}
public class Tester
{
    static void Main( )
    {
      Window[] winArray = new Window[3];
      winArray[0] = new ListBox(1,2,"First List Box");
      winArray[1] = new ListBox(3,4,"Second List Box");
      winArray[2] = new Button(5,6);
      for (int i = 0;i < 3; i++)
      {
        winArray[i].DrawWindow( );
      }
    }
}
```

5.4.1 Giới hạn của lớp trừu tượng

Ví dụ trên, phương thức trừu tượng *DrawWindow()* của lớp trừu tượng *Window* được lớp *ListBox* kế thừa. Như vậy, các lớp sau này kế thừa từ lớp *ListBox* đều phải thực hiện lại phương thức *DrawWindow()*, đây là điểm giới hạn của lớp trừu tượng. Hơn nữa, như thế sau này không bao giờ ta tạo được lớp *Window* đúng nghĩa. Do vậy, nên chuyển lớp trừu tượng thành giao diện trừu tượng.

5.4.2 Lớp niêm phong

Lớp niêm phong với ý nghĩa trái ngược hẳn với lớp trừu tượng. Lớp niêm phong không cho bất kỳ lớp nào khác kế thừa nó. Ta dùng từ khoá *sealed* để thay cho từ khoá *abstract* để được lớp này.

5.5 Lớp gốc của tất cả các lớp: Object

Trong C#, các lớp kế thừa tạo thành cây phân cấp và lớp cao nhất (hay lớp cơ bản nhất) chính là lớp *Object*. Các phương thức của lớp *Object* như sau:

Bảng 5-1 Các phương thức của lớp đối tượng Object

Phương thức	Ý nghĩa sử dụng
Equals	So sánh giá trị của hai đối tượng
GetHashCode	
GetType	Cung cấp kiểu truy cập của đối tượng
ToString	Cung cấp một biểu diễn chuỗi của đối tượng
Finalize()	Xoá sạch bộ nhớ tài nguyên
MemberwiswClone	Tạo sao chép đối tượng; nhưng không thực thi kiểu

Ví dụ 5-3 Minh hoạ việc kế thừa lớp 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("The value of i is: {0}", i.ToString( ));
    SomeClass s = new SomeClass(7);
    Console.WriteLine("The value of s is {0}", s.ToString( ));
  }
}
Kết quả:
The value of i is: 5
The value of s is 7
```

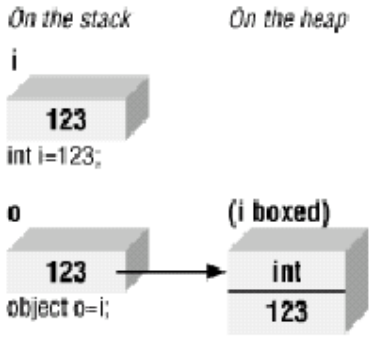
5.6 Kiểu Boxing và Unboxing

Boxing và unboxing là tiến trình cho phép kiểu giá trị (value type) được đối xử như kiểu tham chiếu (reference type). Biến kiểu giá trị được "gói (boxed)" vào đối tượng Object, sau đó ngược lại được "tháo (unboxed)" về kiểu giá trị như cũ.

5.6.1 Boxing là ngầm định

Boxing là tiến trình chuyển đổi một kiểu giá trị thành kiểu *Object*. Boxing là một giá trị được định vị trong một thể hiện của *Object*.

Hình 5-1 Kiểu tham chiếu Boxing



Boxing là ngầm định khi ta cung cấp một giá trị ở đó một tham chiếu đến giá trị này và giá trị được chuyển đổi ngầm định.

Ví dụ 5-4 Minh họa boxing

```
using System;

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

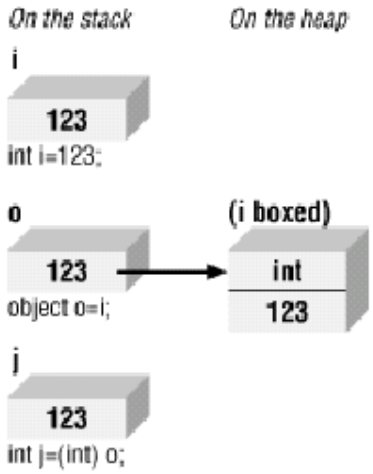
Console.WriteLine() mong chờ một đối tượng, không phải là số nguyên. Để phù hợp với phương thức, kiểu interger được tự động chuyển bởi CLR và *ToString()* được gọi để lấy kết quả đối tượng. Đặc trưng này cho phép ta tạo các phương thức lấy một đối tượng như là một tham chiếu hay giá trị tham số, phương thức sẽ làm việc với nó.

5.6.2 Unboxing phải tường minh

Trả kết quả của một đối tượng về một kiểu giá trị, ta phải thực hiện mở tường minh nó. Ta nên thiết lập theo hai bước sau:

1. Chắc chắn rằng đối tượng là thể hiện của một trị đã được box.
2. Sao chép giá trị từ thể hiện này thành giá trị của biến.

Hình 5-2 Boxing và sau đó unboxing



Ví dụ 5-5 Minh họa boxing và unboxing

```
using System;
public class UnboxingTest
{
  public static void Main( )
  {
    int i = 123;
    //Boxing
    object o = i;
    // unboxing (must be explict)
    int j = (int) o;
    Console.WriteLine("j: {0}", j);
  }
}
```

5.7 Lớp lồng

Lớp được khai báo trong thân của một lớp được gọi là lớp nội (inner class) hay lớp lồng (nested class), lớp kia gọi là lớp ngoại (outer class). Lớp nội có thuận lợi là truy cập được trực tiếp tất cả các thành viên của lớp ngoài. Một phương thức của lớp nội cũng có thể truy cập đến các thành viên kiểu *private* của các lớp ngoài. Hơn nữa, lớp nội nó ẩn trong lớp ngoài so với các lớp khác, nó có thể là thành viên kiểu *private* của lớp ngoài. Khi lớp nội (vd: Inner) được khai báo public, nó sẽ được truy xuất thông qua tên của lớp ngoài (vd: Outer) như: Outer.Inner.

Ví dụ 5-6 Cách dùng lớp nội

```
using System;
using System.Text;
public class Fraction
{
  public Fraction(int numerator, int denominator)
  {
    this.numerator=numerator;
    this.denominator=denominator;
```

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

Chương 6 Nạp chồng toán tử

Mục tiêu thiết kế của C# là kiểu người dùng định nghĩa (lớp) phải được đối xử như các kiểu định sẵn. Ví dụ, chúng ta muốn định nghĩa lớp phân số (Fraction) thì các chức năng như cộng, trừ, nhân, … phân số là điều tất yếu phải có. Để làm được việc đó ta định nghĩa các phương thức: cộng, nhân, … khi đó, ta phải viết là:

```
Phân_số tổng = số_thứ_nhất.cộng(số_thứ_hai);
```

Cách này hơi gượng ép và không thể hiện hết ý nghĩa. Điều ta muốn là viết thành:

```
Phân_số tổng = số_thứ_nhất + số_thứ_hai;
```

để làm được điều này ta dùng từ khoá *operator* để thể hiện.

6.1 Cách dùng từ khoá operator

Trong C#, các toán tử là các phương thức tĩnh, kết quả trả về của nó là giá trị biểu diễn kết quả của một phép toán và các tham số là các toán hạng. Khi ta tạo một toán tử cho một lớp ta nói là ta nạp chồng toán tử, nạp chồng toán tử cũng giống như bất kỳ việc nạp chồng các phương thức nào khác. Ví dụ nạp chồng toán tử cộng (+) ta viết như sau:

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

Nó chuyển tham số *lhs* về phía trái toán tử và *rhs* về phía phải của toán tử.

Cú pháp C# cho phép nạp chồng toán tử thông qua việc dùng từ khoá *operator*.

6.2 Cách hổ trợ các ngôn ngữ .Net khác

C# cung cấp khả năng nạp chồng toán tử cho lớp của ta, nói đúng ra là trong Common Language Specification (CLS). Những ngôn ngữ khác như VB.Net có thể không hổ trợ nạp chồng toán tử, do đó, điều quan trọng là ta cũng cung cấp các phương thức hổ trợ kèm theo các toán tử để có thể thực hiện được ở các môi trường khác. Do đó, khi ta nạp chồng toán tử cộng (+) thì ta cũng nên cung cấp thêm phương thức *add()* với cùng ý nghĩa.

6.3 Sự hữu ích của các toán tử

Các toán tử được nạp chồng có thể giúp cho đoạn mã nguồn của ta dễ nhìn hơn, dễ quản lý và trong sáng hơn. Tuy nhiên nếu ta quá lạm dụng đưa vào các toán tử quá mới hay quá riêng sẽ làm cho chương trình khó sử dụng các toán tử này mà đôi khi còn có các nhầm lẫn vô vị nữa.

6.4 Các toán tử logic hai ngôi

Các toán tử khá phổ biến là toán tử (==) so sánh bằng giữ hai đối tượng, (!=) so sánh không bằng, (<) so sánh nhỏ hơn, (>) so sánh lớn hơn, (<=, >=) tương ứng nhỏ hơn hay bằng và lớn hơn hay bằng là các toán tử phải có cặp toán hạng hay gọi là các toán tử hai ngôi.

6.5 Toán tử so sánh bằng

Nếu ta nạp chồng toán tử so sánh bằng (==), ta cũng nên cung cấp phương thức ảo *Equals()* bởi *object* và hướng chức năng này đến toán tử bằng. Điều này cho phép lớp của ta đa hình và cung cấp khả năng hữu ích cho các ngôn ngữ .Net khác. Phương thức *Equals()* được khai báo như sau:

```
public override bool Equals(object o)
```

Bằng cách nạp chồng phương thức này, ta cho phép lớp *Fraction* đa hình với tất cả các đối tượng khác. Nội dung của *Equals()* ta cần phải đảm bảo rằng có sự so sánh với đối tượng *Fraction* khác. Ta viết như sau:

```
public override bool Equals(object o)
{
  if (! (o is Fraction) )
  {
    return false;
  }
  return this == (Fraction) o;
}
```

Toán tử *is* được dùng để kiểm tra kiểu đang chạy có phù hợp với toán hạng yêu cầu không. Do đó, `o is Fraction` là đúng nếu *o* có kiểu là *Fraction*.

6.6 Toán tử chuyển đổi kiểu (ép kiểu)

Trong C# cũng như C++ hay Java, khi ta chuyển từ kiểu thấp hơn (kích thước nhỏ) lên kiểu cao hơn (kích thước lớn) thì việc chuyển đổi này luôn thành công nhưng khi chuyển từ kiểu cao xuống kiểu thấp có thể ta sẽ mất thông tin. Ví dụ ta chuyển từ *int* thành *long* luôn luôn thành công nhưng khi chuyển ngược lại từ *long* thành *int* thì có thể tràn số không như ý của ta. Do đó khi chuyển từ kiểu cao xuống thấp ta phải chuyển tường minh.

Cũng vậy muốn chuyển từ *int* thành kiểu *Fraction* luôn thành công, ta dùng từ khoá *implicit* để biểu thị toán tử kiểu này. Nhưng khi chuyển từ kiểu *Fraction* có thể sẽ mất thông tin do vậy ta dùng từ khoá *explicit* để biểu thị toán tử chuyển đổi tường minh.

Ví dụ 6-1 Minh hoạ chuyển đổi ngầm định và tường minh

```
using System;
public class Fraction
{
  public Fraction(int numerator, int denominator)
```

```
{
   Console.WriteLine("In Fraction Constructor(int, int)");
   this.numerator=numerator;
   this.denominator=denominator;
}
public Fraction(int wholeNumber)
{
   Console.WriteLine("In Fraction Constructor(int)");
   numerator = wholeNumber;
   denominator = 1;
}
   public static implicit operator Fraction(int theInt)
{
   System.Console.WriteLine("In implicit conversion to Fraction");
   return new Fraction(theInt);
}
public static explicit operator int(Fraction theFraction)
{
   System.Console.WriteLine("In explicit conversion to int");
   return theFraction.numerator / theFraction.denominator;
}
public static bool operator==(Fraction lhs, Fraction rhs)
{
   Console.WriteLine("In operator ==");
   if (lhs.denominator == rhs.denominator &&
        lhs.numerator == rhs.numerator)
   {
      return true;
   }
   // code here to handle unlike fractions
   return false;
}
public static bool operator !=(Fraction lhs, Fraction rhs)
{
   Console.WriteLine("In operator !=");
   return !(lhs==rhs);
}
public override bool Equals(object o)
{
   Console.WriteLine("In method Equals");
   if (! (o is Fraction) )
   {
      return false;
   }
   return this == (Fraction) o;
}
public static Fraction operator+(Fraction lhs, Fraction rhs)
{
   Console.WriteLine("In operator+");
   if (lhs.denominator == rhs.denominator)
   {
      return new Fraction(lhs.numerator+rhs.numerator,
        lhs.denominator);
   }
   // simplistic solution for unlike fractions
   // 1/2 + 3/4 == (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( )
    {
      //implicit conversion to Fraction
      Fraction f1 = new Fraction(3);
      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( ));
      }
      int k = (int)f4; //explicit conversion to int
      Console.WriteLine("int: F5 = {0}", k.ToString());
    }
}
```

Chương 7 Cấu trúc

Một *cấu trúc* (struct) là một kiểu do người dùng định nghĩa, nó tương tự như lớp như nhẹ hơn lớp.

7.1 Định nghĩa cấu trúc

Cú pháp

```
[thuộc tính] [kiểu truy cập] struct <định danh> [: <danh sách các giao diện >]
{
   // Các thành viên của cấu trúc
}
```

Ví dụ 7-1 Minh họa cách khai báo và dùng một cấu trúc

```
using System;
public struct Location
{
  public Location(int xCoordinate, int yCoordinate)
  {
    xVal = xCoordinate;
    yVal = yCoordinate;
  }
  public int x
  {
    get{ return xVal; }
    set{ xVal = value; }
  }
  public int y
  {
    get{ return yVal; }
    set{ yVal = value; }
  }
  public override string ToString( )
  {
    return (String.Format("{0}, {1}", xVal,yVal));
  }
  private int xVal;
  private int yVal;
}
public class Tester
{
  public void myFunc(Location loc)
  {
    loc.x = 50;
    loc.y = 100;
    Console.WriteLine("Loc1 location: {0}", loc);
  }
  static void Main( )
  {
    Location loc1 = new Location(200,300);
```

```
        Console.WriteLine("Loc1 location: {0}", loc1);
        Tester t = new Tester( );
        t.myFunc(loc1);
        Console.WriteLine("Loc1 location: {0}", loc1);
    }
}
Kết quả:
Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 200, 300
```

Không giống như lớp, cấu trúc không hỗ trợ kế thừa. Tất cả các cấu trúc thừa kế ngầm định *object* nhưng nó không thể thừa kế từ bất kỳ lớp hay cấu trúc nào khác. Các cấu trúc cũng ngầm định là đã *niêm phong*. Tuy nhiên, nó có điểm giống với lớp là cho phép cài đặt đa giao diện.

Cấu trúc không có hủy tử cũng như không thể đặt các tham số tuỳ ý cho hàm dựng. Nếu ta không cài đặt bất kỳ hàm dựng nào thì cấu trúc được cung cấp hàm dựng mặc định, đặt giá trị 0 cho tất cả các biến thành viên.

Do cấu trúc được thiết kế cho nhẹ nhàng nên các biến thành viên đều là kiểu private và được gói gọn lại hết. Tuỳ từng tình huống và mục đích sử dụng mà ta cần cân nhắc chọn lựa dùng lớp hay cấu trúc.

7.2 Cách tạo cấu trúc

Muốn tạo một thể hiện của cấu trúc ta dùng từ khoá *new*. Ví dụ như:

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

7.2.1 Cấu trúc như các kiểu giá trị

Khi ta khai báo và tạo mới một cấu trúc như trên là ta đã gọi đến constructor của cấu trúc. Trong Ví dụ 7-1 trình biên dịch tự động đóng gói cấu trúc và nó được đóng gói kiểu object thông qua *WriteLine()*. *ToString()*được gọi theo kiểu của *object*, bởi vì các cấu trúc thừa kế ngầm từ *object*, nên nó có khả năng đa hình, nạp chồng phương thức như bất kỳ đối tượng nào khác.

Cấu trúc là object giá trị và khi nó qua một hàm, nó được thông qua như giá trị.

7.2.2 Gọi hàm dựng mặc định

Theo trên đã trình bày khi ta không tạo bất kỳ này thì khi tạo một thể hiện của cấu trúc thông qua từ khoá *new* nó sẽ gọi đến constructor mặc định của cấu trúc. Nội dung của constructor sẽ đặt giá trị các biến về 0.

7.2.3 Tạo cấu trúc không dùng new

Bởi vì cấu trúc không phải là lớp, do đó, thể hiện của lớp được tạo trên stack. Cấu trúc cũng cho phép tạo mà không cần dùng từ khoá *new*, nhưng trong trường hợp này constructor không được gọi (cả mặc định lẫn người dùng định nghĩa).

Chương 8 Giao diện

Giao diện định nghĩa các hợp đồng (constract). Các lớp hay cấu trúc cài đặt giao diện này phải tôn trọng hợp đồng này. Điều này có nghĩa là khẳng định với client (người dùng lớp hay cấu trúc) rằng "Tôi bảo đảm rằng tôi sẽ hỗ trợ đầy đầy đủ các phương thức, property, event, delegate, indexer đã được ghi trong giao diện"

Một giao diện có thể thừa kế một hay nhiều giao diện khác, và một lớp hay cấu trúc có thể cài đặt một hay nhiều giao diện.

Quan sát về phía lập trình thì giao diện là tập các hàm được khai báo sẵn mà không cài đặt. Các lớp hay cấu trúc cài đặt có nhiệm vụ phải cài tất cả các hàm này.

8.1 Cài đặt một giao diện

Cú pháp của việc định nghĩa một giao diện:

```
[attributes] [access-modifier] interface interface-name [:base-list]
{
   interface-body
}
```

Ý nghĩa của từng thành phần như sau

```
attributes: sẽ đề cập ở phần sau.
modifiers: bổ từ phạm vi truy xuất của giao diện
identifier: tên giao diện muốn tạo
base-list: danh sách các giao diện mà giao diện này thừa kế,
            (nói rõ trong phần thừa kế)
interface-body: thân giao diện luôn nằm giữa cặp dấu {}
```

Trong thư viện .NET Framework các giao diện thường bắt đầu bởi chữ I (i hoa), điều này không bắt buộc. Giả sử rằng chúng ta tạo một giao diện cho các lớp muốn lưu trữ xuống/đọc ra từ cơ sở dữ liệu hay các hệ lưu trữ khác. Đặt tên giao diện này là IStorable, chứa hai phương thức Read( ) và Write( ).

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

Giao diện như đúng tên của nó: không dữ liệu, không cài đặt. Một giao diện chỉ trưng ra các khả năng, và khải năng này sẽ được hiện thực hoá trong các lớp cài đặt nó. Ví dụ như ta tạo lớp Document, do muốn các đối tượng Document sẽ được lưu trữ vào cơ sở dữ liệu, nên ta cho Document kế thừa (cài đặt) giao diện IStorable.

```
// lớp Document thừa kế IStorable,
//  phải cài đặt tất cả các phương thức của IStorable
public class Document : IStorable
```

```
{
  public void Read( ) { // phải cài đặt...}
  public void Write(object obj) { // phải cài đặt...}
  // ...
}
```

8.1.1 Cài đặt nhiều giao diện

Lớp có thể cài đặt một hoặc nhiều giao diện. Chẳng hạn như ở lớp Document ngoài lưu trữ ra nó còn có thể được nén lại. Ta cho lớp Document cài đặt thêm một giao diện thứ hai là ICompressible

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

Tương tự, Document phải cài đặt tất cả phương thức của ICompressible:

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

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

8.1.2 Mở rộng giao diện

Chúng ta có thể mở rộng (thừa kế) một giao diện đã tồn tại bằng cách thêm vào đó những phương thức hoặc thành viên mới. Chẳng hạn như ta có thể mở rộng ICompressable thành ILoggedCompressable với phương thức theo dõi những byte đã được lưu:

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

Lớp cài đặt phải cân nhắc chọn lựa giữa 2 lớp ICompressable hay ILoggedCompressable, điều này phụ thuộc vào nhu cầu của lớp đó. Nếu một lớp có sử dụng giao diện ILoggedCompressable thì nó phải thực hiện toàn bộ các phương thức của ILoggedCompressable (bao gồm ICompressable và phương thức mở rộng).

8.1.3 Kết hợp các giao diện khác nhau

Tương tự, chúng ta có thể tạo một giao diện mới bằng việc kết hợp nhiều giao diện và ta có thể tùy chọn việc có thêm những phương thức hoặc những thuộc tính mới. Ví dụ như ta tạo ra giao diện IStorableCompressable từ giao diện IStorable và ILoggedCompressable và thêm vào một phương thức mới dùng để lưu trữ kích thước tập tin trước khi nén.

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

8.2 Truy xuất phương thức của giao diện

Chúng ta có thể truy xuất thành viên của giao diện IStorable như chúng là thành viên của lớp Document:

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

hoặc ta có thể tạo một thể diện của giao diện bằng việc phân phối tài liệu về kiểu của giao diện và sau đó sử dụng giao diện để truy cập những phương thức:

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

In this case, in Main( ) you *know* that Document is in fact an IStorable, so you can take advantage of that knowledge. As stated earlier, you cannot instantiate an interface directly. That is, you cannot say:

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

Mặc dù vậy, chúng ta có thể tạo một thể hiện của lớp thi công như sau:

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

Sau đấy ta có thể tạo một thể hiện của giao diện bằng việc phân bổ những đối tượng thi công đến những kiểu giao diện, trong trường hợp này là IStorable:

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

Chúng ta kết hợp những bước đã mô tả trên bằng đoạn mã dưới đây:

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

8.2.1 Ép kiểu thành giao diện

Trong nhiều trường hợp, chúng ta không biết đối tượng ấy hỗ trợ những giao diện loại gì. Giả sử như chúng ta có một tập các giao diện của Documents, một số trong chúng có thể lưu trữ còn một số khác thì không thể, chúng ta sẽ thêm vào một giao diện thứ hai ICompressable cho những đối tượng thuộc loại này để chúng có thể nén lại cho công việc chuyển đổi có liên quan đến email nhanh hơn.

```
interface ICompressible
{
void Compress( );
void Decompress( );
}
```

Với kiểu của Document, chúng ta có thể không biết rằng chúng được hỗ trợ bởi giao diện IStorable hoặc giao diện ICompressable hoặc cả hai. Chúng ta có thể giải quyết điều này bằng cách phân bổ những giao diện lại:

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

Nếu Document chỉ hỗ trợ bởi giao diện IStorable thì giá trị trả về là:

```
public class Document : IStorable
```

Việc phân bổ ICompressable phải đến khi biên dịch mới biết được bởi vì ICompressable là một giao diện hợp lệ. Mặc dù vậy, nếu sự phân bổ tồi thì có thể sẽ xảy ra lỗi, và lúc ấy thì một exception sẽ được quăng ra để cảnh báo:

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

Chi tiết về exception sẽ được đề cập trong những chương sau:

8.2.2 Toán tử "is "

Khi chúng ta muốn một đối tượng có khả năng hỗ trợ giao diện, theo nguyên tắc là chúng ta phải gọi phương thức tương ứng lên. Trong C# có 2 phương thức hỗ trợ công việc này.

Cú pháp như sau:

```
expression is type
hay
if (doc is IStorable)
```

Chắc lớp giao diện IStorable chắc bạn vẫn còn nhớ, ở đây câu lệnh if sẽ kiểm tra xem đối tượng doc có hỗ trợ giao diện IStorable không mà thôi.

Thật không may mắn cho chúng ta, tuy rát dễ hiểu với cách viết như thế nhưng chúng lại không hiệu quả cho lắm. Để hiểu vấn đề là tại sao lại như thế thì chúng ta cần phải nhúng chúng vào trong mã MSIL và sau đó phát sinh. Và sau đây là một số kết quả (thể hiện bằng số Hexa)

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

Có một số vấn đề là chúng ta phải chú ý là trong phần kiểm tra ICompressable trong dòng 23. Từ khóa isinst là mã MSIL của tác tử is. Như ta thấy trong phần kiểm tra đối tượng doc ở phía bên phải và ở dòng 2b thì việc kiểm tra thành công khi castclass được gọi.

8.2.3 Toán tử "as"

Toán tử as kết hợp tác tử is và sự phân bổ các thao tác bằng việc kiểm tra sự phân bổ có hợp lệ hay không (giá trị sẽ trả về là true) và sau đấy sẽ hoàn tất công việc. Nếu sự phân bổ không hợp lệ (tác tử is sẽ trả về giá trị false), tác tử as sẽ trả về giá trị null. Cú pháp của việc khai báo:

```
expression as type
```

Đoạn mã sau đây sử dụng tác tử as:

```
static void Main( )
```

```
{
  Document doc = new Document("Test Document");
  IStorable isDoc = doc as IStorable;
  if (isDoc != null)
    isDoc.Read( );
  else
    Console.WriteLine("IStorable not supported");

  ICompressible icDoc = doc as ICompressible;
  if (icDoc != null)
    icDoc.Compress( );
  else
    Console.WriteLine("Compressible not supported");
}
```

Hãy xem qua đoạn mã MSIL, chúng ta thấy có một số điểm thuận tiện:

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

8.2.4 Toán tử is hay toán tử as

Các giao diện xem ra có vẻ là những lớp trừu tượng. Thật ra thì chúng ta có thể thay đổi phần khai báo của giao diện IStorable thành lớp trừu tượng:

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

Lớp Document kế thừa từ lớp Storable, giả sử như chúng ta vừa mua một lớp List từ một hãng thứ ba với mong muốn là có sự kết hợp của List với Storable. Trong C++ ta có thể tạo một lớp StorableList bằng cách kế thừa từ List và Storable nhưng trong C# thì ta không thể vì C# không hỗ trợ đa thừa kế.

Mặc dù vậy, C# cho phép chúng ta chỉ rõ ra số giao diện và kết xuất từ lớp cơ sở. Bằng vệc tạo một giao diện Storable, ta có thể kế thừa từ lớp List và giao diện IStorable như trong ví dụ sau:

```
public class StorableList : List, IStorable
{
  // List methods here ...
  public void Read( ) {...}
  public void Write(object obj) {...}
  // ...
}
```

8.3 Nạp chồng phần cài đặt giao diện

Một lớp thi công thật sự tự do thì phải đánh dấu một vài hoặc toàn bộ các phương thức có thể thực hiện được giao diện như là phương thức ảo. Lớp dẫn xuất từ chúng có thể nạp chồng. Chẳng hạn như lớp Document có thể thực hiện giao diện

IStorable và xem các phương thức Read( ) và Write( ) như là phương thức ảo. Người phát triển có thể kết xuất từ những kiểu của Document, như là kiểu Note hay EmailMessage và anh ta có là quyết định Note với tính năng là sẽ được đọc và viết vào cơ sở dữ liệu hơn là việc thể hiện bằng một tập tin.

8.4 Thực hiện giao diện một cách tường minh

Bởi vì một lớp có thể cài đặt nhiều giao diện nên có thể xảy ra trường hợp đụng độ về tên khi khi hai giao diện có cùng một tên hàm. Để giải quyết xung đột này ta khai báo cài đặt một cách tường minh hơn. Ví dụ như nếu ta có hai giao diện IStorable và ITalk đều cùng có phương thức Read(), lớp Document sẽ cài đặt hai giao diện này. Khi đó ta ta phải thêm tên giao diện vào trước tên phương thức

```
using System;
interface IStorable
{
  void Read( );
  void Write( );
}
interface ITalk
{
  void Talk( );
  void Read( );
}

public class Document : IStorable, ITalk
{
  // document constructor
  public Document(string s)
  {
    Console.WriteLine("Creating document with: {0}", s);
  }
  // tạo read của IStorable
  public virtual void Read( )
  {
    Console.WriteLine("Implementing IStorable.Read");
  }
  public void Write( )
  {
    Console.WriteLine("Implementing IStorable.Write");
  }

  // cài đặt phương htức Read của ITalk
  void ITalk.Read( )
  {
    Console.WriteLine("Implementing ITalk.Read");
  }
  public void Talk( )
  {
    Console.WriteLine("Implementing ITalk.Talk");
  }
}
public class Tester
{
  static void Main( )
```

```
    {
      // create a document object
      Document theDoc = new Document("Test Document");

      // Ép kiểu để có thể gọi IStorable.Read()
      IStorable isDoc = theDoc as IStorable;
      if (isDoc != null)
      {
        isDoc.Read( );
      }

      // Ép kiểu để có thể gọi ITalk.Read()
      ITalk itDoc = theDoc as ITalk;
      if (itDoc != null)
      {
        itDoc.Read( );
      }

      theDoc.Read( );
      theDoc.Talk( );
    }
}
Kết quả:
Creating document with: Test Document
Implementing IStorable.Read
Implementing ITalk.Read
Implementing IStorable.Read
Implementing ITalk.Talk
```

8.4.1 Chọn lựa phơi bày các phương thức của giao diện

Người thiết kế lớp có thêm một thận lợi là khi một giao diện được thi công thì trong suốt quá trình ây sự thi công tường minh ấy không được thể hiện bên phía client ngoại trừ việc phân bổ. Giả sử như đối tượng Document thi công giao diện IStorable nhưng chúng ta không muốn các phương thức Read( ) và Write( ) được xem như là public trong lớp Document. Chúng ta có thể sử dụng phần thực hiện tường minh để chắc rằng chúng không sẵn có trong suốt quá trình phân bổ. Điều này cho phép chúng giữ gìn ngữ nghĩa của lớp Document trong khi ta thực hiện IStorable. Nếu Client muốn một object có thể thi công trên giao diện IStorable, thì chúng phải có sự phân bổ một cách tường minh nhưng khi sử dụng tài liệu của chúng ta như là Document trong ngữ nghĩa là sẽ không có các phương thức Read( ) và Write ( ).

8.4.2 Thành viên ẩn

Với một khả năng mới là một thành viên của giao diện có thể được ẩn đi. Ví dụ như chúng ta tạo một giao diện IBase với property P:

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

Và khi những giao diện được kế thừa từ nó, chẳng hạn IDerived thì property P đượoc ẩn đi với một phương thức mới P( )

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

Việc làm ẩn thành viên như là làm trên đối với IBase có thể xem như là một ý tưởng tốt, bây giờ thì chúng ta có thể ẩn property P trong giao diện cơ sở. Và trong những giao diện được kế thừa từ chúng sẽ phải cần tối thiểu là 1 giao diện thành viên tường minh. Do đó ta có thể sử dụng phần thi công tường minh này cho property cơ sở hoặc cho những phương thức kế thừa hoặc sử dụng cả hai. Do đó mà ta có thể 3 phiên bản cài đặt khác nhau nhưng vận hợp lệ:

```
class myClass : IDerived
{
  // explicit implementation for the base property
  int IBase.P { get {...} }
  // implicit implementation of the derived method
  public int P( ) {...}
}

class myClass : IDerived
{
  // implicit implementation for the base property
  public int P { get {...} }
  // explicit implementation of the derived method
  int IDerived.P( ) {...}
}

class myClass : IDerived
{
  // explicit implementation for the base property
  int IBase.P { get {...} }
  // explicit implementation of the derived method
  int IDerived.P( ) {...}
}
```

Chương 9 Array, Indexer, and Collection

.NET Framework cung cấp cho ta rất nhiều kiểu lớp tập hợp: Array, ArrayList, NameValueCollection, StringCollection, Queue, Stack, và BitArray. Array là lớp đơn giản nhất. Trong C# nó được ánh xạ thành cú pháp dựng sẵn tương tự như C/C++.

Net Framework cũng cung nấp những giao diện chuẩn như IEnumerable, ICollection để tương tác với các lớp tập hợp (túi chứa).

9.1 Mảng (Array)

Mảng là một tập hợp các phần tử có cùng kiểu, được xác định vị trí trong tập hợp bằng chỉ mục. C# cung cấp những dạng cú pháp dạng đơn giản nhất cho việc khai báo một mảng, rất dễ học và sử dụng.

9.1.1 Khai báo mảng

Chúng ta có thể khai báo một mảng kiểu C# như sau:

```
kiểu[] tên_mảng;
```

Ví dụ như:

```
int[] myIntArray;
```

Dấu ngoặc vuông [ ] biểu thị cho tên biến ở sau là một mảng Ví dụ dưới đây khai báo một biến kiểu mảng nguyên myIntArray với số phần tử ban đầu là 5:

```
myIntArray = new int[5];
```

9.1.2 Giá trị mặc định

Giả sử có đoạn mã sau:

```
/*1*/ int[] myArray;
/*2*/ maArray = new int[5];

/*3*/ Button[] myButtonArray;
/*4*/ myButtonArray = new Button[5];
```

dòng /*1*/ khai báo biến myArray là một mảng kiểu int. Khi này biến myArray có giá trị là null do chưa được khởi tạo. Dòng /*2*/ khởi tạo biến myArray, các phần tử trong mảng được khởi tạo bằng giá trị mặc định là 0. Dòng /*3*/ tương tự /*1*/ nhưng Button thuộc kiểu tham chiếu (reference type). Dòng /*4*/ khởi tạo biến myButtonArray, các phần tử trong mảng không được khởi tạo (giá trị "khởi tạo" là null). Sử dụng bất kỳ phần tử nào của mảng cũng gây lỗi chưa khởi tạo biến.

9.1.3 Truy cập đến những phần tử trong mảng

Để truy cập đến những phần tử trong mảng, ta sử dụng toán tử lấy chỉ mục []. Cũng giống như C/C++, chỉ mục mảng được tính bắt đầu từ phần tử 0. Property Length của lớp Array cho biết được kích thước một mảng. Như vậy chỉ mục của mảng đi từ 0 đến Length - 1. Trong mảng myArray ví dụ trên để lấy phần tử thứ 2 (có chỉ số là 1) trong mảng, ta viết như sau:

```
int phan_tu_thu_hai = myArray[1];
```

9.2 Câu lệnh foreach

foreach là một lệnh vòng lặp, dùng để duyệt tất cả các phần tử của một mảng, tập hợp (nói đúng hơn là những lớp có cài đặt giao diện IEnumerable). Cú pháp của foreach nhẹ nhàng hơn vòng lặp for (ta có thể dùng for thay cho foreach)

```
foreach (kiểu tên_biến in biến_mảng)
{
  khối lệnh
}
```

Ví dụ 9-1 Sử dụng foreach

```
using System;
namespace Programming_CSharp
{
  // một lớp đơn giản để chứa trong mảng
  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];
      // populate the array
      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());
```

```
        }
    }
}
```

9.2.1 Khởi tạo các phần tử mảng

Ta có thể khởi tạo các phần tử mảng vào thời điểm khai báo mảng, bằng cách ta cung cấp một danh sách những giá trị của mảng được giới hạn trong hai dấu ngoặc nhọn { }. C# có thể cung cấp những cú pháp ngắn gọn như sau:

```
int[] myIntArray = new int[5] { 2, 4, 6, 8, 10 }
int[] myIntArray = { 2, 4, 6, 8, 10 }
```

Hai cách trên cho cùng kết quả là một mảng 5 phần tử có giá trị là 2, 4, 6, 8, 10.

9.2.2 Từ khóa params

Đôi lúc có những phương thức ta không biết trước số lương tham số được truyền vào như: phương thức Main() không thể biết trước số lượng tham số người dùng sẽ truyền vào. Ta có thể sử tham số là mảng. Tuy nhiên khi gọi hàm ta phải tạo một biến mảng để làm tham số. C# cung cấp cú pháp để ta không cần truyền trực tiếp các phần tử của mảng bằng cách thêm từ khóa params

Ví dụ 9-2 Sử dụng từ khóa params

```
using System;
namespace Programming_CSharp
{
  public class Tester
  {
    static void Main( )
    {
      Tester t = new Tester( );

      /**
       * cách truyền tham số bằng các phần tử
       * không cần phải khởi tạo mảng
       * (cú pháp rất tự do)
       */
      t.DisplayVals(5,6,7,8);

      /**
       * Cách truyền tham số bằng mảng
       * Mảng phải được tạo sẵn
       */
      int [] explicitArray = new int[5] {1,2,3,4,5};
      t.DisplayVals(explicitArray);
    }
    public void DisplayVals(params int[] intVals)
    {
      foreach (int i in intVals)
      {
        Console.WriteLine("DisplayVals {0}",i);
      }
    }
```

```
    }
}
Kết quả:
DisplayVals 5
DisplayVals 6
DisplayVals 7
DisplayVals 8
DisplayVals 1
DisplayVals 2
DisplayVals 3
DisplayVals 4
DisplayVals 5
```

9.2.3 Mảng nhiều chiều

Ma trận là một ví dụ về mảng hai chiều. C# cho phép khai báo mảng n chiều, tuy nhiên thông dụng nhất vẫn là mảng một chiều (mảng) và mảng hai chiều. Ví dụ trong phần này là mảng hai chiều, tuy nhiên đối với n chiều cú pháp vẫn tương tự.

9.2.3.1 Mảng chữ nhật

Trong mảng chữ nhật (Rectangular array) 2 chiều, chiều thứ nhất là số dòng và chiều thứ hai là số cột. Số phần tử trong các dòng là như nhau và bằng số cột (tương tự số phần tử trong các cột là như nhau và bằng số dòng) để khai báo ta sử dụng cú pháp sau:

```
type [,] array-name
```

ví dụ như:

```
int [,] myRectangularArray;
```

9.2.3.2 Mảng Jagged

Mảng jagged là loại mảng trong mảng. Loại mảng này thật sự thì chúng chỉ là mảng một chiều nhưng những phần tử của chúng có khả năng quản lí được một mảng khác nữa, mà kích thước các mảng này thay đổi tùy theo nhu cầu của lập trình viên. Ta có thể khai báo như sau:

```
type [ ] [ ]...
```

Ví dụ như khai báo một mảng hai chiều với tên là myJaggedArray:

```
int [ ] [ ] myJaggedArray;
```

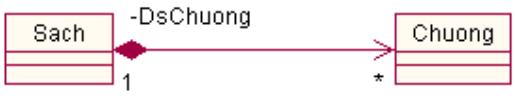
Chúng ta có thể truy cập phần tử thứ 5 của mảng thứ ba bằng cú pháp myJaggedArray[2][4]

9.2.4 Lớp System.Array

Lớp Array có rất nhiều hàm hữu ích, nó làm cho mảng trong C# "thông minh" hơn nhiều ngôn ngữ khác. Chúng được hỗ trợ như là các phương thức được dựng sẵn như trường hợp string. Hai phương thức quan trong nhất của lớp System.Array là Sort() và Reverse().

9.3 Indexers

Indexer tương tự như Property, tuy có khác nhau một chút về ý nghĩa. Xét một ví dụ mô phỏng một quyển sách có nhiều chương



Xây dựng 2 lớp Sách và Chương. Lớp Chương cài đặt bình thường. Với lớp Sách ta sẽ cài đặt một biến thành viên có kiểu túi chứa. Để đơn giản biến này có kiểu là một mảng

```
public class Chuong
{
  // Các biến thành viên
  string m_sTen;
  string m_sNoiDung;
}
public class Sach
{
  // biến thành viên
  Chuong[] m_dsChuong;

  // Property
  public Chuong[] DsChuong
  {
    get{ return m_dsChuong; }
  }
}
```

Cách làm này có vài bất lợi như sau: thứ nhất để lấy nội dung từng chương chúng ta dùng Property để lấy danh sach chương sau đó duyệt qua mảng để lấy chương mong muốn. Thứ hai là mỗi chương được định danh bởi tên chương nên ta mong muốn có cách lấy một chương thông qua tên của nó. Ta có thể cài đặt một hàm để duyệt qua mảng các chương, nhưng Indexer sẽ giúp làm việc này.

Ví dụ 9-3 Sử dụng indexer

```
using System;
using System.Collections;
namespace ConsoleApplication
{
   // Cài đặt lớp Chuong
   public class Chuong
   {
      private string m_sTen;
      private string m_sNoiDung;
      public Chuong()
      {
         m_sTen = "";
         m_sNoiDung = "";
      }
      public Chuong(string sTen, string sNoiDung)
      {
         m_sTen = sTen;
```

```
      m_sNoiDung = sNoiDung;
    }
    public string Ten
    {
      get { return m_sTen; }
      set { m_sTen = value; }
    }
    public string NoiDung
    {
      get { return m_sNoiDung; }
      set { m_sNoiDung = value; }
    }
  } // hết class Chuong
  // Cài đặt lớp Sach
  public class Sach
  {
    private string m_sTen;
    private ArrayList m_dsChuong;
    public Sach()
    {
      m_sTen = "";
      m_dsChuong = new ArrayList();
    }
    public Sach(string sTen)
    {
      m_sTen = sTen;
      m_dsChuong = new ArrayList();
    }
    public string Ten
    {
      get { return m_sTen; }
      set
      {
        if ( value == null )
          throw new ArgumentNullException();

        m_sTen = value;
      }
    }
    // indexer thứ nhất có một tham số kiểu int
    public Chuong this[int index]
    {
      get
      {
        if ( index < 0 || index > m_dsChuong.Count - 1 )
          return null;

        return (Chuong)m_dsChuong[index];
      }
      set
      {
        if ( index < 0 || index > m_dsChuong.Count - 1 )
          throw new ArgumentOutOfRangeException();

        m_dsChuong[index] = value;
      }
    }
```

```
        // indexer thứ hai có một tham số kiểu string
        public Chuong this[string tenChuong]
        {
          get
          {
            foreach (Chuong chuong in m_dsChuong)
            {
              if ( chuong.Ten == tenChuong )
              {
                return chuong;
              }
            }
            return null;
          }
        }
        public int add (Chuong chuong)
        {
          if ( chuong == null )
            throw new ArgumentNullException();

          return m_dsChuong.Add(chuong);
        }
      }// hết class Sach

      class Class
      {
        static void Main(string[] args)
        {
          Sach s = new Sach("tlv");
          s.add(new Chuong("CS", "Tac gia CS"));
           s.add(new Chuong("VB", "Tac gia VB"));

          Console.WriteLine(s.Ten);
          // dùng indexer thứ nhất
          Console.WriteLine(s[0].Ten + ": "+ s[0].NoiDung);
          // dùng indexer thứ hai
          Console.WriteLine("VB: " + s["VB"].NoiDung);

          Console.Read();
        }
      }
    }
```

Trước hết quan sát lớp Sach để xem khai báo một indexer

```
// indexer thứ nhất có một tham số kiểu int
public Chuong this[int index]
```

public: phạm vi truy xuất của indexer
Chuong: kiểu trả về
int index: kiểu và tên tham số nhận vào
this[...]: bắt buộc để khai báo indexer

Thân hàm Indexer cũng chia thành 2 hàm get và set y hệt như Property. Indexer cung cấp thêm một hoặc nhiều tham số và cho ta cách sử dụng như sử dụng một mảng:

```
          // dùng indexer thứ nhất
          Console.WriteLine(s[0].Ten + ": "+ s[0].NoiDung);
```

```
// dùng indexer thứ hai
Console.WriteLine("VB: " + s["VB"].NoiDung);
```

9.4 Các giao diện túi chứa

.NET Framework cung cấp một số các giao diện chuẩn để tương tác với các lớp túi chứa hay để cài đặt các lớp túi chứa mới tương thích (có cùng giao diện) với các lớp chuẩn của .NET Framework. Các giao diện được liệt kê ở Bảng 9-1 Các giao diện túi chứa

Bảng 9-1 Các giao diện túi chứa

Giao diện	Ý nghĩa
IEnumerable	Khi một lớp cài đặt giao diện này, đối tượng thuộc lớp có được dùng trong câu lệnh foreach
ICollection	Được cài đặt bởi tất cả các lớp túi chứa có thành viên CopyTo(), Count, IsReadOnly(), IsSyncronize(), SyncRoot()
IComparer	So sánh hai đối tượng trong túi chứa
IList	Dùng bởi các lớp túi chứa truy xuất phần tử thông qua chỉ mục (số)
IDictionary	Dùng bởi các lớp túi chứa truy xuất phần tử thông qua quan hệ khóa/giá trị như Hashtabe, StoredList.
IDictionaryEnumerator	Cho phép duyệt đối với các túi chứa cài đặt IDictionary

9.5 Array Lists

Một vấn đề cổ điển trong khi sử dụng lớp Array là kích thước: kích thước một mảng cố định. Nếu không thể biết trước cần có bao nhiêu phần tử, ta có thể khai báo quá nhiều (lãng phí) hay quá ích (chương trình có lỗi). ArrayList cài đãt cấu trúc dữ liệu danh sach liệt kê cho phép cấp phát động các phần tử. Lớp này cài đặt giao diện IList, ICollection, IEnumerable và có rất nhiều hàm dùng để thao tác lên danh sách.

IComparable

ArrayList có phương thức Sort( ) giúp chúng ta sắp xếp các phần tử. Điều bắt buộc là phần tử phải thuộc lớp có cài đặt giao diện IComparable (có duy nhất một phương thức CompareTo()).

9.6 Hàng đợi

Hàng đợi (queue) là một túi chứa hoạt động theo cơ chế FIFO (First in first out - vào trước ra trước). Cũng giống như ta đi xếp hàng mua vé xem phim, nếu ta vào trước mua vé thì ta sẽ được mua vé trước.

Hàng đợi là một tập hợp tốt cho việc ta sử dụng để quản lí nguồn tài nguyên có giới hạn. Ví dụ như ta gửi đi những thông điệp đến tài nguyên, mà tài nguyên thì chỉ có thể giải quyết cho một thông điệp. Do đó chúng ta phải tạo một hàng đợi những

thông điệp để cho tài nguyên có thể dựa vào đó mà xử lí "từ từ". Lớp Queue cài đặt túi chứa này.

9.7 Stacks

Stack là túi chứa hoạt động theo cơ chế LIFO (Last in first out - vào sau ra trước). Hai phương thức cơ bản trong việc thêm hoặc xóa Stack là:Push( ) và Pop( ). Ngoài ra Stack còn có phương thức Peek( ), tương tự như Pop() nhưng không xóa bỏ phần tử này.

Các lớp ArrayList, Queue và Stack đều có những phương thức giống nhau như CopyTo( ) và ToArray( ) dùng để sao chép những phần tử vào trong một mảng. Trong trường hợp Stack thì phương thức CopyTo( ) sẽ sao chép những phần tử vào một mảng một chiều đã tồn tại, và nội dung chúng sẽ ghi đè lên vị trí bắt đầu mà chúng ta đã chỉ định. Phương thức ToArray( ) trả về một mảng mới với nội dung là nhữngphần tử trong stack.

9.8 Dictionary

Dictionary là tên gọi chung cho các túi chứa lưu trữ các phần tử theo quan hệ khóa/giá trị. Điều này có nghĩa là tương ứng với một "khóa", ta tìm được một và chỉ duy nhất một "giá trị" tương ứng.Nói cách khác là một "giá trị" có một "khóa" duy nhất không trùng với bất kỳ "khóa" của giá trị khác.

Một lớp muốn là một Dictionary thì cài đặt giao diện IDictionary. Lớp Dictionary muốn được sử dụng trong câu lệnh foreach thì cài đặt giao diện IDictionaryEnumerator.

Dictionary thường được dùng nhất là bảng băm (Hashtable).

Bảng băm

Hashtable là cấu trúc dữ liệu có mục tiêu tối ưu hóa việc tìm kiếm. .et Framework cung cấp lớp Hashtable cài đặt cấu trúc dữ liệu này.

Một đối tượng được dùng như "khóa" phải cài đặt hay thừa kế phương thức Object.GetHashCode() và Object.Equals() (các lớp thư viện .NET Framework hiển nhiên thỏa điều kiện này). Một điều kiện nữa là đối tượng này phải immutable (dữ liệu các trường thành viên không thay đổi) trong lúc đang là khóa.

Chương 10 Chuỗi

Chuỗi (string) trong C# là một kiểu dựng sẵn như các kiểu `int`, `long`…, có đầy đủ tính chất mềm dẻo, mạnh mẽ và dễ dùng. Một đối tượng chuỗi trong C# là một hay nhiều ký tự Unicode không thể thay đổi thứ tự. Nói cách khác là các phương thức áp dụng lên chuỗi không làm thay đổi bản thân chuỗi, chúng chỉ tạo một bản sao có sửa đổi, chuỗi gốc vẫn giữ nguyên.

Để khai báo một đối tượng chuỗi, sử dụng từ khóa `string`; đối tượng này thật sự trùng với đối tượng `System.String` trong thư viện lớp .NET Framework. Việc sử dụng hai đối tượng này là như nhau trong mọi trường hợp.

Khai báo lớp `System.String` như sau:

```
public sealed class String:IComparable,ICloneable,Iconvertible
```

Khai báo này có nghĩa như sau:

```
seal - không thể thừa kế từ lớp này
ICompareable - các đối tượng chuỗi có thể được sắp thứ thự
IClonable - có thể tạo một đối tượng B mới y hệt đối tượng A.
IConvertible - có thể chuyển thành các kiểu dựng sẵn khác như
ToInt32(), ToDouble() …
```

10.1 Tạo chuỗi mới

Cách đơn giản nhất để tạo một biến kiểu chuỗi là khai báo và gán một chuỗi cho nó

```
string sChuoi = "Khai báo và gán một chuỗi";
```

Một số ký tự đặc biệt có qui tắc riêng như `"\n"`, `"\\"` hay `"\t"`… đại diện cho ký tự xuống dòng, dấu xuyệt (\), dấu tab…Ví dụ khai báo

```
string sDuongDan = "C:\\WinNT\\Temp";
```

biến sDuongDan sẽ có giá trị C:\WinNT\Temp. C# cung cấp một cách khai báo theo đúng nguyên gốc chuỗi bằng cách thêm ký tự @. Khai báo `sDuongDan` sẽ như sau

```
string sDuongDan = @"C:\WinNT\Temp";
```

10.2 Phương thức ToString()

Đây là phương thức của đối tượng `object` (và của tất cả các đối tượng khác) thường dùng để chuyển một đối tượng bất kỳ sang kiểu chuỗi.

```
int myInteger = 5;
string integerString = myInteger.ToString();
```

Chuỗi `intergerString` có giá trị là "5". Bằng cách này ta cũng có thể tạo một chuỗi mới. Chuỗi cũng có thể được tạo thông qua hàm dựng của lớp `System.String`. Lớp này có hàm dựng nhận vào một mảng các ký tự. Như vậy ta cũng tạo được chuỗi từ mảng ký tự.

10.3 Thao tác chuỗi

Lớp chuỗi cung cấp nhiều phương thức cho việc so sánh, tìm kiếm… được liệt kê trong bảng sau:

Bảng 10-1 Các thành viên lớp string

Thành viên	Giải thích
Empty	Biến thành viên tĩnh đại diện cho một chuỗi rỗng
Compare()	Phương thức tĩnh so sánh hai chuỗi
CompareOrdinal()	Phương thức tĩnh, so sánh 2 chuỗi không quan tâm đến ngôn ngữ
Concat()	Phương thức tĩnh, tạo một chuỗi mới từ nhiều chuỗi
Copy()	Phương thức tĩnh, tạo một bản sao
Equals()	Phương thức tĩnh, so sánh hai chuỗi có giống nhau
Format()	Phương thức tĩnh, định dạng chuỗi bằng các định dạng đặc tả
Intern()	Phương thức tĩnh, nhận về một tham chiếu đến chuỗi
IsInterned()	Phương thức tĩnh, nhận về một tham chiếu đến chuỗi đã tồn tại
Join()	Phương thức tĩnh, ghép nối nhiều chuỗi, mềm dẻo hơn Concat()
Chars	Indexer của chuỗi
Length	Chiều dài chuỗi (số ký tự)
Clone()	Trả về một chuỗi
CompareTo()	So sánh với chuỗi khác
CopyTo()	Sao chép một lượng ký tự trong chuỗi sang mảng ký tự
EndsWith()	Xác định chuỗi có kết thúc bằng chuỗi tham số không
Equals()	Xác định hai chuỗi có cùng giá trị
Insert()	Chèn một chuỗi khác vào chuỗi
LastIndexOf()	vị trí xuất hiện cuối cùng của một chuỗi con trong chuỗi
PadLeft()	Canh phải các ký tự trong chuỗi, chèn thêm các khoảng trắng bên trái khi cần
PadRight()	Canh trái các ký tự trong chuỗi, chèn thêm các khoảng trắng bên phải khi cần
Remove()	Xóa một số ký tự
Split()	Cắt một chuỗi thành nhiều chuỗi con
StartsWith()	Xác định chuỗi có bắt đầu bằng một chuỗi con tham số
SubString()	Lấy một chuỗi con
ToCharArray()	Sao chép các ký tự của chuỗi thành mảng các ký tự
ToLower()	Tạo bản sao chuỗi chữ thường
ToUpper()	Tạo bản sao chuỗi chữ hoa
Trim()	Cắt bỏ các khoảng trắng hai đầu chuỗi

Thành viên	Giải thích
TrimEnd()	Cắt bỏ khoảng trắng cuối chuỗi
TrimStart()	Cắt bỏ khoảng trắng đầu chuỗi

Để biết chi tiết các sử dụng của các hàm trên, có thể tham thảo tài liệu của Microsoft, đặc biệt là MSDN. Dưới đây chỉ giới thiệu vài phương thức thao dụng để thao tác chuỗi.

Ghép chuỗi

Để ghép 2 chuỗi ta dùng toán tử +

```
string a = "Xin";
string b = "chào";
string c = a + " " + b; // c = "Xin chào"
```

Chú ý: việc ghép nối bằng toán tử + tuy cho mã nguồn đẹp, tự nhiên nhưng sẽ không cho hiệu quả tốt khi thực hiện nhiều lần vì C# sẽ cấp phát vùng nhớ lại sau mỗi phép ghép chuỗi.

Lấy ký tự

Để lấy một ký tự tại một ví trí trên chuỗi ta dùng toán tử []

```
string s = "Xin chào mọi người";
char c = s[5]; // c = 'h'
```

Chú ý: vị trí rên chuỗi bắt đầu từ vị trí số 0

Chiều dài chuỗi

Để biết số ký tự của chuỗi, dùng thuộc tính Length

```
string s =  "Xin chào";
int l = s.Length; // l = 8
```

Chú ý: không cần đóng ngoặc sau property

Lấy chuỗi con

Để lấy chuỗi con của một chuỗi, sử dụng phương thức Substring().

```
string s;
/* 1 */ s = "Lay chuoi con".Substring(4);// s = "chuoi con"
/* 2 */ s = "Lay chuoi con".Substring(4, 5); // s = "chuoi"
```

Trong `/*1*/ s` lấy chuỗi con tính từ vị trí thứ 4 trở về sau, còn trong `/*2*/ s` lấy chuỗi con từ vị trí thứ 4 và lấy chuỗi con có chiều dài là 5.

Thay thế chuỗi con

Để thay thế chuỗi con trong chuỗi bằng một chuỗi con khác, sử dụng phương thức `Replace()`

```
string s;
/* 1 */ s = "thay the chuoi.".Replace('t', 'T');
        // s = "Thay The chuoi"
/* 2 */ s = "thay the chuoi.".Replace("th", "TH");
```

```
// s = "THay THe chuoi"
```

Trong `/*1*/ s` là chuỗi đã thay thế ký tự `'t'` thành `'T'`, còn trong `/*2*/` là chuỗi đã thay thế chuỗi `"th"` thành `"TH"`.

Định dạng chuỗi

Chuỗi được sử dụng nhiều trong trường hợp kết xuất kết quả ra cho người dùng. Trong nhiều trường hợp ta không thể có được chính xác chuỗi cần thiết mà phải phụ thuộc vào một số biến. Vì vậy hàm định dạng chuỗi giúp ta định dạng lại chuỗi trước khi kết xuất.

```
double d = tinh_toan_phuc_tap_1();
double e = tinh_toan_phuc_tap_2();
// giả sử d = 2.5, e = 3.5
string s;
s = string.Format("Kết quả là: {0:C} va {1:c} đôla", d, e);
// s = "Kết quả là: $2.5 và $3.5 đôla"
```

Hàm định dạng chuỗi khá phức tạp vì có nhiều tùy chọn. Cú pháp củ hàm định dạng tổng quát như sau

```
string.Format(provider, format, arguments)
provider: nguồn cung cấp định dạng
format: chuỗi cần định dạng chứa thông tin định dạng
arguments: các thông số cho định dạng
```

C# tạo sẵn các nguồn định đạng cho kiểu số, kiểu dùng nhiều nhất, vì vậy ta chỉ quan tâm đến cú pháp rút gọn sau và các thông tin định dạng cho kiểu số.

```
string.Format (format, arguments);
```

Hình 10-1 Vài định dạng thông dụng

Ký tự	Mô tả	Ví dụ	Kết quả
C hoặc c	Tiền tệ (Currency)	string.Format("{0:C}", 2.5); string.Format("{0:C}", -2.5);	$2.50 ($2.50)
D hoặc d	Decimal	string.Format("{0:D5}", 25);	00025
E hoặc e	Khoa hoc (Scientific)	string.Format("{0:E}", 250000);	2.500000E+005
F hoặc f	Cố định phần thập phân (Fixed-point)	string.Format("{0:F2}", 25); string.Format("{0:F0}", 25);	25.00 25
G hoặc g	General	string.Format("{0:G}", 2.5);	2.5
N hoặc n	Số (Number)	string.Format("{0:N}", 2500000);	2,500,000.00
X hoặc x	Hệ số 16 (Hexadecimal)	string.Format("{0:X}", 250); string.Format("{0:X}", 0xffff);	FA FFFF

10.4 Thao tác chuỗi động

Sau mỗi thao tác lên chuỗi sẽ tạo ra một bản sao chuỗi mới. Vì vậy sử dụng đối tượng `string` có thể làm giảm hiệu năng hệ thống. Khi đó ta nên sử dụng lớp `StringBuilder` (một loại chuỗi khác). Các thao tác lên chuỗi làm thay đổi trên chính chuỗi. Vài phương thức quan trọng của lớp được liệt kê dưới đây.

Phương thức	Giải thích
Capacity	Lấy/thiết đặt số ký tự tối đa chuỗi có thể lưu giữ
Chars	Indexer
Length	Kích thước chuỗi
MaxCapacity	Lấy số ký tự tối đa lớp có thể lưu giữ
Append()	Thêm một đối tượng vào cuối chuỗi
AppendFormat()	Định dạng chuỗi tham số, sau đó thêm chuỗi này vào cuối
EnsureCapacity()	Xác định chuỗi có thể lưu giữ tối thiểu một lượng ký tự không
Insert()	Chèn một đối tượng vào chuỗi tại vị trí
Remove()	Xóa một số ký tự trong chuỗi
Replace()	Thay một ký tự/chuỗi con bằng ký tự/chuỗi con mới

Ví dụ 10-1 Sử dụng StringBuilder

```
using System;
using System.Text;
namespace Programming_CSharp
{
  public class StringTester
  {
    static void Main( )
    {
      // một chuỗi bất kỳ để thao tác
      string s1 = "One,Two,Three Liberty Associates, Inc.";
      // hằng ký tự
      const char Space = ' ';
      const char Comma = ',';
      // mảng các dấu cách
      char[] delimiters = new char[]{ Space, Comma };
      // dùng StringBuilder để tạo một chuỗi
      StringBuilder output = new StringBuilder( );
      int ctr = 1;
      // tách chuỗi, sau đó ghép lại theo dang mong muốn
      // tách chuỗi theo các dấu phân cách trong delimiter
      foreach (string subString in s1.Split(delimiters))
      {
        // chèn một chuỗi sau khi định dạng chuỗi xong
        output.AppendFormat("{0}: {1}\n",ctr++,subString);
      }
      Console.WriteLine(output);
    }
  }
}
Kết quả:
1: One
2: Two
3: Three
4: Liberty
5: Associates
6:
7: Inc.
```

Chương 11 Quản lý lỗi

C# quản lý lỗi và các trạng thái bất thường bằng *biệt lệ* (exception). Một biệt lệ là một đối tượng chứa các thông tin về sự cố bất thường của chương trình.

Điều quan trọng trước hết là phải phân biệt rõ sự khác nhau giữa bug, error và biệt lệ. Bug là lỗi về mặt lập trình do chính lập trình viên không kiểm soát được mã nguồn. Biệt lệ không thể sửa các bug. Mặc dù bug sẽ phát sinh (ném) một biệt lệ, chúng ta không nên dựa vào các biệt lệ để sửa các bug, mà nên viết lại mã nguồn cho đúng.

Error là lỗi gây ra bởi người dùng. Chẳng hạn như người dùng nhập một con số thay vì phải nhập các ký tự chữ cái. Một error cũng ném ra một biệt lệ, nhưng ta có thể ngăn chặn bằng cách bắt lấy lỗi này, yêu cầu người dùng chỉnh sửa cho đến khi hợp lệ. Bất cứ khi nào có thể, error nên được tiên đoán trước và ngăn chặn.

Ngay cả khi các bug đã được sửa, các error đã được tiên đoán hết thì vẫn còn nhiều tình huống không thể lường trước như: hệ thống đã hết bộ nhớ hay chương trình đang truy cập một tập tin không tồn tại…. Chúng ta không thể ngăn chặn được biệt lệ nhưng có lại có thể quản lý được chúng để chúng không làm gẫy đỗ ứng dụng.

Khi chương trình gặp phải tình huống trên, chẳng hạn hết bộ nhớ, nó sẽ ném (phát sinh) một biệt lệ. Khi một biệt lệ được ném ra, hàm đang thực thi sẽ bị tạm dừng và vùng nhớ stack sẽ được duyệt ngược cho đến khi gặp trình giải quyết biệt lệ.

Điều này có nghĩa là nếu hàm hiện hành không có trình giải quyết biệt lệ thì hàm sẽ bị ngắt và hàm gọi sẽ có cơ hội để giải quyết lỗi. Nếu không có hàm gọi nào giải quyết biệt lệ thì biệt lệ sẽ được ném cho CLR giải quyết. Điều này đồng nghĩa với việc chương trình sẽ bị dừng một cách bất thường.

Trình quản lý lỗi (exception handler) là một đoạn mã được thiết kế để giải quyết các biệt lệ được ném ra. Trình giải quyết lỗi được cài đặt trong khối lệnh bắt đầu bởi từ khóa `catch{}`. Một cách lý tưởng thì khi biệt lệ được bắt và giải quyết thì chương trình tiếp tục thực thi và vấn đề được giải quyết. Ngay cả trong trường hợp chương trình không thể tiếp tục được thì bằng cách bắt biệt lệ ta vẫn còn một cơ hội in (hoặc ghi lại thành tập tin) các thông báo lỗi và kết thúc chương trình một êm đẹp.

Nếu trong hàm có những đoạn mã phải được thực thi bất chấp có hay không có xảy ra biệt lệ (như đoạn mã giải phóng các nguồn lực được cấp phát), đoạn mã này nên được bỏ trong khối lệnh `finnally{}`.

11.1 Ném và bắt biệt lệ

Trong C# chúng ta có thể ném bất kỳ một đối tượng nào thuộc lớp hay lớp con của lớp `System.Exception` (viết tắt là `Exception`). Vùng tên `System` khai báo sẵn nhiều lớp biệt lệ hữu ích chẳng hạn như `ArgumentNullException`, `InValidCastException`, `OverflowException`...

11.1.1 Lệnh ném `throw`

Để báo hiệu một tình huống bất thường trong một lớp C#, ta ném ra một biệt lệ bằng cách sử dụng từ khóa `throw`. Dòng lệnh sau tạo một thể hiện của lớp `Exception` và sau đó ném nó ra

```
throw new System.Exception();
```

Ném một biệt lệ sẽ làm chương trình tạm dừng lập tức và CLR tìm kiếm một trình quản lý biệt lệ. Nếu hàm ném không có trình giải quyết biệt lệ, `stack` sẽ được duyệt ngược (`unwind`) bằng cách `pop` ra cho đến khi gặp được trình giải quyết biệt lệ. Nếu vẫn không tìm thấy cho đến tận hàm `Main()`, chương trình sẽ bị dừng lại.

Ví dụ 11-1. Ném một biệt lệ

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Console.WriteLine("Enter Main...");
      Test t = new Test( );
      t.Func1( );
      Console.WriteLine("Exit Main...");
    }
    public void Func1( )
    {
      Console.WriteLine("Enter Func1...");
      Func2( );
      Console.WriteLine("Exit Func1...");
    }
    public void Func2( )
    {
      Console.WriteLine("Enter Func2...");
      throw new System.Exception( );
      Console.WriteLine("Exit Func2...");
    }
  }
}
Kết quả:
Enter Main...
Enter Func1...
Enter Func2...
Exception occurred: System.Exception: An exception of type
System.Exception was thrown.
at Programming_CSharp.Test.Func2( )
```

```
in ...exceptions01.cs:line 26
at Programming_CSharp.Test.Func1( )
in ...exceptions01.cs:line 20
at Programming_CSharp.Test.Main( )
in ...exceptions01.cs:line 12
```

Ví dụ trên in thông báo ra màn hình `console` khi bắt đầu và kết thúc mỗi hàm. Hàm `Main()` tạo một đối tượng kiểu `Test` và gọi hàm `Func1()`. Sau khi in thông báo `Enter Func1`, hàm `Func1()` gọi hàm `Func2()`. `Func2()` in ra câu thông báo bắt đầu và ném ra một biệt lệ.

Chương trình sẽ tạm ngưng thực thi và CLR tìm trình giải quyết biệt lệ trong hàm `Func2()`. Không có, vùng nhớ `stack` được `unwind` cho đến hàm `Func1()`. Vẫn không có, vùng nhớ `stack` tiếp tục được `unwind` cho đến hàm `Main()`. Vẫn không có, trình giải quyết biệt lệ mặc định được gọi. Thông báo lỗi hiển thị trên màn hình.

11.1.2 Lệnh bắt `catch`

Trình giải quyết biệt lệ đặt trong khối lệnh `catch`, bắt đầu bằng từ khóa `catch`. Trong ví dụ 11-2, lệnh ném `throw` được đặt trong khối lệnh `try`, lệnh bắt đặt trong khối `catch`.

Ví dụ 11-2.Bắt một biệt lệ.

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Console.WriteLine("Enter Main...");
      Test t = new Test( );
      t.Func1( );
      Console.WriteLine("Exit Main...");
    }
    public void Func1( )
    {
      Console.WriteLine("Enter Func1...");
      Func2( );
      Console.WriteLine("Exit Func1...");
    }
    public void Func2( )
    {
      Console.WriteLine("Enter Func2...");
      try
      {
        Console.WriteLine("Entering try block...");
        throw new System.Exception( );
        Console.WriteLine("Exiting try block...");
      }
      catch
      {
```

```
            Console.WriteLine(
              "Exception caught and handled.");
          }
          Console.WriteLine("Exit Func2...");
        }
      }
    }
    Kết quả:
    Enter Main...
    Enter Func1...
    Enter Func2...
    Entering try block...
    Exception caught and handled.
    Exit Func2...
    Exit Func1...
    Exit Main...
```

Ví dụ này y hệt như ví dụ 11-1 ngoại trừ chương trình được đặt trong khối lệnh `try/catch`. Ta đặt các đoạn mã dễ gây lỗi trong khối lệnh `try`, chẳng hạn như đoạn mã truy cập tập tin, xin cấp phát vùng nhớ….

Theo sau khối lệnh `try` là khối lệnh `catch`. Khối lệnh `catch` trong ví dụ là khối lệnh `catch` chung vì ta không thể đoán trước được loại biệt lệ nào sẽ phát sinh. Nếu biết chính xác loại biệt lệ nào phát sinh, ta sẽ viết khối lệnh `catch` cho loại biệt lệ đó (sẽ đề cập ở phần sau).

11.1.2.1 Sửa chữa lỗi lầm

Trong ví dụ 11-2, lệnh bắt `catch` chỉ đơn giản thông báo rằng một biệt lệ đã được bắt và quản lý. Trong ứng dụng thực tế, chúng ta sẽ viết các đoạn mã giải quyết lỗi ở đây. Ví dụ nếu người dùng cố mở một tập chỉ đọc, ta hẳn cho gọi một phương thức cho phép người dùng thay đổi thuộc tính tập tin. Nếu trường hợp hết bộ nhớ, ta hẳn cho người dùng cơ hội đóng các ứng dụng khác. Nếu tất cả đều thất bại, khối lệnh `catch` sẽ cho in các thông báo mô tả chi tiết lỗi để người dùng biết rõ vấn đề.

11.1.2.2 Duyệt lại (`unwind`) vùng nhớ `stack`

Nếu xem kết quả ví dụ 11-2 cẩn thận, ta sẽ thấy các thông báo bắt đầu hàm `Main(), Func1(), Func2()` và khối lệnh `try`; tuy nhiên lại không thấy thông báo kết thúc khối `try` mặc dù nó đã thoát khỏi hàm `Func2(), Func1()` và hàm `Main()`.

Khi một biệt lệ xảy ra, khối `try` ngừng thực thi ngay lập tức và quyền được trao cho khối lệnh `catch`. Nó sẽ không bao giờ quay trở lại khối `try` và vì thế không thể in dòng lệnh thoát khối `try`. Sau khi hoàn tất khối lệnh `catch`, các dòng lệnh sau khối `catch` được thực thi tiếp tục.

Không có khối `catch`, vùng nhớ `stack` được duyệt ngược, nhưng nếu có khối `catch` việc này sẽ không xảy ra. Biệt lệ đã được giải quyết, không còn lỗi nữa,

chương trình tiếp tục thực thi. Điều này sẽ rõ ràng hơn nếu đặt `try/catch` trong hàm `Func1()` như trong ví dụ 11-3

Ví dụ 11-3. Bắt biệt lệ trong hàm gọi.

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Console.WriteLine("Enter Main...");
      Test t = new Test( );
      t.Func1( );
      Console.WriteLine("Exit Main...");
    }
    public void Func1( )
    {
      Console.WriteLine("Enter Func1...");
      try
      {
        Console.WriteLine("Entering try block...");
        Func2( );
        Console.WriteLine("Exiting try block...");
      }
      catch
      {
        Console.WriteLine( "Exception caught and handled." );
      }
      Console.WriteLine("Exit Func1...");
    }
    public void Func2( )
    {
      Console.WriteLine("Enter Func2...");
      throw new System.Exception( );
      Console.WriteLine("Exit Func2...");
    }
  }
}
Kết quả:
Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Exception caught and handled.
Exit Func1...
Exit Main...
```

Bây giờ biệt lệ không được giải quyết trong trong hàm `Func2()`, nó được giải quyết trong hàm `Func1()`. Khi `Func2()` được gọi, nó in dòng `Enter Func2` và sau đó ném một biệt lệ. Chương trình tạm ngừng thực thi, CLR tìm kiếm trình giải quyết biệt lệ trong hàm `Func2()`. Không có. Vùng nhớ `stack` được duyệt ngược và CLR tìm thấy trình giải quyết biệt lệ trong hàm `Func1()`. Khối lệnh `catch` được gọi, chương trình tiếp tục thực thi sau khối lệnh `catch` này, in ra dòng `Exit`

của `Func1()` và sau đó là của `Main()`. Dòng `Exit Try Block` và dòng `Exit Func2` không được in.

11.1.2.3 Tạo một lệnh catch chuyên dụng

Ta có thể tạo một lệnh `catch` chuyên dụng quản lý một loại biệt lệ. Ví dụ 11-4 mô tả cách xác định loại biệt lệ nào ta nên quản lý.

Ví dụ 11-4. Xác định biệt lệ phải bắt

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Test t = new Test( );
      t.TestFunc( );
    }
    // thử chia hai số
    // và giải quyết các biệt lệ
    public void TestFunc( )
    {
      try
      {
        double a = 5;
        double b = 0;
        Console.WriteLine("{0}/{1}={2}", a, b, DoDivide(a,b));
      }
      // các biệt lệ thuộc lớp con phải đứng trước
      catch (System.DivideByZeroException)
      {
        Console.WriteLine("DivideByZeroException caught!");
      }
      catch (System.ArithmeticException)
      {
        Console.WriteLine("ArithmeticException caught!");
      }
      // biệt lệ tổng quát đứng sau cùng
      catch
      {
        Console.WriteLine("Unknown exception caught");
      }
    }
    // thực hiện phép chia hợp lệ
    public double DoDivide(double a, double b)
    {
      if (b == 0)
        throw new System.DivideByZeroException( );
      if (a == 0)
        throw new System.ArithmeticException( );
      return a/b;
    }
  }
}
```

```
Kết quả:
DivideByZeroException caught!
```

Trong ví dụ này, `DoDivide()` sẽ không cho phép chia một số cho 0 hay chia 0 cho số khác. Nó sẽ ném ra biệt lệ `DivideByZeroException` nếu ta cố chia cho không. Nếu ta đem chia 0 cho số khác, sẽ không có biệt lệ thích hợp: vì chia không cho một số là một phép toán hợp lệ và không nên ném bất kỳ biệt lệ nào. Tuy nhiên giả sử trong ví dụ này ta không muốn đem 0 chia cho số khác và sẽ ném ra biệt lệ `ArithmeticException`.

Khi một biệt lệ được ném ra, CLR tìm kiếm trình giải quyết biệt lệ theo theo trình tự, và chọn trình giải quyết phù hợp với biệt lệ. Khi chạy chương trình với `a = 5` và `b = 7`, kết quả là:

```
5 / 7 = 0.7142857142857143
```

Không có biệt lệ nào phát sinh. Tuy nhiên nếu thay `a = 0`, kết quả sẽ là:

```
ArithmeticException caught!
```

Một biệt lệ được ném ra, và CLR xác định trình giải quyết biệt lệ đầu tiên: `DivideByZeroException`. Không đúng, CLR sẽ đi đến trình giải quyết biệt lệ kết tiếp, `ArithmeticException`.

Cuối cùng, nếu `a=7`, và `b=0` biệt lệ `DivideByZeroException` được ném ra.

> *Ghi chú: Bởi vì DevideByZero thừa kế từ ArithmeticException, nên trong ví dụ trên nếu hoán vị hai khối lệnh catch thì có thể khối lệnh catch bắt biệt lệ DivideByZeroException sẽ không bao giờ được thực thi. Thay vào đó khối catch bắt biệt lệ ArithmeticException sẽ bắt các biệt lệ DivideByZeroException. Trình biên dịch sẽ nhận ra điều này và báo lỗi.*

Thông thường hàm sẽ bắt các biệt lệ chuyên dụng cho riêng mục tiêu của hàm, còn các biệt lệ tổng quát hơn sẽ do các hàm cấp cao hơn bắt.

11.1.3 Lệnh finally

Trong một số trường hợp, ném một biệt lệ và `unwind` vùng nhớ `stack` có thể gây thêm vấn đề. Ví dụ như nếu ta đang mở một tập tin hoặc nói chung là đang giữ một tài nguyên nào khác, ta mong muốn có một cơ hội để đóng tập tin hay giải phóng tài nguyên đó.

Trong trường hợp đóng một tập tin đang mở, ta có thể giải quyết bằng cách viết một lệnh đóng ở khối `try` một ở khối `catch` (như vậy lệnh đóng sẽ luôn được gọi). Tuy nhiên đoạn mã này lặp lại một cách vô lý. Mặc khác cách này có thể không giải quyết được nếu ta quyết định không viết khối `catch` ở hàm này mà giao cho hàm gọi xử lý. Khi đó không thể viết lệnh đóng tập tin.

Cách viết đẹp nhất là trong khối `finally`. Khối lệnh này chắc chắn được gọi cho dù có hay không có xảy ra biệt lệ. Ví dụ 11-5 chứng minh cho điều này

Ví dụ 11-5. Sử dụng khối lệnh finally

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Test t = new Test( );
      t.TestFunc( );
    }
    // thử chia hai số
    // và giải quyết các biệt lệ
    public void TestFunc( )
    {
      try
      {
        Console.WriteLine("Open file here");
        double a = 5;
        double b = 0;
        Console.WriteLine ("{0} / {1} = {2}",
                           a, b, DoDivide(a,b));
        Console.WriteLine ("This line may or may not print");
      }
      catch (System.DivideByZeroException)
      {
        Console.WriteLine("DivideByZeroException caught!");
      }
      catch
      {
        Console.WriteLine("Unknown exception caught");
      }
      finally
      {
        Console.WriteLine ("Close file here.");
      }
    }
    // thực hiện phép chia hợp lệ
    public double DoDivide(double a, double b)
    {
      if (b == 0)
        throw new System.DivideByZeroException( );
      if (a == 0)
        throw new System.ArithmeticException( );
      return a/b;
    }
  }
}
Kết quả:
Open file here
DivideByZeroException caught!
Close file here.
Output when b = 12:
Open file here
5 / 12 = 0.41666666666666669
This line may or may not print
Close file here.
```

Trong ví dụ này dòng thông báo `Close file here` luôn luôn xuất hiện, cho dù biệt lệ có xảy ra hay không.

> *Ghi chú: khối lệnh `finally` có thể được tạo mà không cần khối `catch`, nhưng bắt buộc phải có khối `try`. Không thể dùng các lệnh `break`, `continue`, `return` và `goto` trong khối `finally`.*

11.2 Đối tượng Exception

Đối tượng `System.Exception` cung cấp nhiều phương thức và property hữu ích cho việc bẫy lỗi. Chẳng hạn property `Message` cung cấp thông tin tại sao nó được ném. `Message` là thuộc tính chỉ đọc, nó được thiết đặt vào lúc khởi tạo biệt lệ.

Property `HelpLink` cung cấp một kết nối đến tập tin giúp đỡ. Property này có thể đọc và thiết đặt. Property `StackTrace` chỉ đọc và được thiết lập vào lúc chạy.

Trong ví dụ 11-6, property `Exception.HelpLink` được thiết đặt và nhận về để thông tin thêm cho người dùng về biệt lệ `DivideByZeroException`. Property `StackTrace` được dùng để cung cấp các vết của vùng nhớ `stack`. Nó hiển thị hàng loạt các phương thức đã gọi dẫn đến phương thức mà biệt lệ được ném ra.

Ví dụ 11-6. Làm việc với đối tượng Exception

```
using System;
namespace Programming_CSharp
{
  public class Test
  {
    public static void Main( )
    {
      Test t = new Test( );
      t.TestFunc( );
    }
    public void TestFunc( )
    {
      try
      {
        Console.WriteLine("Open file here");
        double a = 12;
        double b = 0;
        Console.WriteLine ("{0} / {1} = {2}",
                           a, b, DoDivide(a,b));
        Console.WriteLine ("This line may or may not print");
      }
      catch (System.DivideByZeroException e)
      {
        Console.WriteLine(
          "\nDivideByZeroException! Msg: {0}", e.Message);
        Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
        Console.WriteLine(
          "\nHere's a stack trace: {0}\n", e.StackTrace);
      }
      catch
```

```
        {
          Console.WriteLine("Unknown exception caught");
        }
        finally
        {
          Console.WriteLine ("Close file here.");
        }
      }
      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 ArithmeticException( );
        return a / b;
      }
    }
}
Kết quả:
Open file here
DivideByZeroException! Msg: Attempted to divide by zero.
HelpLink: http://www.libertyassociates.com
Here's a stack trace:
at Programming_CSharp.Test.DoDivide(Double a, Double b)
in c:\...exception06.cs:line 56
at Programming_CSharp.Test.TestFunc( )
in...exception06.cs:line 22
Close file here.
```

Kết quả liệt kê các phương thức theo trình tự ngược với trình tự chúng được gọi. Đọc kết quả trên như sau: Có một biệt lệ xảy ra tại hàm `DoDivide()`, hàm `DoDivide` này được gọi bởi hàm `TestFunc()`.

Trong ví dụ này ta tạo một thể hiện của `DivideByZeroException`

```
DivideByZeroException e = new DivideByZeroException();
```

Do không truyền tham số, thông báo mặc định được dùng:

```
DivideByZeroException! Msg: Attempted to divide by zero.
```

Ta có thể thay thông báo mặc định này bằng cách truyền tham số khi khởi tạo:

```
new DivideByZeroException(
    "You tried to divide by zero which is not meaningful");
```

Trong trường hợp này kết quả sẽ là:

```
DivideByZeroException! Msg:You tried to divide by zero which is not
meaningful
```

Trước khi ném biệt lệ này, ta thiết đặt thuộc tính `HelpLink`

```
e.HelpLink = "http://www.libertyassociates.com";
```

Khi biệt lệ được bắt, chương trình in thông báo và cả đường dẫn đến kết nối giúp đỡ

```
catch (System.DivideByZeroException e)
```

```
{
  Console.WriteLine("\nDivideByZeroException! Msg: {0}",
                    e.Message);
  Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
```

Nhờ vậy ta có thể cung cấp các thông tin cần thiết cho người dùng. Sau đó là in `StackTrace`

```
Console.WriteLine("\nHere's a stack trace:{0}", e.StackTrace);
```

Ta có kết quả cuối cùng.

11.3 Các biệt lệ tự tạo

Với các biệt lệ có thể tùy biến thông báo do CLR cung cấp, thường đủ cho hầu hết các ứng dụng. Tuy nhiên sẽ có lúc ta muốn thêm nhiều dạng thông tin hơn cho đối tượng biệt lệ, khi đó ta phải tự tạo lấy các biệt lệ mong muốn. Biệt lệ tự tạo bắt buộc thừa kế từ lớp `System.Exception`. Ví dụ 11-7 mô tả cách tạo một biệt lệ mới.

Ví dụ 11-7. Tự tạo biệt lệ

```
using System;
namespace Programming_CSharp
{
  public class MyCustomException : System.ApplicationException
  {
    public MyCustomException(string message) : base(message)
    {
    }
  }
  public class Test
  {
    public static void Main( )
    {
      Test t = new Test( );
      t.TestFunc( );
    }
    public void TestFunc( )
    {
      try
      {
        Console.WriteLine("Open file here");
        double a = 0;
        double b = 5;
        Console.WriteLine("{0}/{1}={2}", a, b, DoDivide(a,b));
        Console.WriteLine("This line may or may not print");
      }
      catch (System.DivideByZeroException e)
      {
        Console.WriteLine("\nDivideByZeroException! Msg: {0}",
                                          e.Message);
        Console.WriteLine("\nHelpLink: {0}\n", e.HelpLink);
      }
      catch (MyCustomException e)
      {
        Console.WriteLine("\nMyCustomException! Msg: {0}",
                                          e.Message);
        Console.WriteLine("\nHelpLink: {0}\n", e.HelpLink);
```

```
            }
            catch
            {
               Console.WriteLine("Unknown exception caught");
            }
            finally
            {
               Console.WriteLine ("Close file here.");
            }
         }
         // do the division if legal
         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;
         }
      }
   }
```

`MyCustomException` thừa kế từ `System.ApplicationException` và nó không có gì khác hơn là một hàm dựng nhận tham số là một thông báo. Câu thông báo này sẽ được chuyển tới lớp cha. Biệt lệ `MyCustomException` được thiết kế cho chính lớp `Test`, không cho phép chia cho 0 và không chia 0 cho số khác. Sử dụng `ArithmeticException` cũng cho kết quả tương tự nhưng có thể gây nhầm lẫn cho lập trình viên khác do phép chia 0 cho một số không phải là một lỗi toán học.

11.4 Ném biệt lệ lần nữa.

Sẽ có trường hợp ta muốn rằng trong khối lệnh `catch` ta sẽ khởi động một hành động sửa lỗi, và sau đó ném biệt lệ cho khối `try` khác (khối `try` của hàm gọi). Biệt lệ này có thể cùng loại hay khác loại với biệt lệ khối `catch` bắt được. Nếu là cùng loại, khối `catch` sẽ ném biệt lệ này một lần nữa; còn nếu khác loại, ta sẽ nhúng biệt lệ cũ vào biệt lệ mới để khối `try` hàm gọi biết được lịch sử của biệt lệ. Property `InnerException` được dủng để thực hiện việc này. Biệt lệ đem nhúng gọi là biệt lệ nội.

Bởi vì `InnerException` cũng chính là một biệt lệ nên nó cũng có `InnerException` của nó. Cứ như vậy tạo nên một loạt các biệt lệ.

Ví dụ 11-8. Ném biệt lệ lần nữa và biệt lệ nội (inner exception)

```
using System;

namespace Programming_CSharp
{
  public class MyCustomException : System.Exception
  {
    public MyCustomException(string message,Exception inner):
           base(message,inner)
    {
    }
  }
  public class Test
  {
    public static void Main()
    {
      Test t = new Test();
      t.TestFunc();
    }
    public void TestFunc()
    {
      try
      {
        DangerousFunc1();
      }
      // khi bắt được biệt lệ tự tạo
      // in lịch sử các biệt lệ
      catch (MyCustomException e)
      {
        Console.WriteLine("\n{0}", e.Message);
        Console.WriteLine("Retrieving exception history...");
        Exception inner = e.InnerException;
        while (inner != null)
        {
          Console.WriteLine("{0}",inner.Message);
          inner = inner.InnerException;
        }
      }
    }
    public void DangerousFunc1( )
    {
      try
      {
        DangerousFunc2( );
      }
      // nếu bắt được một biệt lệ
      // ném một biệt lệ tự tạo
      catch(System.Exception e)
      {
        MyCustomException ex = new MyCustomException(
                        "E3 - Custom Exception Situation!",e);
        throw ex;
      }
    }
    public void DangerousFunc2( )
    {
      try
```

```
        {
            DangerousFunc3( );
        }
        // nếu bắt được biệt lệ DivideByZeroException thực hiện
        // vài công việc sữa lỗi và ném ra biệt lệ tổng quát
        catch (System.DivideByZeroException e)
        {
          Exception ex = new Exception(
                          "E2 - Func2 caught divide by zero",e);
          throw ex;
        }
      }
      public void DangerousFunc3( )
      {
        try
        {
          DangerousFunc4( );
        }
        catch (System.ArithmeticException)
        {
          throw;
        }
        catch (System.Exception)
        {
          Console.WriteLine("Exception handled here.");
        }
      }
      public void DangerousFunc4( )
      {
        throw new DivideByZeroException(
                         "E1 - DivideByZero Exception");
      }
    }
}
Kết quả:
E3 - Custom Exception Situation!
Retrieving exception history...
E2 - Func2 caught divide by zero
E1 - DivideByZeroException
```

Ghi chú: Kết quả xuất hiện trên màn hình không đủ để thể hiện hết ý, cách tốt nhất là nên chạy chương trình ở chế độ từng dòng lệnh để hiểu rõ vấn đề hơn.

Chúng ta bắt đầu bằng lời gọi hàm `DangerousFunc1()` trong khối `try`

```
try
{
  DangerousFunc1( );
}
```

`DangerousFunc1()` gọi `DangerousFunc2()`, `DangerousFunc2()` gọi `DangerousFunc3()`, `DangerousFunc3()` gọi `DangerousFunc4()`. Tất cả các lời gọi này đều có khối `try` của riêng nó. Cuối cùng `DangerousFunc4()` ném một biệt lệ `DivideByZeroException` với câu thông báo `E1 - DivideByZero Exception`.

Khối lệnh `catch` trong hàm `DangerousFunc3()` sẽ bắt biệt lệ này. Theo logic, tất cả các lỗi toán học đều được bắt bởi biệt lệ `ArithmeticException` (vì vậy cả `DivideByZeroException`). Nó chẳng làm gì, chỉ ném biệt lệ này lần nữa.

```
catch (System.ArithmeticException)
{
  throw;
}
```

Cú pháp trên ném cùng một loại biệt lệ cho khối `try` bên ngoài (chỉ cần từ khóa `throw`)

`DangerousFunc2()` sẽ bắt được biệt lệ này, nó sẽ ném ra một biệt lệ mới thuộc kiểu `Exception`. Khi khởi tạo biệt lệ này, ta truyền cho nó hai tham số: thông báo `E2 - Func2 caught divide by zero`, và biệt lệ cũ để làm biệt lệ nội.

`DangerousFunc1()` bắt biệt lệ này, làm vài công việc nào đó, sau đó tạo một biệt lệ có kiểu `MyCustomException`. Tương tự như trên khi khởi tạo biệt lệ ta truyền cho nó hai tham số: thông báo `E3 - Custom Exception Situation!`, và biệt lệ vừa bắt được làm biệt lệ nội. Đến thời điểm này biệt lệ đã có hai mức biệt lệ nội.

Cuối cùng, khối catch sẽ bắt biệt lệ này và in thông báo

```
E3 - Custom Exception Situation!
```

Sau đó sẽ tiếp tục in các thông báo của các biệt lệ nội

```
while (inner != null)
{
  Console.WriteLine("{0}",inner.Message);
  inner = inner.InnerException;
}
```

Và ta có kết quả

```
Retrieving exception history...
E2 - Func2 caught divide by zero
E1 - DivideByZero Exception
```

Chương 12 Delegate và Event

Delegate có nghĩa là ủy quyền hay ủy thác. Trong lập trình đôi lúc ta gặp tình huống phải thực thi một hành động nào đó, nhưng lại không biết sẽ gọi phương thức nào của đối tượng nào. Chẳng hạn, một nút nhấn button khi được nhấn phải thông báo cho đối tượng nào đó biết, nhưng đối tượng này không thể được tiên đoán trong lúc cài đặt lớp button. Vì vậy ta sẽ kết nối lớp button với một đối tượng ủy thác và ủy thác (hay thông báo) cho đối tượng này trách nhiệm thực thi khi button được nhấn. Đối tượng ủy thác sẽ được gán (đăng ký ủy thác) vào thời điểm khác thích hợp.

Event có nghĩa là sự kiện. Ngày nay mô hình lập trình giao diện người dùng đồ họa (Graphical User Interface - GUI) đòi hỏi cách tiếp cận theo hướng sự kiện. Một ứng dụng ngày nay hiển thị giao diện người dùng và chờ người dùng thao tác. Ứng với mỗi thao tác như chọn một trình đơn, nhấn một nút button, nhập liệu vào ô textbox … sẽ một sự kiện sẽ phát sinh. Một sự kiện có nghĩa là có điều gì đó đã xảy ra và chương trình phải đáp trả.

Delegate và event là hai khái niệm có liên quan chặt chẽ với nhau. Bởi vì để quản lý các sự kiện một cách mềm dẻo đòi hỏi các đáp trả phải được phân phối đến các trình giải quyết sự kiện. Trình giải quyết sự kiện trong C# được cài đặt bằng delegate.

Delegate còn được sử dụng như một hàm callback. Hàm callback là hàm có thể được tự động gọi bởi hàm khác. Công dụng thứ hai này của delegate được đề cập trong chương 19.

12.1 Delegate (ủy thác, ủy quyền)

Trong C#, `delegate` được hỗ trợ hoàn toàn bởi ngôn ngữ. Về mặt kỹ thuật, `delegate` thuộc kiểu tham chiếu được dùng để đóng gói phương thức đã xác định kiểu trả về và số lượng, kiểu tham số. Chúng ta có thể đóng gói bất kỳ phương thức thức nào phù hợp với phương thức của `delegate`. (Trong C++ có kỹ thuật tương tự là con trỏ hàm, tuy nhiên `delegate` có tính hướng đối tượng và an toàn về kiểu)

Một `delegate` có thể được tạo bằng từ khóa `delagate`, sau đó là kiểu trả về, tên `delegate` và các tham số của phương thức mà `delegate` chấp nhận:

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

Dòng trên khai báo một `delegate` tên là `WhichIsFirst` có thể đóng gói (nhận) bất kỳ một phương thức nào nhận vào hai tham số kiểu `object` và trả về kiểu `int`.

Khi một `delegate` được định nghĩa, ta có thể đóng gói một phương thức với `delegate` đó bằng cách khởi tạo với tham số là phương thức cho delegate.

12.1.1 Dùng delegate để xác định phương thức vào lúc chạy

Delegate được dùng để xác định (specify) loại (hay kiểu) của các phương thức dùng để quản lý các sự kiện; hoặc để cài đặt các hàm callback trong ứng dụng. Chúng cũng được dùng để xác định các phương thức tĩnh và không tĩnh (còn gọi là phương thức thể hiện - instance methods: là phương chỉ gọi được thông qua một thể hiện của lớp) chưa biết trước vào lúc thiết kế (có nghĩa là chỉ biết vào lúc chạy).

Ví dụ, giả sử chúng ta muốn tạo một lớp túi chứa đơn giản có tên là `Pair` (một cặp). Lớp này chứa 2 đối tượng được sắp xếp. Chúng ta không biết trước được đối tượng nào sẽ được truyền vào cho một thể hiện của lớp `Pair`, vì vậy không thể xây dựng hàm sắp xếp tốt cho tất cả các trường hợp. Tuy nhiên ta sẽ đẩy trách nhiệm này cho đối tượng bằng cách tạo phương thức mà công việc sắp xếp có thể ủy thác. Nhờ đó ta có thể sắp thứ thự của các đối tượng chưa biết bằng cách ủy thác trách nhiệm này chính phương thức của chúng.

Ta định nghĩa một `delegate` có tên `WhichIsFirst` trong lớp `Pair`. Phương thức `sort` sẽ nhận một tham số kiểu `WhichIsFirst`. Khi lớp `Pair` cần biết thứ tự của đối tượng bên trong, nó sẽ gọi delegate với hai đối tượng làm tham số. Trách nhiệm quyết định xem đối tượng nào trong 2 đối tượng có thứ tự trước được ủy thác cho phương thức được đóng gói trong `delegate`.

Để kiểm thử `delegate`, ta tạo ra hai lớp: `Dog` và `Student`. Lớp `Dog` và `Student` không giống nhau ngoại trừ cả hai cùng cài đặt phương thức có thể được đóng gói bởi `WhichIsFirst`, vì vậy cả `Dog` lẫn `Student` đều thích hợp được giữ trong đối tượng `Pair`.

Để kiểm thử chương trình chúng ta tạo ra một cặp đối tượng `Student` và một cặp đối tương `Dog` và lưu trữ chúng trong hai đối tượng `Pair`. Ta sẽ tạo một đối tượng `delegate` để đóng gói cho từng phương thức, sau đó ta yêu cầu đối tượng `Pair` sắp xếp đối tượng `Dog` và `Student`. Sau đây là các bước thực hiện:

```
public class Pair
{
   // cặp đối tượng
   public Pair(object firstObject, object secondObject)
   {
      thePair[0] = firstObject;
      thePair[1] = secondObject;
   }
   // biến lưu giữ hai đối tượng
   private object[]thePair = new object[2];
```

Kế tiếp, ta override hàm `ToString()`

```
public override string ToString( )
{
  return thePair[0].ToString() + ", " + thePair[1].ToString();
```

```
}
```

Chúng ta đã có hai đối tượng trong lớp `Pair` và có thể in chúng ra. Bây giờ là phần sắp xếp chúng và in kết quả sắp xếp. Chúng ta không thể biết trước sẽ có loại đối tượng nào, và vì vậy chúng ta sẽ ủy thác quyền quyết định đối tượng nào có thứ tự trước cho chính các đối tượng. Như vậy ta sẽ yêu cầu đối tượng được xếp thứ tự trong lớp `Pair` phải cài đặt phương thức cho biết trong hai đối tượng, đối tượng nào có thứ tự trước. Phương thức này sẽ nhận vào hai đối tượng (thuộc bất kỳ loại nào) và trả về kiểu liệt kê: `theFirstComeFirst` nếu đối tượng đầu có thứ tự trước và `theSecondComeFirst` nếu đối tượng sau có thứ tự trước.

Những phương thức này sẽ được đóng gói bởi `delegate WhichIsFirst` định nghĩa trong lớp `Pair`.

```
public delegate comparisn WhichIsFirst(object obj1,object obj2)
```

Trị trả về thuộc kiểu kiểu liệt kê `comparison`.

```
public enum comparison
{
  theFirstComesFirst = 1,
  theSecondComesFirst = 2
}
```

Bất kỳ một phương thức tĩnh nào nhận hai tham số kiểu `object` và trả về kiểu `comparison` đều có thể được đóng gói bởi `delegate` này vào lúc chạy.

Bây giờ ta định nghĩa phương thức `Sort` của lớp `Pair`

```
public void Sort(WhichIsFirst theDelegatedFunc)
{
  if ( theDelegatedFunc(thePair[0],thePair[1]) ==
       comparison.theSecondComesFirst )
  {
    object temp = thePair[0];
    thePair[0] = thePair[1];
    thePair[1] = temp;
  }
}
```

Phương thức này nhận một tham số `delegate` tên `WhichIsFirst`. Phương thức `Sort` ủy thác quyền quyết định đối tượng nào có thứ tự trước cho phương thức được đóng gói trong `delegate`. Trong thân hàm `Sort()`, có lời gọi phương thức được ủy thác và xác định giá trị trả về.

Nếu trị trả về là `theSecondComesFirst`, hai đối tượng trong `Pair` sẽ hoán chuyển vị trí, ngược lại không có gì xảy ra.

Chúng ta sẽ xắp xếp các sinh viên theo thứ tự tên. Chúng ta phải viết một phương thức trả về `theFirstComesFirst` nếu tên của sinh viên đầu có thứ tự trước và ngược lại `theSecondComesFirst` nếu tên sinh viên sau có thứ tự trước. Nếu hàm trả về `theSecondComesFirst` ta sẽ thực hiện việc đảo vị trí của hai sinh viên trong `Pair`.

Bây giờ thêm phương thức `ReverseSort`, để sắp các đối tượng theo thứ tự ngược.

```
public void ReverseSort(WhichIsFirst theDelegatedFunc)
{
  if ( theDelegatedFunc(thePair[0], thePair[1]) ==
        comparison.theFirstComesFirst )
  {
    object temp = thePair[0];
    thePair[0] = thePair[1];
    thePair[1] = temp;
  }
}
```

Bây giờ chúng ta cần vài đối tượng để sắp xếp. Ta sẽ tạo hai lớp `Student` và `Dog`. Gán tên cho `Student` lúc khởi tạo

```
public class Student
{
  public Student(string name)
  {
    this.name = name;
  }
```

Lớp `Student` yêu cầu hai phương thức, một `override` từ hàm `ToString()` và một để đóng gói như phương thức được ủy thác.

`Student` phải `override` hàm `ToString()` để phương thức `ToString()` trong lớp `Pair` gọi. Hàm chỉ đơn giản trả về tên của sinh viên.

```
public override string ToString()
{
  return name;
}
```

Cũng cần phải cài đặt phương thức để `Pair.Sort()` có thể ủy thác quyền quyết định thứ tự hai đối tượng.

```
return (String.Compare(s1.name, s2.name) < 0 ?
                  comparison.theFirstComesFirst :
                  comparison.theSecondComesFirst );
```

Hàm `String.Compare` là phương thức của lớp `String` trong thư viện .Net Framework. Hàm so sánh hai chuỗi và trả về số nhỏ hơn 0 nếu chuỗi đầu nhỏ hơn và trả về số lớn hơn 0 nếu ngược lại. Chú ý rằng hàm trả về `theFirstComesFirst` nếu chuỗi đầu nhỏ hơn, và trả về `theSecondComesFirst` nếu chuỗi sau nhỏ hơn.

Lớp thứ hai là `Dog`. Các đối tượng `Dog` sẽ được sắp xếp theo trọng lượng, con nhẹ sẽ đứng trước con nặng. Đây là khai báo đầy đủ lớp `Dog`:

```
public class Dog
{
  public Dog(int weight)
  {
    this.weight=weight;
  }
  // dogs are ordered by weight
```

```
    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;
}
```

Chú ý rằng lớp `Dog` cũng override phương thức `ToString()` và cài đặt phương thức tĩnh với nguyên mẫu hàm được khai báo trong `delegate`. Cũng chú rằng hai phương thức chuẩn bị ủy thác của hai lớp `Dog` và `Student` không cần phải trùng tên. Ví dụ 12 - 1 là chương tình hoàn chỉnh. Chương trình này giải thích cách các phương thức ủy thác được gọi.

Ví dụ 12 - 1. Làm việc với delegate

```
using System;
namespace Programming_CSharp
{
   public enum comparison
   {
      theFirstComesFirst = 1,
      theSecondComesFirst = 2
   }
   // túi chứa đơn giản chứa 2 đối yượng
   public class Pair
   {
      // khai báo delegate
      public delegate comparison WhichIsFirst( object obj1,
                                               object obj2 );

      // hàm khởi tạo nhận 2 đối tượng
      // ghi nhận theo đúng trình tự nhận vào
      public Pair( object firstObject, object secondObject)
      {
         thePair[0] = firstObject;
         thePair[1] = secondObject;
      }
      // phương thức sắp thứ tự (tăng) hai đối tượng
      // theo thứ tự do chính chúng qui định.
      public void Sort(WhichIsFirst theDelegatedFunc)
      {
         if ( theDelegatedFunc(thePair[0],thePair[1]) ==
                comparison.theSecondComesFirst )
         {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }
```

```
    // phương thức sắp thứ tự ngược (giảm) các đối tượng
    // theo thứ tự do chính chúng qui định.
    public void ReverseSort(     WhichIsFirst theDelegatedFunc)
    {
      if (theDelegatedFunc(thePair[0],thePair[1]) ==
                comparison.theFirstComesFirst )
      {
        object temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
      }
    }
    // kết hợp hai hàm ToString() của hai đối tượng
    public override string ToString( )
    {
      return thePair[0].ToString( ) + ", " +
             thePair[1].ToString( );
    }
    // mảng giữ hai đối tượng
    private object[] thePair = new object[2];
}
public class Dog
{
    public Dog(int weight)
    {
      this.weight=weight;
    }
    // chó được sắp theo trọng lượng
    public static comparison WhichDogComesFirst( object o1,
                                      object o2)
    {
      Dog d1 = (Dog) o1;
      Dog d2 = (Dog) o2;
      return d1.weight > d2.weight ?
        comparison.theSecondComesFirst :
        comparison.theFirstComesFirst;
    }
    public override string ToString()
    {
      return weight.ToString();
    }
    private int weight;
}
public class Student
{
    public Student(string name)
    {
      this.name = name;
    }
    // sinh viên sắp theo thứ tự tên
    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 Test
    {
        public static void Main( )
        {
          // tạo hai đối tượng sinh viên và hai đối tượng chó
          // đẩy chúng vào 2 đối tượng Pair
          Student Jesse = new Student("Jesse");
          Student Stacey = new Student ("Stacey");

          Dog Milo = new Dog(65);
          Dog Fred = new Dog(12);

          Pair studentPair = new Pair(Jesse,Stacey);
          Pair dogPair = new Pair(Milo, Fred);
          Console.WriteLine("studentPair\t\t\t: {0}",
            studentPair.ToString( ));
          Console.WriteLine("dogPair\t\t\t\t: {0}",
            dogPair.ToString( ));
          // tạo thể hiện của 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("After Sort studentPair\t\t: {0}",
            studentPair.ToString( ));
          studentPair.ReverseSort(theStudentDelegate);
          Console.WriteLine("After ReverseSort studentPair\t:{0}",
            studentPair.ToString( ));
          dogPair.Sort(theDogDelegate);
          Console.WriteLine("After Sort dogPair\t\t: {0}",
            dogPair.ToString( ));
          dogPair.ReverseSort(theDogDelegate);
          Console.WriteLine("After ReverseSort dogPair\t: {0}",
            dogPair.ToString( ));
        }
    }
}
Kết quả:
studentPair : Jesse, Stacey
dogPair : 65, 12
After Sort studentPair : Jesse, Stacey
After ReverseSort studentPair : Stacey, Jesse
After Sort dogPair : 12, 65
After ReverseSort dogPair : 65, 12
```

Chương trình `test` tạo ra hai đối tượng `Student` và hai đối tượng `Dog`, sau đó đưa chúng vào túi chứa `Pair`. Hàm khởi tạo của `Student` nhận vào tên sinh viên cò hàm khởi tạo `Dog` nhận vào trọng lượng của chó.

```
Student Jesse = new Student("Jesse");
Student Stacey = new Student ("Stacey");
Dog Milo = new Dog(65);
Dog Fred = new Dog(12);
Pair studentPair = new Pair(Jesse,Stacey);
Pair dogPair = new Pair(Milo, Fred);
Console.WriteLine("studentPair\t\t\t:{0}",studentPair.ToString());
Console.WriteLine("dogPair\t\t\t\t: {0}", dogPair.ToString( ));
```

Sau đó in nội dung của hai túi chứa `Pair` để xem thứ tự của chúng. Kết quả như sau:

```
studentPair : Jesse, Stacey
dogPair : 65, 12
```

Như mong đợi thứ tự của các đối tượng là thứ tự chúng được thêm vào túi chứa Pair. Kế tiếp chúng ta khởi tạo hai đối tượng `delegate`

```
Pair.WhichIsFirst theStudentDelegate =
     new Pair.WhichIsFirst( Student.WhichStudentComesFirst );
Pair.WhichIsFirst theDogDelegate =
     new Pair.WhichIsFirst( Dog.WhichDogComesFirst );
```

Ở `delegate` thứ nhất, `theStudentDelegate`, được tạo bằng cách truyền phương thức tĩnh thích hợp từ lớp `Student`. Ở `delegate` thứ hai, `theDogDelegate` được truyền phương thức tĩnh của lớp `Dog`.

Các `delegate` bây giờ có thể được truyền cho các phương thức. Ta truyền `delegate` thứ nhất cho phương thức `Sort()` của đối tượng `Pair`, và sau đó là phương thức `ReverseSort`. Kết quả được in trên màn hình `Console` như sau.

```
After Sort studentPair : Jesse, Stacey
After ReverseSort studentPair : Stacey, Jesse
After Sort dogPair : 12, 65
After ReverseSort dogPair : 65, 12
```

12.1.2 Delegate tĩnh

Điểm bất lợi của ví dụ 12-1 là nó buộc lớp gọi, trong trường hợp này là lớp `Test`, phải khởi tạo các `delegate` nó cần để sắp thứ tự các đối tượng trong một cặp. Sẽ tốt hơn nếu như có thể lấy các `delegate` từ lớp `Dog` và `Student`. Chúng ta có thể làm điều này bằng cách tạo cho trong mỗi lớp một `delegate` tĩnh. Đối với lớp `Student` ta thêm như sau:

```
public static readonly Pair.WhichIsFirst OrderStudents =
          new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

Dòng lệnh này tạo một `delegate` tĩnh, chỉ đọc có tên là `OrderStudent`

Ta có thể tạo tương tự cho lớp `Dog`

```
public static readonly Pair.WhichIsFirst OrderDogs =
              new Pair.WhichIsFirst(Dog. WhichDogComesFirst);
```

Như vậy mỗi lớp có một `delegate` riêng, khi cần ta lấy các `delegate` này và truyền như tham số.

```
studentPair.Sort(theStudentDelegate);
Console.WriteLine("After Sort studentPair\t\t: {0}",
                            studentPair.ToString( ));
studentPair.ReverseSort(Student.OrderStudents);
Console.WriteLine("After ReverseSort studentPair\t: {0}",
                            studentPair.ToString( ));
dogPair.Sort(Dog.OrderDogs);
Console.WriteLine("After Sort dogPair\t\t: {0}",
                            dogPair.ToString( ));
dogPair.ReverseSort(Dog.OrderDogs);
Console.WriteLine("After ReverseSort
                            dogPair.ToString( ));
```

Kết quả hoàn toàn như ví dụ trên.

12.1.3 Delegate như Property

Một vấn đề với `delagate` tĩnh là nó phải được khởi tạo trước, cho dù có được dùng hay không. Ta có thể cải tiến bằng cách thay đổi biến thành viên tĩnh thành property

Đối với lớp `Student`, ta bỏ khai báo sau:

```
public static readonly Pair.WhichIsFirst OrderStudents =
     new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

và thay thế bằng

```
public static Pair.WhichIsFirst OrderStudents
{
  get{ return new Pair.WhichIsFirst(WhichStudentComesFirst); }
}
```

Tương tự thay thế cho lớp `Dog`

```
public static Pair.WhichIsFirst OrderDogs
{
  get{ return new Pair.WhichIsFirst(WhichDogComesFirst);}
}
```

Khi property `OrderStudent` được truy cập, `delegate` sẽ được tạo:

```
return new Pair.WhichIsFirst(WhichStudentComesFirst);
```

Khác biệt chính ở đây là `delegate` sẽ chỉ được khởi tạo khi có yêu cầu.

12.1.4 Thứ tự thực thi với mảng các các delegate

`Delegate` có thể giúp ta xây dựng một hệ thống cho phép người dùng có thể quyết định một cách động trình tự thực thi các thao tác. Giả sử chúng ta có hệ thống sử lý ảnh, hệ thống này có thể thao tác ảnh theo nhiều cách như: làm mờ (blur) ảnh, làm sắc nét, quay, lọc v.v…ảnh. Cũng giả sử rằng trình tự áp dụng các hiệu ứng trên ảnh hưởng lớn đến đến chất lượng của ảnh. Người dùng sẽ mong muốn chọn các hiệu ứng họ lẫn trình tự của chúng từ một thực đơn, sau đó hệ thống sẽ thực hiện các hiệu ứng này theo trình tự họ đã định.

Ta có thể tạo một delegate cho mỗi thao tác (hiệu ứng) và đẩy chúng vào một túi chứa có thứ tự, như một mảng chẳng hạn, theo đúng trình tự nó sẽ được thực thi. Khi tất cả các delegate được tạo và thêm vào túi chứa, ta chỉ đơn giản duyệt suốt mảng, gọi các delegate khi tới lượt.

Ta bắt đầu tạo lớp Image để đại diện cho một bức ảnh sẽ được xử lý bởi ImageProcessor:

```
public class Image
{
  public Image( )
  {
    Console.WriteLine("An image created");
  }
}
```

Lớp ImageProcessor khai báo một delegate không tham số và trả về kiểu void

```
public delegate void DoEffect( );
```

Sau đó khai báo một số phương thức để thao tác ảnh có nguyên mẫu hàm như delegate đã khai báo ở trên.

```
public static void Blur( )
{
  Console.WriteLine("Blurring image");
}
public static void Filter( )
{
  Console.WriteLine("Filtering image");
}
public static void Sharpen( )
{
  Console.WriteLine("Sharpening image");
}
public static void Rotate( )
{
  Console.WriteLine("Rotating image");
}
```

Lớp ImageProccessor cần một mảng để giữ các delegate người dùng chọn; một biến để giữ số lượng hiệu ứng muốn xử lý và hiển nhiên một bức ảnh image

```
DoEffect[] arrayOfEffects;
Image image;
int numEffectsRegistered = 0;
```

ImageProccessor cũng cần một phương thức để thêm delegate vào mảng

```
public void AddToEffects(DoEffect theEffect)
{
  if (numEffectsRegistered >= 10)
  {
    throw new Exception("Too many members in array");
  }
  arrayOfEffects[numEffectsRegistered++] = theEffect;
}
```

Một phương thức để gọi thực thi các hiệu ứng

```
public void ProcessImages( )
{
   for (int i = 0;i < numEffectsRegistered;i++)
   {
      arrayOfEffects[i]( );
   }
}
```

Cuối cùng ta khai báo các `delegate` tĩnh để `client` có thể gọi.

```
public DoEffect BlurEffect = new DoEffect(Blur);
public DoEffect SharpenEffect = new DoEffect(Sharpen);
public DoEffect FilterEffect = new DoEffect(Filter);
public DoEffect RotateEffect = new DoEffect(Rotate);
```

`Client` sẽ có các đoạn mã để tương tác với người dùng, nhưng chúng ta sẽ làm lơ chuyện này, mặc định các hiệu ứng, thêm chúng vào mảng và sau đó gọi `ProcessImage`

Ví dụ 12-2. Sử dụng mảng các deleage

```
using System;
namespace Programming_CSharp
{
   // ảnh ta sẽ thao tác
   public class Image
   {
      public Image( )
      {
         Console.WriteLine("An image created");
      }
   }
   public class ImageProcessor
   {
      // khai báo delegate
      public delegate void DoEffect( );
      // tạo các delegate tĩnh gắn với các phương thức 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 dựng khởi tạo ảng và mảng
      public ImageProcessor(Image image)
      {
         this.image = image;
         arrayOfEffects = new DoEffect[10];
      }
      public void AddToEffects(DoEffect theEffect)
      {
         if (numEffectsRegistered >= 10)
         {
            throw new Exception( "Too many members in array" );
         }
         arrayOfEffects[numEffectsRegistered++] = theEffect;
      }
      // các hiệu ứng ảnh
      public static void Blur( )
```

```
        {
          Console.WriteLine("Blurring image");
        }
          public static void Filter( )
        {
          Console.WriteLine("Filtering image");
        }
        public static void Sharpen( )
        {
          Console.WriteLine("Sharpening image");
        }
        public static void Rotate( )
        {
          Console.WriteLine("Rotating image");
        }
        public void ProcessImages( )
        {
          for (int i = 0;i < numEffectsRegistered;i++)
          {
            arrayOfEffects[i]( );
          }
        }
        // các biến thành viên
        private DoEffect[] arrayOfEffects;
        private Image image;
        private int numEffectsRegistered = 0;
      }
      // lớp kiểm thử
      public class Test
      {
        public static void Main( )
        {
          Image theImage = new Image( );
          // không giao diện để làm đơn giản vấn đề
          ImageProcessor theProc = new ImageProcessor(theImage);
          theProc.AddToEffects(theProc.BlurEffect);
          theProc.AddToEffects(theProc.FilterEffect);
          theProc.AddToEffects(theProc.RotateEffect);
          theProc.AddToEffects(theProc.SharpenEffect);
          theProc.ProcessImages( );
        }
      }
    }
    Kết quả:
    An image created
    Blurring image
    Filtering image
    Rotating image
    Sharpening image
```

Trong lớp `Test`, `ImageProcessor` được khởi tạo và các hiệu ứng được thêm vào. Nếu người dùng chọn làm mờ ảnh (blur) trước khi lọc ảnh (filter), chỉ cần đơn giản thay đổi thứ tự của chúng trong mảng Tương tự, bất kỳ một hiệu ứng nào cũng có thể được lặp lại bằng cách thêm vào túi chứa `delegate` nhiều lần.

12.1.5 Multicasting

Multicasting là cách để gọi hai phương thức thông qua một delegate đơn. Điều này sẽ trở nên quan trọng khi quản lý các sự kiện. Mục tiêu chính là để có một delegate đơn có thể gọi nhiều phương thức cùng một lúc. Nó khác với mảng các delagte, trong mảng delegate mỗi delegate chỉ gọi một phương thức. Ví dụ trước dùng một mảng làm túi chứa nhiều delegate khác nhau.

Với multicasting ta có thể tạo một delegate đơn đóng gói nhiều phương thức. Ví dụ khi một button được nhấn, ta hằn muốn thao tác nhiều hành động cùng một lúc. Ta có thể cài đặt điều này bằng cách cho mỗi button một mảng các delegate, nhưng sẽ dễ hơn và rõ nghĩa hơn khi tạo một multicasting delegate.

Bất kỳ một delegate nào trả về void đều là multicast delegate, mặc dù ta có thể đối xử nó như single cast delegate (là delegate đề cập ở phần trên) nếu muốn. Hai multicast delegate có thể kết nối với nhau bằng toán tử cộng (+). Kết quả là một multicast delegate mới đóng gói tất cả các phương thức của hai delegate toán hạng. Ví dụ, giả sử Writer và Logger là các delegate trả về kiểu void, dòng lệnh sau sẽ kết nối chúng và tạo ra một multicast delegate mới có tên là myMulticastDelegate

```
myMulticastDelegate = Writer + Logger;
```

Ta cũng có thể thêm một delegate vào một multicast delegate bằng toán tử cộng bằng (+=). Giả sử ta có Transmitter và myMulticastDelegate là các delegate, dòng lệnh sau:

```
myMulticastDelegate += Transmitter;
```

tương tự như dòng:

```
myMulticastDelegate = myMulticastDelegate + Transmitter;
```

Để xem cách multicast delegate được tạo và sử dụng, xem qua toàn bộ ví dụ 12-3. Trong ví dụ này ta tạo một lớp tên là MyClassWithDelegate, lớp này định nghĩa một delegate nhận một tham số kiểu chuỗi và trả về kiểu void.

```
public delegate void StringDelegate(string s);
```

Sau đó ta định nghĩa một lớp tên là MyImplementingClass có ba phương thức, tất cả đều trả về void và nhận một tham số kiểu chuỗi: WriteString, LogString và TransmitString. Phương thức đầu viết một chuỗi ra màn hình (đầu ra chuẩn), phương thức thứ hai viết ra tập tin lỗi (log file) và phương thức thứ ba chuyển chuỗi lên Internet. Ta tạo các delegate để gọi các phương thức thích hợp.

```
Writer("String passed to Writer\n");
Logger("String passed to Logger\n");
Transmitter("String passed to Transmitter\n");
```

Để xem cách kết hợp các delegate ta tạo ra một delegate khác

```
MyClassWithDelegate.StringDelegate myMulticastDelegate;
```

và gán nó bằng kết quả của phép cộng hai `delegate` đã tồn tại

```
myMulticastDelegate = Writer + Logger;
```

Ta cũng có thể thêm bằng toán tử cộng bằng

```
myMulticastDelegate += Transmitter;
```

Cuối cùng ta có thể bỏ một `delegate` bằng toán tử trừ bằng (-=)

```
DelegateCollector -= Logger;
```

Ví dụ 12-3. Kết hợp các delegate

```
using System;
namespace Programming_CSharp
{
  public class MyClassWithDelegate
  {
    // khai báo delegate
    public delegate void StringDelegate(string s);
  }
  public class MyImplementingClass
  {
    public static void WriteString(string s)
    {
      Console.WriteLine("Writing string {0}", s);
    }
    public static void LogString(string s)
    {
      Console.WriteLine("Logging string {0}", s);
    }
    public static void TransmitString(string s)
    {
      Console.WriteLine("Transmitting string {0}", s);
    }
  }
  public class Test
  {
    public static void Main( )
    {
      // định nghĩa ba đối tượng StringDelegate
      MyClassWithDelegate.StringDelegate Writer,Logger,Transmitter;
      // định nghĩa một SringDelegate khác
      // hành động như một multicast delegate
      MyClassWithDelegate.StringDelegate myMulticastDelegate;
      // khởi tạo 3 delegate đầu tiên,
      // truyền vào các phương thức định đóng gói
      Writer = new MyClassWithDelegate.StringDelegate(
                    MyImplementingClass.WriteString);
      Logger = new MyClassWithDelegate.StringDelegate(
                    MyImplementingClass.LogString);
      Transmitter = new MyClassWithDelegate.StringDelegate(
                     MyImplementingClass.TransmitString);
            // gọi phương thức của delegate Writer
      Writer("String passed to Writer\n");
      // gọi phương thức của delegate Logger
      Logger("String passed to Logger\n");
      // gọi phương thức của delegate Transmitter
      Transmitter("String passed to Transmitter\n");
      // thông báo kết nối hai deleagte
```

```
            // thành một multicast deleagte
            Console.WriteLine("myMulticastDelegate = Writer + Logger");
            // kết nối hai deleagte
            // thành một multicast deleagte
            myMulticastDelegate = Writer + Logger;
            // gọi delegated, hai phương thức được gọi
            myMulticastDelegate("First string passed to Collector");
            // thông báo thêm deleagte thứ ba
            // vào một multicast deleagte
            Console.WriteLine("\nmyMulticastDelegate += Transmitter");
            // thêm delegate thứ ba
            myMulticastDelegate += Transmitter;
            // gọi delegate, ba phương thức được gọi
            myMulticastDelegate("Second string passed to Collector");
            // thông báo loại bỏ delegate logger
            Console.WriteLine("\nmyMulticastDelegate -= Logger");
            // bỏ delegate logger
            myMulticastDelegate -= Logger;
            // gọi delegate, hai phương htức còn lại được gọi
            myMulticastDelegate("Third string passed to Collector");
        }
    }
}
Kết quả:
Writing string String passed to Writer
Logging string String passed to Logger
Transmitting string String passed to Transmitter
myMulticastDelegate = Writer + Logger
Writing string First string passed to Collector
Logging string First string passed to Collector
myMulticastDelegate += Transmitter
Writing string Second string passed to Collector
Logging string Second string passed to Collector
Transmitting string Second string passed to Collector
myMulticastDelegate -= Logger
Writing string Third string passed to Collector
Transmitting string Third string passed to Collector …
```

Sức mạnh của `multicast delegate` sẽ dễ hiểu hơn trong khái niệm `event`.

12.2 Event (Sự kiện)

Giao diện người dùng đồ họa (Graphic user inteface - GUI), Windows và các trình duyệt yêu cầu các chương trình đáp ứng các sự kiện. Một sự kiện có thể là một `button` được nhấn, một nục thực đơn được chọn, một tập tin đã chuyển giao hoàn tất v.v…. Nói ngắn gọn, là một việc gì đó xảy ra và ta phải đáp trả lại. Ta không thể tiên đoán trước trình tự các sự kiện sẽ phát sinh. Hệ thống sẽ im lìm cho đến khi một sự kiện xảy ra, khi đó nó sẽ thực thi các hành động để đáp trả kiện này.

Trong môi trường GUI, có rất nhiều điều khiển (`control`, `widget`) có thể phát sinh sự kiện Ví dụ, khi ta nhấn một `button`, nó sẽ phát sinh sự kiện `Click`. Khi ta thêm vào một `drop-down list` nó sẽ phát sinh sự kiện `ListChanged`.

Các lớp khác sẽ quan tâm đến việc đáp trả các sự kiện này. Cách chúng đáp trả như thế nào không được quan tâm đến (hay không thể) ở lớp phát sinh sự kiện. Nút button sẽ nói "Tôi được nhấn" và các lớp khác đáp trả phù hợp.

12.2.1 Publishing và Subcribing

Trong C#, bất kỳ một lớp nào cũng có thể phát sinh (`publish`) một tập các sự kiện mà các lớp khác sẽ bắt lấy (`subscribe`). Khi một lớp phát ra một sự kiện, tất cả các lớp `subscribe` đều được thông báo.

Với kỹ thuật này, đối tượng của ta có thể nói "Đây là các vấn đề mà tôi có thể thông báo cho anh biết" và các lớp khác sẽ nói "Vâng, hãy báo cho tôi khi nó xảy ra". Ví dụ như, một `button` sẽ thông báo cho bất ký các lớp nào quan tâm khi nó được nhấn. `Button` được gọi là `publisher` bởi vì button `publish` sự kiện `Click` và các lớp khác sẽ gọi là `subscribers` bởi vì chúng `subscribe` sự kiện `Click`

12.2.2 Event và Delegate

`Event` trong C# được cài đặt bằng `delegate`. Lớp `publish` định nghĩa một `deleagte` mà các lớp `subscribe` phải cài đặt. Khi một sự kiện phát sinh, phương thức của lớp `subscribe` sẽ được gọi thông qua `delegate`.

Cách quản lý các sự kiện được gọi là *event handler (trình giải quyết sự kiện)*. Ta có thể khai báo một `event handler` như là ta đã làm với delegate.

Để thuận tiện, `event handler` trong .NET Framework trả về kiểu `void` và nhận vào 2 tham số. Tham số thứ nhất cho biết nguồn của sự kiện; có nghĩa là đối tượng `publish`. Tham số thứ hai là một đối tượng thừa kế từ lớp `EventArgs`. Có lời khuyên rằng ta nên thiết kế theo mẫu được qui định này.

`EventArgs` là lớp cơ sở cho tất cả các dữ liệu về sự kiện. Ngoại trừ hàm khởi tạo, lớp `EventArgs` thừa kế hầu hết các phương thức của lớp `Object`, mặc dù nó cũng có thêm vào một biến thành viên `empty` đại diện cho một sự kiện không có trạng thái (để cho phép sử dụng có hiệu quả hơn các sự kiện không có trạng thái). Các lớp con thừa kế từ `EventArgs` chứa các thông tin về sự kiện.

Events are properties of the class publishing the event. The keyword event controls how the event
property is accessed by the subscribing classes. The event keyword is designed to maintain the
publish/subscribe idiom.

Giả sử ta muốn tạo một lớp đồng hồ (`Clock`) sử dụng event để thông báo các lớp `subscribe` biết khi nào thời gian thay đổi (theo đơn vị giây). Gọi sự kiện này là `OnSecondChange`. Ta khai báo sự kiện và `event handler` theo cú pháp sau đây:

```
[attributes] [modifiers] event type member-name
```

Ví dụ như:

```
public event SecondChangeHandler OnSecondChange;
```

Ví dụ này không có attribute (attribute sẽ được đề cập trong chương 18). "modifier" có thể là abstract, new, override, static, virtual hoặc là một trong bốn acess modifier, trong trường hợp này là public

Từ khóa event theo sau modifier

type là kiểu delegate liên kết với event, trong trường hợp này là SecondChangeHandler

member name là tên của event, trong trường hợp này là OnSecondChange. Thông thường nó được bắt đầu bằng từ On (không bắt buộc)

Tóm lại dòng lệnh này khai báo một event tên là OnSecondChange, cài đặt một delegate có kiểu là SecondChangeHandler.

Khai báo của SecondChangeHandler là

```
public delegate void SecondChangeHandler( object clock,
                      TimeInfoEventArgs timeInformation );
```

Như đã đề cập, để cho thuận tiện một event handler phải trả về kiểu void và nhận vào hai tham số: nguồn phát sinh sự kiện (trường hợp này là clock) và một đối tượng thừa kế từ lớp EventArgs, trong trường hợp này là TimeInfoEventArgs. TimeInfoEventArgs được khai báo 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;
}
```

Một đối tượng TimeInfoEventArgs sẽ có các thông tin về giờ, phút, giây hiện hành. Nó định nghĩa một hàm dựng và ba biến thành viên kiểu số nguyên (int), public và chỉ đọc.

Lớp Clock có ba biến thành viên hour, minute và second và chỉ duy nhất một phương thức Run():

```
public void Run( )
{
  for(;;)
  {
    // ngủ 10 milli giây
    Thread.Sleep(10);
    // lấy giờ hiện hành
    System.DateTime dt = System.DateTime.Now;
    // nếu biến giây thay đổi
```

```
        // thông báo cho subscriber
        if (dt.Second != second)
        {
          // tạo đối tượng TimeInfoEventArgs
          // để truyền cho subscriber
          TimeInfoEventArgs timeInformation =
          new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);
          // nếu có subscribed, thông báo cho chúng
          if (OnSecondChange != null)
          {
            OnSecondChange(this,timeInformation);
          }
        }
        // cập nhật trạng thái
        this.second = dt.Second;
        this.minute = dt.Minute;
        this.hour = dt.Hour;
      }
    }
```

Hàm `Run` có vòng lặp `for` vô tận luôn luôn kiểm tra giờ hệ thống. Nếu thời gian thay đổi nó sẽ thông báo đến tất cả các `subscriber`.

Đầu tiên là ngủ trong 10 mili giây

```
Thread.Sleep(10);
```

`Sleep` là phương thức tĩnh của lớp `Thread`, thuộc về vùng tên `System.Threading`. Lời gọi `Sleep` nhằm ngăn vòng lặp không sử dụng hết tài nguyên CPU của hệ thống. Sau khi ngủ 10 mili giây, kiểm tra giờ hiện hành

```
System.DateTime dt = System.DateTime.Now;
```

Khoảng sau 100 lần kiểm tra , giá trị giây sẽ tăng. Phương thức sẽ thông báo thay đổi này cho các `subscriber`. Để thực hiện điều này, đầu tiên tạo một đối tượng `TimeInfoEventArgs` mới.

```
if (dt.Second != second)
{
  TimeInfoEventArgs timeInformation =
         new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);
```

Sau đó thông báo cho các `subscriber` bằng cách phát ra sự kiện `OnSecondChange`

```
    if (OnSecondChange != null)
    {
      OnSecondChange(this,timeInformation);
    }
```

Nếu không có `subsrciber` nào đăng ký, `OnSecondChange` có trị `null`, kiểm tra điều này trước khi gọi.

Nhớ rằng `OnSecondChange` nhận 2 tham số: nguồn phát sinh sự kiện và đối tượng thừa kế từ lớp `EventArgs`. Quan sát kỹ ta thấy phương thức dùng từ khóa `this` làm tham số bởi chính `clock` là nguồn phát sinh sự kiện.

Phát sinh một sự kiện sẽ gọi tất cả các phương thức đã đăng ký với `Clock` thông qua `deleagte`. Chúng ta xem xét vấn đề này ngay bây giờ.

Mỗi lần sự kiện phát sinh, ta cập nhật trạng thái của lớp `Clock`:

```
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
```

Vấn đề còn lại là tạo lớp `subcriber`. Ta sẽ tạo ra 2 lớp. Lớp thứ nhất là `DisplayClock`. Lớp này hiển thị thời gian ra màn hình `Console`. Ví dụ này đơn giản tạo ra 2 phương thức, phương thức thứ nhất là `Subscribe` có nhiệm vụ `subscribe` sự kiện `OnSecondChange`. Phương thức thứ hai là một `event handler` tên `TimeHasChanged`

```
public class DisplayClock
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }
  public void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
      ti.hour.ToString( ),
      ti.minute.ToString( ),
      ti.second.ToString( ));
  }
}
```

Khi phương thức đầu, `Subscribe`, được gọi, nó tạo một `delegate SecondChangeHandler` truyền cho phương thức `TimeHasChanged`. Việc này đăng ký `delegate` cho sự kiện `OnSecondChange` của `Clock`

Ta sẽ tạo lớp thứ hai, lớp này cũng sẽ đáp ứng sự kiện, tên là `LogCurrentTime`. Lớp này chỉ đơn giản ghi lại thời gian vào một tập tin, nhưng để đơn giản lớp này cũng xuất ra màn hình `console`.

```
public class LogCurrentTime
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(WriteLogEntry);
  }
  // phương thức sẽ ghi lên tập tin
  // nhưng để đơn giản ta cũng ghi ra console
  public void WriteLogEntry( object theClock,
                             TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
      ti.hour.ToString( ),
      ti.minute.ToString( ),
      ti.second.ToString( ));
```

```
        }
    }
```

Mặc dù trong ví dụ này hai lớp tương tự như nhau, nhưng bất kỳ lớp nào cũng có thể `subscribe` một `event`.

Chú ý rằng `event` được thêm vào bằng toán tử +=. Điều này cho phép các sự kiện mới được thêm vào sự kiện `OnSecondChange` của đối tượng `Clock` mà không làm hỏng đi các sự kiện đã đăng ký trước đó. Khi `LogCurrentTime` `subscribe` vào sự kiện `OnSecondChanged`, ta không cần quan tâm rằng `DisplayClock` đã `subscribe` hay chưa.

Ví dụ 12-4. Làm việc với event

```
using System;
using System.Threading;
namespace Programming_CSharp
{
  // lớp giữ thông tin về một sự kiện
  // trong trường hợp này là thông tin về đồng hồ
  // nhưng tốt hơn là phải có thêm thông tin trạng thái
  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;
  }
  // lớp chính của ta.
  public class Clock
  {
    // delegate mà subscribers phải cài đặt
    public delegate void SecondChangeHandler( object clock,
                          TimeInfoEventArgs timeInformation);
    // sự kiện publish
    public event SecondChangeHandler OnSecondChange;

    // vận hành đồng hồ
    // hàm sẽ phát sinh sự kiện sau mỗi giây
    public void Run( )
    {
      for(;;)
      {
        // ngủ 10 milli giây
        Thread.Sleep(10);
        // lấy giờ hiện tại
        System.DateTime dt = System.DateTime.Now;
        // nếu thời gian thay đổi
        // thông báo cho các subscriber
        if (dt.Second != second)
        {
```

```
            // tạo đối tượng TimeInfoEventArgs
            // để truyền cho subscriber
            TimeInfoEventArgs timeInformation=new TimeInfoEventArgs(
                                    dt.Hour,dt.Minute,dt.Second);
            // nếu có subscriber, thông báo cho chúng
            if (OnSecondChange != null)
            {
              OnSecondChange( this,timeInformation );
            }
          }
          // cập nhật trạng thái
          this.second = dt.Second;
          this.minute = dt.Minute;
          this.hour = dt.Hour;
        }
      }
      private int hour;
      private int minute;
      private int second;
    }
    public class DisplayClock
    {
      // subscribe sự kiện SecondChangeHandler của theClock
      public void Subscribe(Clock theClock)
      {
        theClock.OnSecondChange +=
          new Clock.SecondChangeHandler(TimeHasChanged);
      }
      // phương thức cài đặt hàm delegated
      public void TimeHasChanged( object theClock,
                              ` TimeInfoEventArgs ti)
      {
        Console.WriteLine("Current Time: {0}:{1}:{2}",
                  ti.hour.ToString( ),
                  ti.minute.ToString( ),
                  ti.second.ToString( ));
      }
    }
    public class LogCurrentTime
    {
      public void Subscribe(Clock theClock)
      {
          theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(WriteLogEntry);
      }

      // phương thức này nên viết lên tập tin
      // nhưng để đơn giản ta xuất ra màn hình console
      public void WriteLogEntry(object theClock,TimeInfoEventArgs ti)
      {
        Console.WriteLine("Logging to file: {0}:{1}:{2}",
          ti.hour.ToString( ),
          ti.minute.ToString( ),
          ti.second.ToString( ));
      }
    }
    public class Test
    {
```

```
        public static void Main( )
        {
           // tạo đồng hồ mới
           Clock theClock = new Clock( );
           // tạo một displayClock
           // subscribe với clock vừa tạo
           DisplayClock dc = new DisplayClock( );
           dc.Subscribe(theClock);
           // tạo đối tượng Log
           // subscribe với clock vừa tạo
           LogCurrentTime lct = new LogCurrentTime( );
           lct.Subscribe(theClock);
           // bắt đầu chạy
           theClock.Run( );
        }
     }
}
Kết quả:
Current Time: 14:53:56
Logging to file: 14:53:56
Current Time: 14:53:57
Logging to file: 14:53:57
Current Time: 14:53:58
Logging to file: 14:53:58
Current Time: 14:53:59
Logging to file: 14:53:59
Current Time: 14:54:0
Logging to file: 14:54:0
```

12.2.3 Tách rời Publisher khỏi Subsciber

Lớp `Clock` chỉ nên đơn giản in thời gian hơn là phải phát sinh sự kiện, vậy tại sao phải bị làm phiền bằng việc sử dụng gián tiếp delegate? Thuận lợi của ý tưởng `publish/subscribe` là bất kỳ lớp nào (bao nhiêu cũng được) cũng có thể được thông báo khi một sự kiện phát sinh. Lớp `subscribe` không cần phải biết cách làm việc của `Clock`, và `Clock` cũng không cần biết chuyện sẽ xảy ra khi một sự kiện được đáp trả. Tương tự một `button` có thể phát ra sự kiện `OnClick` và bất kỳ lớp nào cũng có thể `subscribe` sự kiện này, nhận về thông báo khi nào `button` bị nhấn.

`Publisher` và `Subscriber` được tách biệt nhờ `delegate`. Điều này được mong chờ nhất vì nó làm cho mã nguồn được mềm dẻo (flexible) và dễ hiểu. Lớp `Clock` có thể thay đổi cách nó xác định thời gian mà không ảnh hưởng tới các lớp `subscriber`. Tương tự các lớp `subscriber` cũng có thể thay đổi cách chúng đáp trả sự kiện mà không ảnh hưởng tới lớp `Clock`. Hai lớp này hoàn toàn độc lập với nhau, và nó giúp cho mã nguồn dễ bảo trì hơn.

Chương 13 Lập trình với C#

Phần này sẽ giới thiệu chi tiết về cách viết các chương trình .NET, bao gồm Windows Forms và Web Forms. Ngoài ra, chúng ta sẽ khảo sát thêm về cách tương tác với cơ sở dữ liệu (*Database*) và các dịch vụ Web (*Web Services*).

Quan điểm về kiến trúc .NET là tạo sự dễ dàng, thuận tiện khi phát triển các phần mềm theo tính hướng đối tượng. Với mục đích này, tầng trên cùng của kiến trúc .NET được thiết kế để bao gồm hai phần: ASP.NET và Windows Form. ASP.NET được dùng cho hai mục đích chính: hoặc để tạo các ứng dụng Web với Web Forms hoặc tạo các đối tượng Web (Web Objects) không có giao diện người dùng (User Interface: UI) với Web Services.

Ta sẽ khảo sát chi tiết các mục chính sau :

1. Cách tạo ra các ứng dụng Windows có tính chuyên nghiệp cao trong môi trường phát triển Windows Form một cách nhanh chóng theo mô hình RAD (Rapid Application Development).
2. Để có thể truy cập dữ liệu trong các hệ quản trị dữ liệu, ta sẽ thảo luận chi tiết về ADO.NET và cách tương tác với Microsoft SQL Server và các trình cung cấp dữ liệu (Providers Data) khác.
3. Là sự kết hợp công nghệ RAD trong phần (1) và ADO.NET trong phần (2) để minh họa việc tạo ra các ứng dụng Web với Web Forms.
4. Không phải tất cả mọi ứng dụng đều có giao diện người dùng thân thiện. Web Services giúp tạo các ứng dụng như vậy, chúng là những ứng dụng có tính phân phối, cung cấp các chức năng dựa trên các nghi thức Web chuẩn, thường dùng nhất là XML và HTTP.

13.1 Ứng dụng Windows với Windows Form

Trước tiên, chúng ta cần phân biệt sự khác nhau giữa hai kiểu ứng dụng: Windows và Web. Khi các ứng dụng Web đầu tiên được tạo ra, người ta phân biệt hai loại ứng dụng trên như sau : ứng dụng Windows chạy trên Desktop hay trên một mạng cục bộ LAN (Local-Area Network), còn ứng dụng Web thì được chạy trên Server ở xa và được truy cập bằng trình duyệt Web (web browser). Sự phân biệt này không còn rõ ràng nữa vì các ứng dụng Windows hiện nay có xu hướng dùng các dịch vụ của Web. Ví dụ như phần mềm Outlook chuyển nhận thư thông qua kết nối Web.

Theo quan điểm của Jesse Liberty, tác giả của cuốn sách "**Programming C#**", xuất bản vào tháng 7 năm 2001. Ông cho rằng điểm phân biệt chủ yếu giữa ứng dụng Windows và Web là ở chỗ : Cái gì sở hữu *UI* ?, Ứng dụng dùng trình duyệt để hiển

thị hay *UI* của ứng dụng được xây dựng thành chương trình có thể chạy trên Desktop.

Có một số thuận lợi đối với các ứng dụng Web, ứng dụng có thể được truy cập bởi bất kỳ trình duyệt nào kết nối đến Server, việc hiệu chỉnh được thực hiện trên Server, không cần phải phân phối thư viện liên kết động (*Dynamic Link Libraries - DLLs*) mới cần để chạy ứng dụng cho người dùng.

.NET cũng có sự phân biệt này, điển hình là có những bộ công cụ thích hợp cho từng loại ứng dụng: Windows hay Web. Cả hai loại này đều dựa trên khuôn mẫu Form và sử dụng các điều khiển (Control) như là Buttons, ListBox, Text …

Bộ công cụ dùng để tạo ứng dụng Web được gọi là Web-Form, được thảo luận trong mục (3). Còn bộ công cụ dùng để tạo ứng dụng Windows được gọi là Windows-Form, sẽ được thảo luận ngay trong mục này.

> *Chú ý : Theo tác giả JesseLiberty, ông cho rằng hiện nay ứng dụng kiểu Windows và Web có nhiều điểm giống nhau, và ông cho rằng .NET nên gộp lại thành một bộ công cụ chung cho cả ứng dụng Windows và Web trong phiên bản tới.*

Trong các trang kế, chúng ta sẽ học cách tạo một Windows Form đơn giản bằng cách dùng trình soạn mã hoặc công cụ thiết kế (Design Tool) trong Visual Studio .NET. Kế tiếp ta sẽ khảo sát một ứng dụng Windows khác phức tạp hơn, ta sẽ học các dùng bộ công cụ kéo thả của Visual Studio .NET và một số kỹ thuật lập trình C# mà ta đã thảo luận trong phần trước.

13.1.1 Tạo một Windows Form đơn giản

Windows Form là công cụ dùng để tạo các ứng dụng Windows, nó mượn các ưu điểm mạnh của ngôn ngữ Visual Basic : dễ sử dụng, hỗ trợ mô hình RAD đồng thời kết hợp với tính linh động, hướng đối tượng của ngôn ngữ C#. Việc tạo ứng dụng Windows trở lên hấp dẫn và quen thuộc với các lập trình viên.

Trong phần này, ta sẽ thảo luận hai cách khi tạo một ứng dụng Windows : Dùng bộ soạn mã để gõ mã trực tiếp hoặc dùng bộ công cụ kéo thả của IDE.

Ứng dụng của chúng ta khi chạy sẽ xuất dòng chữ "**Hello World!**" ra màn hình, khi người dùng nhấn vào Button "**Cancel**" thì ứng dụng sẽ kết thúc.

13.1.1.1 Dùng bộ soạn mã (Nodepad)

Mặc dù Visual Studio .NET cung cấp một bộ các công cụ phục vụ cho việc kéo thả, giúp tạo các ứng dụng Windows một các nhanh chóng và hiệu quả, nhưng trong phần này ta chỉ cần dùng bộ soạn mã.

Hình 13-1 Ứng dụng minh họa việc hiển thị chuỗi và bắt sự kiện của Button.

Đầu tiên, ta dùng lệnh **using** để thêm vùng tên sau :

```
using System.Windows.Forms;
```

Ta sẽ cho ứng dụng của ta thừa kế từ vùng tên Form :

```
public class HandDrawnClass : Form
```

Bất kỳ một ứng dụng Windows Form nào cũng đều thừa kế từ đối tượng **Form**, ta có thể dùng đối tượng này để tạo ra các cửa sổ chuẩn như : các cửa sổ trôi (floating form), thanh công cụ (tools), hộp thoại (dialog box) … Mọi *Điều khiển* trong bộ công cụ của Windows Form (Label, Button, Listbox …) đều thuộc vùng tên này.

Ta sẽ khai báo 2 đối tượng, một *Label* để giữ chuỗi ' **Hello World !**' và một *Button* để bắt sự kiện kết thúc ứng dụng.

```
private System.Windows.Forms.Label lblOutput;
private System.Windows.Forms.Button btnCancel;
```

Tiếp theo ta sẽ khởi tạo 2 đối tượng trên trong hàm khởi tạo của Form:

```
this.lblOutput = new System.Windows.Forms.Label( );
this.btnCancel = new System.Windows.Forms.Button( );
```

Sau đó ta gán chuỗi tiêu đề cho Form của ta là '**Hello World**' :

```
this.Text = "Hello World";
```

> *Chú ý :Do các lệnh trên được đặt trong hàm khởi tạo của Form* ***HandDrawClass****, vì thế từ khóa this sẽ tham chiếu tới chính nó.*

Gán vị trí, chuỗi và kích thước cho đối tượng Label :

```
lblOutput.Location = new System.Drawing.Point (16, 24);
lblOutput.Text = "Hello World!";
lblOutput.Size = new System.Drawing.Size (216, 24);
```

Vị trí của Label được xác định bằng một đối tượng Point, đối tượng này cần hai thông số : vị trí so với chiều ngang (horizontal) và đứng (vertical) của thanh cuộn. Kích thước của Label cũng được đặt bởi đối tượng Size, với hai thông số là chiều rộng (width) và cao (height) của Label. Cả hai đối tượng Point và Size đều thuộc vùng tên System.Drawing : chứa các đối tượng và lớp dùng cho đồ họa.

Tương tự làm với đối tượng Button :

```
btnCancel.Location = new System.Drawing.Point (150,200);
btnCancel.Size = new System.Drawing.Size (112, 32);
btnCancel.Text = "&Cancel";
```

Để bắt sự kiện click của Button, đối tượng Button cần đăng ký với trình quản lý sự kiện, để thực hiện điều này ta dùng '**delegate**'. Phương thức được ủy thác (sẽ bắt sự kiện) có thể có tên bất kỳ nhưng phải trả về kiểu void và phải có hai thông số : một là đối tượng '**sender**' và một là đối tượng '**System.EventArgs**'.

```
protected void btnCancel_Click( object sender, System.EventArgs e)
{
  //...
}
```

Ta đăng ký phương thức bắt sự kiện theo hai bước. Đầu tiên, ta tạo một trình quản lý sự kiện mới System.EventHandler, rồi đẩy tên của phương thức bắt sự kiện vào làm tham số :

```
new System.EventHandler (this.btnCancel_Click);
```

Tiếp theo ta sẽ ủy thác trình quản lý vừa tạo ở trên cho sự kiện click của Button bằng toán tử +=

Mã gộp của hai bước trên :

```
one:btnCancel.Click +=new System.EventHandler
(this.btnCancel_Click);
```

Để kết thúc việc viết mã trong hàm khởi tạo của Form, ta sẽ thêm hai đối tượng Label và button vào Form của ta :

```
this.Controls.Add (this.btnCancel);
this.Controls.Add (this.lblOutput);
```

Sau khi ta đã định nghĩa hàm bắt sự kiện click trên Button, ta sẽ viết mã thi hành cho hàm này. Ta sẽ dùng hàm tĩnh (static) Exit() của lớp Application để kết thúc ứng dụng :

```
protected void btnCancel_Click( object sender, System.EventArgs e)
{
  Application.Exit();
}
```

Cuối cùng, ta sẽ gọi hàm khởi tạo của Form trong hàm Main(). Hàm Main() là điểm vào đầu tiên của Form.

```
public static void Main( )
{
  Application.Run(new HandDrawnClass( ));
}
```

Sau đây là mã hoàn chỉnh của toàn bộ ứng dụng

```
using System;
using System.Windows.Forms;
namespace ProgCSharp
{
  public class HandDrawnClass : Form
  {
    // Label dùng hiển thị chuỗi 'Hello World'
    private System.Windows.Forms.Label lblOutput;

    // Button nhấn 'Cancel'
    private System.Windows.Forms.Button btnCancel;
```

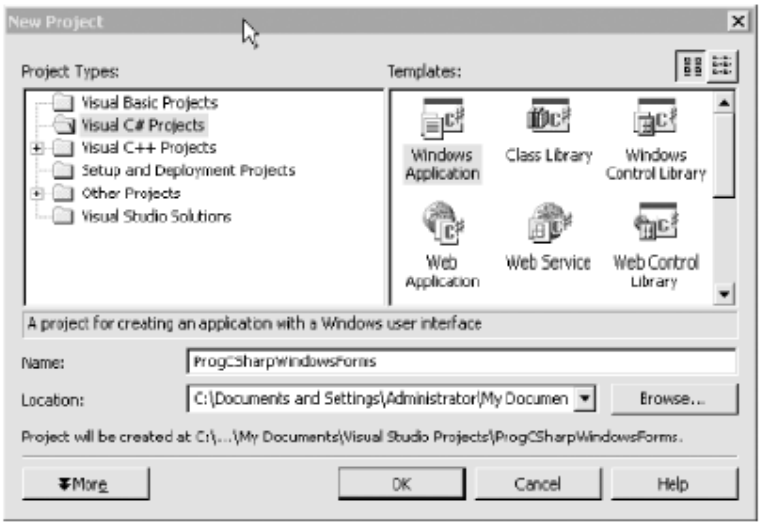
```
        public HandDrawnClass( )
        {
          // Tạo các đối tượng
          this.lblOutput = new System.Windows.Forms.Label ( );
          this.btnCancel = new System.Windows.Forms.Button ( );

          // Gán tiêu đề cho Form
          this.Text = "Hello World";

          // Hiệu chỉnh Label
          lblOutput.Location = new System.Drawing.Point(16,24);
          lblOutput.Text = "Hello World!";
          lblOutput.Size = new System.Drawing.Size (216, 24);

          // Hiệu chỉnh Button
          btnCancel.Location = newSystem.Drawing.Point(150,20);
          btnCancel.Size = new System.Drawing.Size (112, 32);
          btnCancel.Text = "&Cancel";

          // Đăng ký trình quản lý sự kiện
          btnCancel.Click +=
            new System.EventHandler (this.btnCancel_Click);

          //Thêm các điều khiển vào Form
          this.Controls.Add (this.btnCancel);
          this.Controls.Add (this.lblOutput);
        }

        // Bắt sự kiện nhấn Button
        protected void btnCancel_Click(object sender, EventArgs e)
        {
          Application.Exit( );
        }

        // Chạy ứng dụng
        public static void Main()
        {
          Application.Run(new HandDrawnClass( ));
        }
      }
    }
```

13.1.1.2 Dùng kéo thả trong Visual Studio .NET

Bên cạnh trình soạn mã, .NET còn cung cấp một bộ các công cụ kéo thả để làm việc trong môi trường phát triển tích hợp *IDE* (*Intergrate Development Enviroment*), IDE cho phép kéo thả rồi tự động phát sinh mã tương ứng.

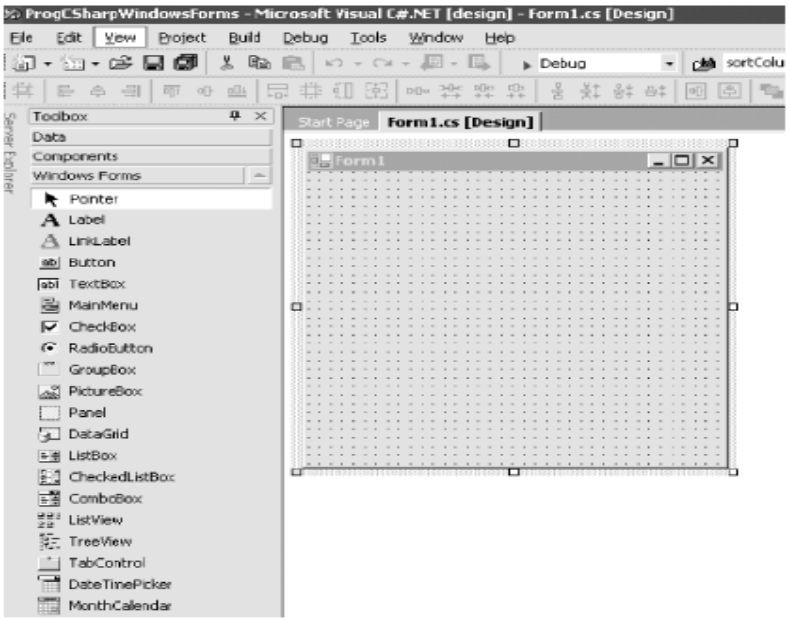
Ta sẽ tạo lại ứng dụng trên bằng cách dùng bộ công cụ trong Visual Studio, ta mở Visual Studio và chọn 'New Project'. Trong cửa sổ 'New Project', chọn loại dự án là Visual C# và kiểu ứng dụng là 'Windows Applications', đặt tên cho ứng dụng là `ProgCSharpWindowsForm`.

Hình 13-2 Màn hình tạo ứng dụng Windows mới.



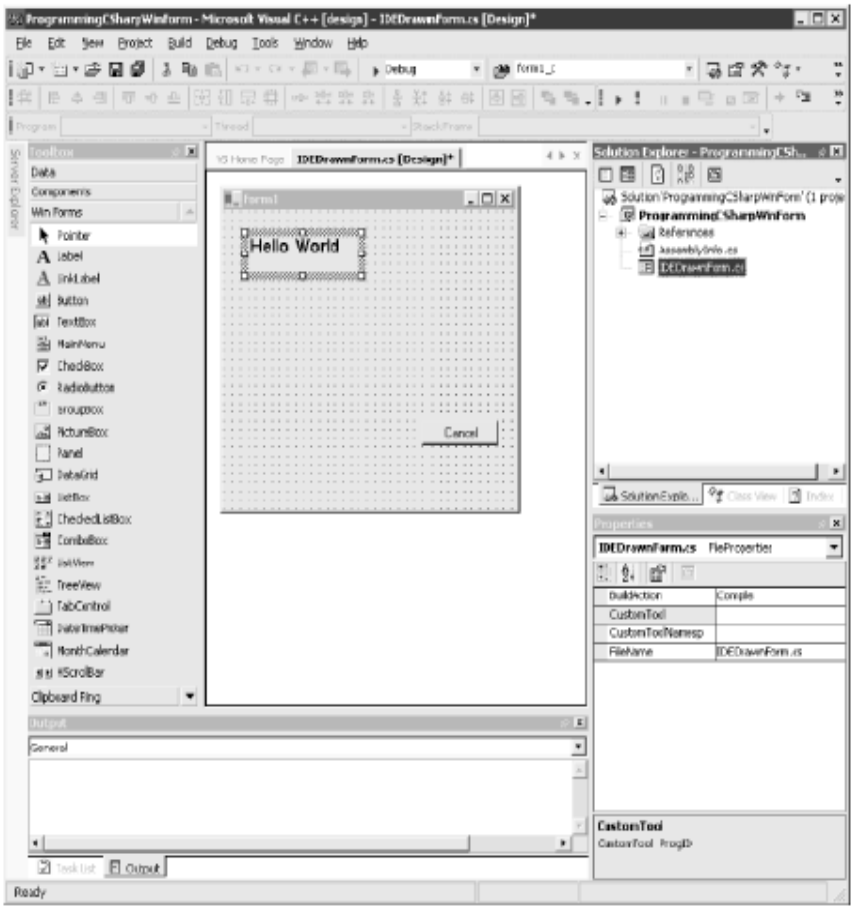
Vs.NET sẽ tạo một ứng dụng Windows mới và đặt chúng vào *IDE* như hình dưới :

Hình 13-3 Môi trường thiết kế kéo thả



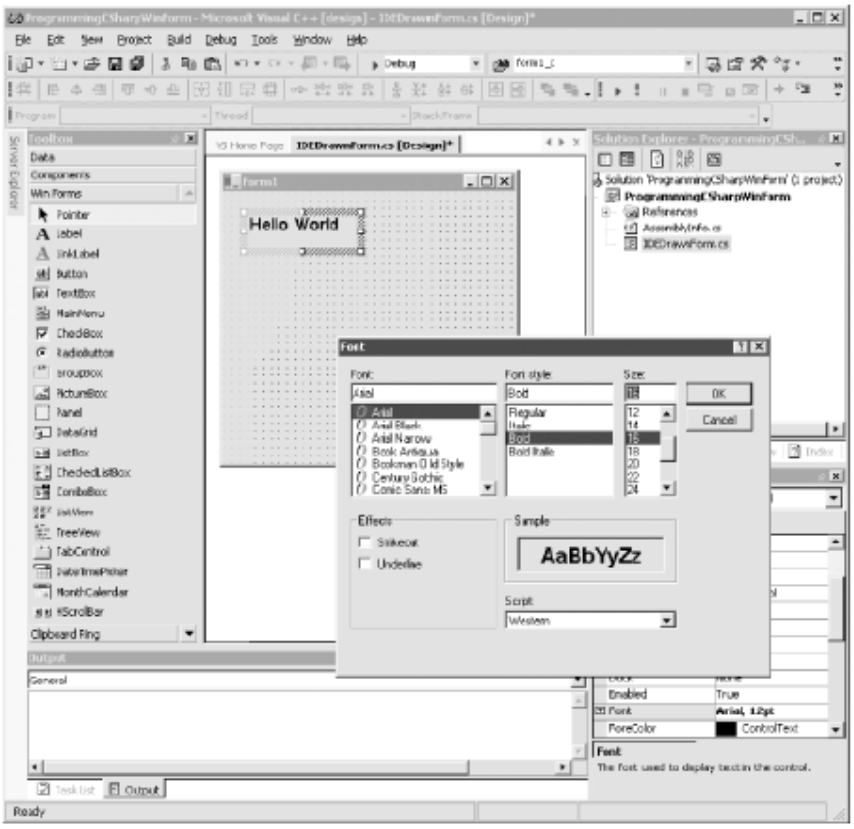
Phía bên trái của cửa hình trên là một bộ các công cụ (Toolbox) kéo thả dành cho các ứng dụng Windows Form, chính giữa là một Form được .NET tạo sẵn có tên Form1. Với bộ công cụ trên, ta có thể kéo và thả một Label hay Button trực tiếp vào Form, như hình sau :

Hình 13-4 Môi trường phát triển Windows Form.



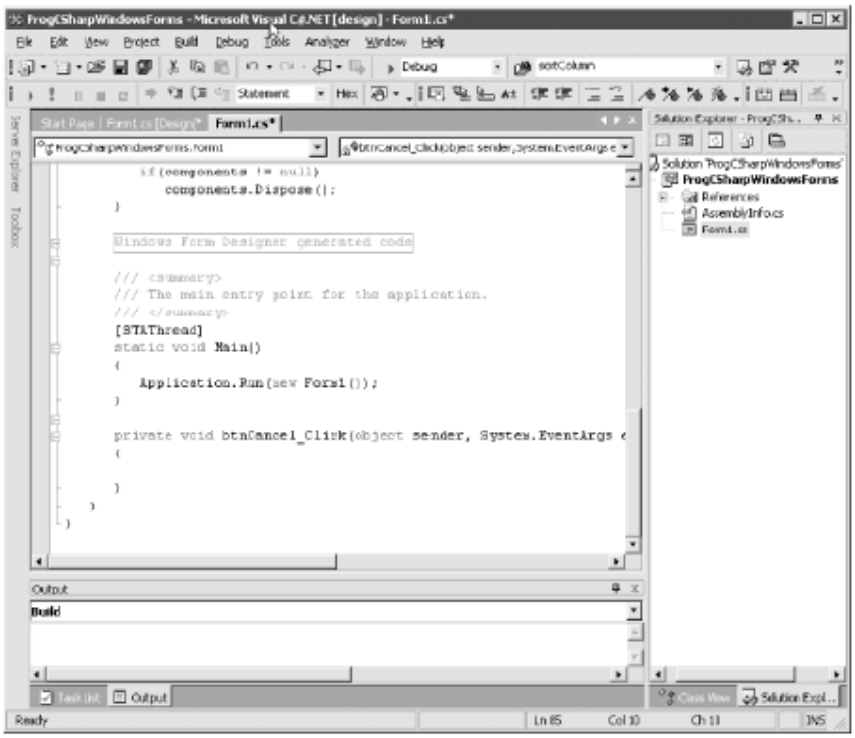
Với thanh công cụ Toolbox ở bên trái, ta có thể thêm các thành phần mới vào nó bằng các chọn *View/Add Reference*. Góc bên phải phía trên là cửa sổ duyệt toàn bộ các tập tin trong giải pháp (*Solution*, một giải pháp có một hay nhiều dự án con). Phía dưới là cửa sổ thuộc tính, hiển thị mọi thuộc tính về mục chọn hiện hành. Ta có thể gán giá trị chuỗi hiển thị hoặc thay đổi font cho Label một cách trực tiếp trong cửa sổ thuộc tính.

Hình 13-5 Thay đổi font trực tiếp bằng hộp thoại font.



Với IDE này, ta có thể kéo thả một Button và bắt sự kiện click của nó một cách dễ dàng, chỉ cần Nhấn đúp vào Button thì tự động .NET sẽ phát sinh ra các mã tương ứng trong trang mã của Form (*Code-Behind page*) như : khai báo, tạo Button và hàm bắt sự kiện click của Button.

Hình 13-6 Sau khi nhấn đúp vào nút Cancel.



Bây giờ, ta chỉ cần gõ thêm một dòng code nữa trong hàm bắt sự kiện của Button là ứng dụng có thể chạy được y như ứng dụng mà ta đã tạo bằng cách gõ code trong phần trên.

```
Application.Exit( );
```

Sau đây là toàn bộ mã được phát sinh bởi IDE và dòng mã bạn mới gõ vào :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace ProgCSharpWindowsForm
{

   /// <summary>
   /// Summary description for Form1.
   /// </summary>
   public class Form1 : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label lblOutput;
      private System.Windows.Forms.Button btnCancel;

      /// <summary>
      /// Required designer variable.
      /// </summary>
```

```
        private System.ComponentModel.Container components;

        public Form1( )
        {
           InitializeComponent( );
        }

        public override void Dispose( )
        {
           base.Dispose( );
           if(components != null)
              components.Dispose( );
        }

    #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.lblOutput = new System.Windows.Forms.Label( );
           this.btnCancel = new System.Windows.Forms.Button( );
           this.SuspendLayout( );

           //
           // lblOutput
           //
           this.lblOutput.Font = new System.Drawing.Font("Arial",
                15.75F, System.Drawing.FontStyle.Bold,
                System.Drawing.GraphicsUnit.Point,((System.Byte)(0)));
           this.lblOutput.Location =  new System.Drawing.Point(24, 16);
           this.lblOutput.Name = "lblOutput";
           this.lblOutput.Size = new System.Drawing.Size(136, 48);
           this.lblOutput.TabIndex = 0;
           this.lblOutput.Text = "Hello World";

           // btnCancel
           this.btnCancel.Location = new System.Drawing.Point(192, 208);
           this.btnCancel.Name = "btnCancel";
           this.btnCancel.TabIndex = 1;
           this.btnCancel.Text = "Cancel";
           this.btnCancel.Click +=
                new System.EventHandler( this.btnCancel_Click );
           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.btnCancel, this.lblOutput});
           this.Name = "Form1";
           this.Text = "Form1";
           this.ResumeLayout(false);
        }

        private void btnCancel_Click(object sender, System.EventArgs e)
        {
           Application.Exit( );
        }
    #endregion
```

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main( )
        {
          Application.Run(new Form1( ));
        }
    }
}
```

So với đoạn mã ta gõ vào trong ứng dụng trước thì mã do IDE phát sinh không khác gì nhiều. Các dòng chú thích được dùng để làm sưu liệu báo cáo cho dự án. (mục này sẽ được thảo luận sau)

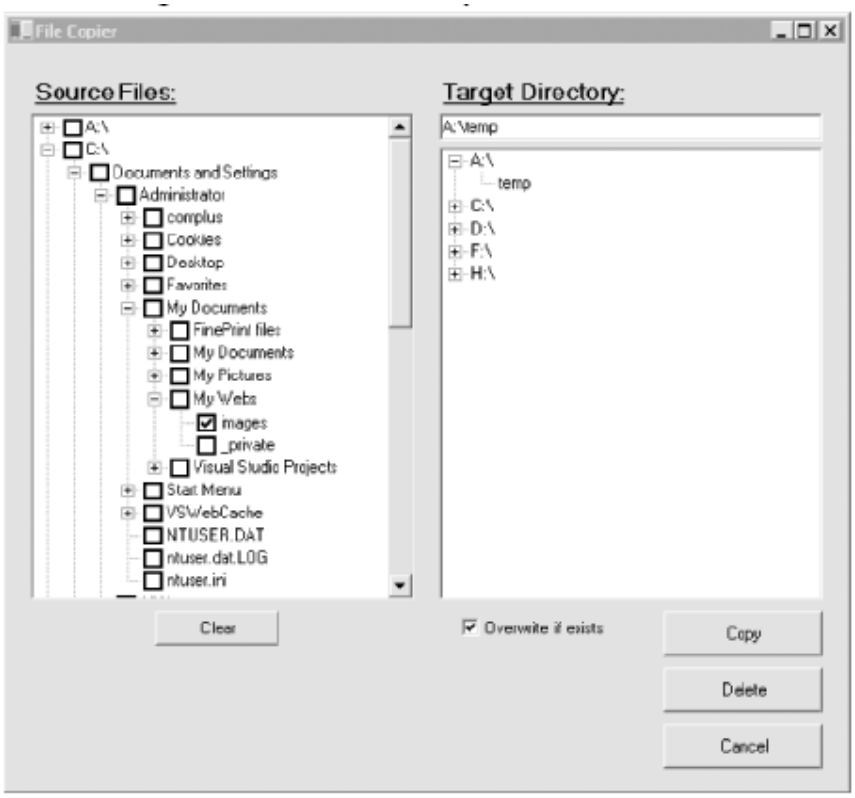
```
/// <summary>
/// Summary description for Form1.
/// </summary>
```

Các mã tạo và hiệu chỉnh đối tượng thay vì được đặt trực tiếp vào hàm khởi tạo của Form, thì ở đây IDE đặt chúng vào trong hàm `InitializeComponent()`, Sau đó hàm này được gọi bởi hàm khởi tạo của Form. Mọi ứng dụng Windows Form đều phát sinh ra hàm này.

13.1.2 Tạo một ứng dụng Windows Form khác

Trong ứng dụng trên ta đã thảo luận sơ qua về ứng dụng Windows Form, phần này ta sẽ tạo một ứng dụng Windows khác thực tế hơn. Ứng dụng có tên là `FileCopier`, cho phép chép hay xóa một hoặc nhiều tập tin từ vị trí này sang vị trí khác. Mục đích của ứng dụng là minh họa sâu hơn về các kỹ năng lập trình C# và giúp người đọc hiểu thêm về namespace `Windows.Forms`. Giao diện của ứng dụng sau khi hoàn chỉnh sẽ như sau :

Hình 13-7 Giao diện người dùng của ứng dụng FileCopier.



Giao diện của ứng dụng gồm các thành phần sau :

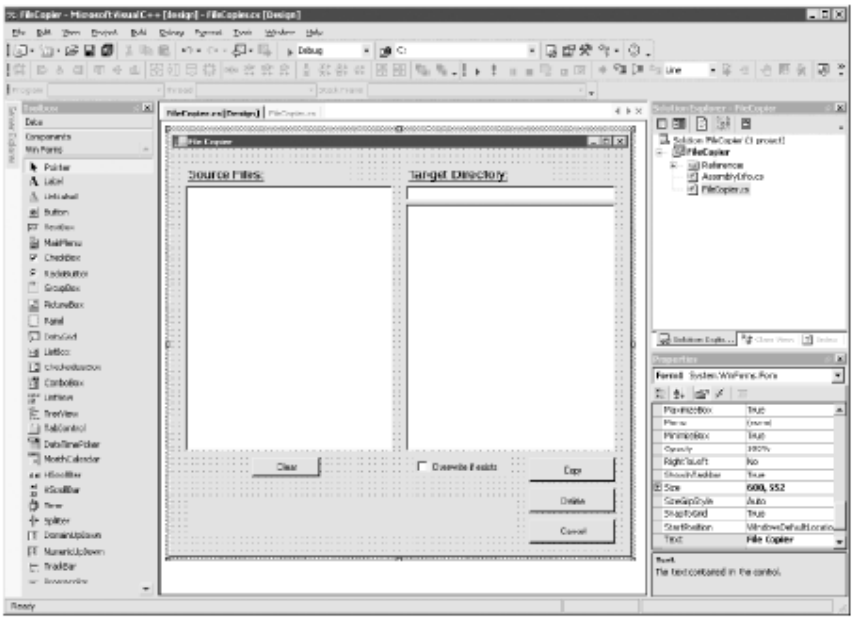
- Labels: Các tập tin nguồn (Source Files) and Thư múc đích (Target Directory).
- Buttons: Bỏ các dấu chọn trên cây bên trái (Clear), Copy, Delete, and Cancel.
- Checkbox : ghi đè lên nếu đã có sẵn ("Overwrite if exists")
- Checkbox : hiển thị đường dẫn của mục được trọn ở cây bên phải.
- Hai cây (TreeView) chứa tập tin.

Khi người dùng nhấn vào Button '`Copy`' thì tất các tập tin được chọn ở cây bên trái sẽ được chép qua cây bên phải, cũng như khi nhấn vào Button 'Delete' thì sẽ xóa các tập tin được chọn.

13.1.2.1 Tạo giao diện cho ứng dụng

Đầu tiên ta tạo một dự án Windows Form mới có tên FileCopier. IDE sẽ hiển thị màn hình thiết kế (`Designer`) lên, ta sẽ thực hiện kéo thả các Label, Button, Checkbox và TreeView cho đến khi thích hợp như hình dưới đây :

Hình 13-8 Tạo giao diện ứng dụng bằng cách kéo thả dùng Designer



Sau khi tạo giao diện xong, ta đặt thuộc tính `CheckBoxes` cho cây bên trái có tên `tvwSource` thành `true`, còn cây bên phải có tên `tvwTargetDir` thành `false`, để thực hiện ta đơn giản chỉ chọn và sửa đổi trên cửa sổ thuộc tính của từng đối tượng. Khi ta nhấn đúp lên bất kỳ Điều khiển nào thì tự động Visual Studio .NET sẽ phát sinh ra mã tương ứng để bắt sự kiện của Điều khiển đó và đặt con trỏ (Cursor) vào ngay tại hàm đó, ta nhấn đúp vào Button "`Cancel`" và bổ sung mã như sau :

```
protected void btnCancel_Click( object sender, System.EventArgs e)
{
  Application.Exit( );
}
```

13.1.2.2 Quản lý điều khiển TreeView

Trong ứng dụng này, hai điều khiển `TreeView` hoạt động tương tự nhau, ngoại trừ điều khiển cây bên trái `tvwTargetDir` có thuộc tính `CheckBoxes` là `true` và liệt kê cả tập tin lẫn thư mục, còn cây bên phải là `false` và chỉ liệt ke thư mục. Mặc nhiên thì điều khiển cây cho phép chọn nhiều mục một lúc, nhưng ta sẽ chỉnh lại sao cho chỉ cây bên trái `tvwSource` mới được chọn nhiều mục một lúc,bên phải thì không.

Ta sẽ tạo ra một hàm đẩy dữ liệu vào cây :

```
private void FillDirectoryTree(TreeView tvw, bool isSource)
```

Có 2 tham số :

`TreeView tvw:` điều khiển cây cần đẩy dữ liệu vào
`Bool isSource:` cờ xác định là dữ liệu đẩy cho cây. Nếu isSource là true thì cây sẽ liệt kê cả tập tin và thư mục, false thì chỉ có tập tin.

Hàm này sẽ được dùng chung cho cả hai điều khiển cây :

```
FillDirectoryTree(tvwSource, true);
FillDirectoryTree(tvwTargetDir, false);
```

Đối tượng TreeNode

Điều khiển `TreeView` có một thuộc tính `Nodes`. thuộc tính này nhận vào một đối tượng `TreeNodeCollection` , đối tượng này thực chất là một mảng chứa các đối tượng `TreeNode`, mỗi một `TreeNode` là một nút trên cây.

Trước tiên ta cần khởi tạo cây về rỗng :

```
tvw.Nodes.Clear( );
```

Sau đó ta gọi hàm tĩnh GetLogicalDrives() của đối tượng Enviroment để lấy về tất cả các ổ đĩa logic hiện đang có trên máy. Đối tượng Enviroment cung cấp các thông tin như : tên máy tính, phiên bản hệ điều hành, hệ thống thư mục … trên máy tính hiện hành.

```
string[] strDrives = Environment.GetLogicalDrives( );
```

strDrives sẽ chứa tên các ổ đĩa logic hiện có trên máy.

Sau đó ta sẽ duyệt qua từng ổ đĩa bằng cách dùng lệnh foreach. Với mỗi ổ đĩa logic, ta gọi hàm GetDirectories() của đối tượng DirectoryInfo. Hàm này sẽ trả về danh sách các đối tượng DirectoryInfo, chứa tất cả các tập tin và thư mục trên ổ đĩa logic đó. Những tại đây ta không quan tâm đến kết quả mà nó trả về, mục đích ta gọi hàm này chủ yếu là để kiểm tra xem các ổ đĩa có hợp lệ hay không, nếu có bất kỳ một lỗi nào trên ổ đĩa thì hàm GetDirectories() sẽ quăng ra một ngoại lệ. Ta sẽ dùng khối bắt lỗi try…catch để bắt lỗi này.

```
foreach (string rootDirectoryName in strDrives)
{
  DirectoryInfo dir = new DirectoryInfo(rootDirectoryName);
  dir.GetDirectories( );
  ...
}
```

Khi ổ đĩa hợp lệ, ta sẽ tạo ra một TreeNode ứng với **rootDirectoryName** ổ đĩa đó, chẳng hạn như : “C:\”, “D:\” …Rồi thêm TreeNode này vào điều khiển cây dùng hàm Add() thông qua thuộc tính Nodes của cây.

```
TreeNode ndRoot = new TreeNode(rootDirectoryName);
tvw.Nodes.Add(ndRoot);
```

Tiếp theo ta tiến hành duyệt trên mọi thư mục con của đối tượng TreeNode gốc trên, để làm điều này ta gọi hàm GetSubDirectoriesNodes( ), hàm này cần nhận vào các đối số : TreeNode gốc, tên của nó và cờ xác định là có đẩy cả tập tin vào cây hay không.

```
if (isSource)
{
  GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
}
else
{
  GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
}
```

Duyệt đệ qui trên các thư mục con

Hàm GetSubDirectoryNodes() bắt đầu bằng việc gọi hàm GetDirectories() để nhận về một danh sách các đối tượng DirectoryInfo :

```
private void GetSubDirectoryNodes(
TreeNode parentNode, string fullName, bool getFileNames)
{
  DirectoryInfo dir = new DirectoryInfo(fullName);
  DirectoryInfo[] dirSubs = dir.GetDirectories( );
```

Ở đây ta thấy node truyền vào có tên là parentNode (nút cha), nghĩa là những nút sau này sẽ được xem là nút con của nó. Bạn sẽ rõ hơn khi tìm hiểu hết hàm này.

Ta tiến hành duyệt qua danh sách các thư mục con dirSubs, bỏ qua các mục có trạng thái là ẩn (Hidden).

```
foreach (Directory dirSub in dirSubs)
{
  if ( (dirSub.Attributes & FileSystemAttributes.Hidden) != 0 )
  {
    continue;
  }
```

FileSystemAttributes là biến có kiểu enum, nó chứa một số giá trị như : Archive, Compressed, Encrypted, Hidden, Normal, ReadOnly …Nếu như mục hiện hành không ở trạng thái ẩn, ta sẽ tạo ra một TreeNode mới với tham số là tên của nó. Sau đó Thêm nó vào nút cha parentNode :

```
TreeNode subNode = new TreeNode(dirSub.Name);
parentNode.Nodes.Add(subNode);
```

Ta sẽ gọi lại đệ qui hàm GetDirectoriesNodes() để liệt kê hết mọi mục con trên thư nút hiện hành, với ba thông số : nút được chuyển vào như nút cha, tên đường dẫn đầy đủ của mục hiện hành và cờ trạng thái.

```
GetSubDirectoryNodes(subNode,dirSub.FullName,getFileNames);
```

> *Chú ý : Thuộc tính dirSubs.FullName sẽ trả về đường dẫn đầy đủ của mục hiện hành ("C:\dir1\dir2\file1"), còn thuộc tính dirSubs.Name chỉ trả về tên của mục hiện hành ("file1"). Khi ta tạo ra một nút con subNode, ta chỉ truyền cho nó tên của mục hiện hành, vì ta chỉ muốn hiển thị thị tên của nó trên cây. Còn khi ta gọi đệ qui hàm GetSubDirectoryNodes() thì ta cần truyền cho nó tên đường dẫn đầy đủ của mục hiện hành, để có thể liệt kê toàn bộ mục con cùa thực mục đang xét.*

Đến đây chắc bạn đã hiểu được sự phân cấp của cấu trúc cây và tại sao hàm GetSubDirectoryNodes() cần truyền có đối số FullName.

Lấy về các tập tin trong thư mục

Nếu biến cờ getFileNames là True thì ta sẽ tiến hành lấy về tất cả các tập tin thuộc thư mục. Để thực hiện ta gọi hàm GetFiles() của đối tượng DirectoryInfo, hàm này sẽ trả về danh sách các đối tượng FileInfo. Ta sẽ duyệt qua danh sách này để lấy ra

tên của từng tập tin một, sau đó tạo ra một nút TreeNode với tên này, nút này sẽ được thêm vào nút cha parentNode hiện hành.

13.1.2.3 Quản lý sự kiện trên điều khiển cây

Trong ứng dụng này, chúng ta sẽ phải quản lý một số sự kiện. Đầu tiên là sự kiện người dùng nhấn lên ô CheckBox để chọn các tập tin hay thư mục ở cây bên phải hay nhấn các nút ở cây bên phải. Tiếp theo là các sự kiện nhấn vào Button 'Cancel', 'Copy','Delete' hay 'Clear'.

Ta sẽ khảo sát sự kiện trên điều khiển cây trước.

Sự kiện chọn một nút trên điều khiển cây bên trái

Khi người dùng muốn chọn một tập tin hay thư mục để chép hay xóa. Ứng với mỗi lần chọn sẽ phát sinh ra một số sự kiện tương ứng. Ta sẽ bắt sự kiện AfterCheck của điều khiển cây. Ta gõ vào các đoạn mã sau :

```
tvwSource.AfterCheck +=
     new TreeViewEventHandler( this.tvwSource_AfterCheck );
```

Ta viết lệnh thực thi cho hàm bắt sự kiện AfterCheck có tên là tvwSource_AfterCheck, hàm này có hai tham số : đầu tiên là biến Sender chứa thông tin về đối tượng phát sinh ra sự kiện, thứ hai là đối tượng TreeViewEventArgs chứa thông tin về sự kiện phát ra. Ta sẽ đánh dấu là chọn cho thư mục được chọn và tất cả các tập tin hay thư mục con của thư mục đó thông qua hàm SetCheck() :

```
protected void tvwSource_AfterCheck (
object sender, System.Windows.Forms.TreeViewEventArgs e)
{
   SetCheck(e.node,e.node.Checked);
}
```

Hàm SetCheck() sẽ tiến hành thực hiện đệ qui trên nút hiện hành, hàm gồm hai tham số : nút cần đánh dấu và cờ xác định là đánh dấu hay bỏ đánh dấu chọn, nếu thuộc tính Count bằng không (nghĩa là nút này là nút lá) thì ta sẽ đánh dấu chọn cho nút đó. Nếu không ta gọi đệ qui lại hàm SetCheck() :

```
private void SetCheck(TreeNode node, bool check)
{
   node.Checked = check;
   foreach (TreeNode n in node.Nodes)
   {
      if (node.Nodes.Count == 0)
      {
         node.Checked = check;
      }
      else
      {
         SetCheck(n,check);
      }
   }
}
```

Sự kiện chọn một nút trên điều khiển cây bên phải

Khi người dùng chọn một nút ở cây bên phải, ta sẽ phải cho hiện đường dẫn đầy đủ của nút đó lên TextBox ở góc phíc trên bên phải. Ta sẽ bắt sự kiện AfterSelect của cây. Sự kiện này sẽ được gọi sau khi người dùng nhấn một nút nào đó trên cây, hàm bắt sự kiện này như sau :

```
protected void tvwTargetDir_AfterSelect( object sender,
                 System.Windows.Forms.TreeViewEventArgs e)
{
  string theFullPath = GetParentString(e.node);
```

Sau khi ta có được đường dẫn đầy đủ của nút chọn, ta sẽ bỏ đi dấu \\ (Backslash) nếu có. Rồi cho hiển thị lên hộp thoại TextBox.

```
  if (theFullPath.EndsWith("\\"))
  {
    theFullPath =theFullPath.Substring(0,theFullPath.Length-1);
  }
  txtTargetDir.Text = theFullPath;
}
```

Hàm GetParentString() trả về đường dẫn đầy đủ của nút được truyền vào làm thông số. Hàm này cũng tiến hành lặp đệ qui trên nút truyền vào nếu nút này không là nút lá và thêm dấu \\ vào nó. Quá lặp sẽ kết thúc nếu nút hiện hành là không là nút cha.

```
private string GetParentString(TreeNode node)
{
  if(node.Parent == null)
  {
    return node.Text;
  }
  else
  {
    return GetParentString(node.Parent) + node.Text +
              (node.Nodes.Count == 0 ? "" : "\\");
  }
}
```

Quản lý sự kiện nhấn nút bỏ chọn (Clear)

Ta tiến hành bổ sung mã lệnh sau cho hàm bắt sự kiện nhấn vào nút ‘Clear’ :

```
protected void btnClear_Click( object sender, System.EventArgs e)
{
  foreach ( TreeNode node in tvwSource.Nodes )
  {
    SetCheck(node, false);
  }
}
```

Hàm này chỉ đơn giản là duyệt qua tất cả các nút thuộc cây bên trái, sau đó gọi lại hàm SetCheck() với biến cờ là false, nghĩa là bỏ chọn tất cả các nút hiện đang được chọn trên điều khiển cây.

Quản lý sự kiện nhấn nút chép tập tin (Copy)

Cái ta cần để hoàn chỉnh thao tác này là danh sách các đối tượng FileInfo. Để có thể quản lý linh hoạt trên danh sách này ta sẽ dùng đối tượng ArrayList, nó cho phép ta

thực hiện hầu hết mọi thao tác trên danh sách một các dễ dàng. Để lất về danh sách các đối tượng FileInfo, ta sẽ gọi hàm GetFileList() của ta :

```
protected void btnCopy_Click( object sender, System.EventArgs e)
{
  ArrayList fileList = GetFileList( );
```

Lấy về danh sách các tập tin

Đầu tiên ta sẽ khởi tạo một đối tượng ArrayList để lưu trữ danh sách tên các tập tin được chọn, có tên là fileNames :

```
private ArrayList GetFileList( )
{
  ArrayList fileNames = new ArrayList( );
```

Ta lấy về danh sách tên các tập tin được chọn bằng cách duyệt toàn bộ các nút trong điều khiển cây bên phải :

```
foreach (TreeNode theNode in tvwSource.Nodes)
{
  GetCheckedFiles(theNode, fileNames);
}
```

Hàm GetCheckedFiles() thêm danh sách tên các tập tin được đánh dấu của nút hiện hành theNode vào đối tượng fileNames. Nếu nút truyền vào là nút lá và được đánh dấu chọn, ta sẽ lấy đường dẫn đầy đủ của nút và thêm vào đối tượng fileNames:

```
private void GetCheckedFiles(TreeNode node, ArrayList fileNames)
{
  if (node.Nodes.Count == 0)
  {
    if (node.Checked)
  {
    string fullPath = GetParentString(node);
    fileNames.Add(fullPath);
  }
}
```

Nếu không là nút lá, ta sẽ lập đệ qui để tìm nút lá :

```
  else
  {
    foreach (TreeNode n in node.Nodes)
      GetCheckedFiles(n,fileNames);
  }
}
```

Sau khi thực hiện hết hàm này (nghĩa là duyệt hết cây tvwSource), đối tượng fileNames sẽ chứa toàn bộ các tập tin được đánh dấu chọn của cây.

Quay trở lại khảo sát tiếp tục hàm GetFileList(), ta tạo thêm một đối tượng ArrayList nữa, tên fileList. Mảng này sẽ chứa danh sách các đối tượng FileInfo ứng với các tên tập tin tìm được trong mảng fileNames. Thuộc tính Exists của đối tượng FileInfo dùng để kiểm tra là tập tin hay thư mục. Thuộc tính Exists là True thì đối tượng FileInfo đó là tập tin và ta sẽ thêm vào mảng fileList, ngược lại là thư mục thì không thêm .

```
foreach (string fileName in fileNames)
```

```
{
  FileInfo file = new File(fileName);
  if (file.Exists)
    fileList.Add(file);
}
```

13.1.2.4 Quản lý sự kiện nhấn chọn nút xóa (Delete)

Trước tiên ta cần đảm bảo rằng người dùng chắc chắn muốn xóa bằng cách cho hiện lên một hộp thoại xác nhận xóa. Để hiển thị hộp thoại ta dùng hàm tĩnh Show() của đối tượng MessageBox.

```
protected void btnDelete_Click( object sender, System.EventArgs e)
{
  System.Windows.Forms.DialogResult result =
  MessageBox.Show( "Are you quite sure?",// Thông điệp
            "Delete Files",               // Tiêu đề cho hộp thoại
            MessageBox.Buttons.OKCancel,// nút nhấn
            MessageBox.Icon.Exclamation,// biểu tượng hộp thoại
            MessageBox.DefaultButton.Button2); // nút mặc định
}
```

Khi người dùng nhấn nút **OK** hay **Cancel,** ta sẽ nhận được giá trị trả về từ đối tượng DialogResult thuộc namespace Forms và tiến hành xử lý tương ứng :

```
if (result == System.Windows.Forms.DialogResult.OK)
{
```

Nếu người dùng chọn nút **OK** thì ta sẽ lấy về danh sách tên các tập tin fileNames, sau đó duyệt qua từng tên và xóa chúng đi :

```
ArrayList fileNames = GetFileList( );
foreach (FileInfo file in fileNames)
{
  try
  {
    lblStatus.Text = "Deleting " +
    txtTargetDir.Text + "\\" +
    file.Name + "...";
    Application.DoEvents( );
    file.Delete( );
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
lblStatus.Text = "Done.";
Application.DoEvents( );
```

Sau đây là mã của toàn bộ ứng dụng :

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
```

```
/// <remarks>
/// chép tập tin - ứng dụng minh họa cho Windows Form
/// </remarks>
namespace FileCopier
{
   /// <summary>
   /// Form minh họa cho ứng dụng Windows Form
   /// </summary>
   public class Form1 : System.Windows.Forms.Form
   {
      /// <summary>
      /// lớp bên trong của lớp Form1, so sánh 2 tập tin
      /// </summary>
      public class FileComparer : IComparer
      {
         public int Compare (object f1, object f2)
         {
            FileInfo file1 = (FileInfo) f1;
            FileInfo file2 = (FileInfo) f2;
            if (file1.Length > file2.Length)
            {
               return -1;
            }
            if (file1.Length < file2.Length)
            {
               return 1;
            }
            return 0;
         }
      }

      public Form1( )
      {
         InitializeComponent( );
         // đẩy dữ liệu vào cây bên trái và bên phải
         FillDirectoryTree(tvwSource, true);
         FillDirectoryTree(tvwTargetDir, false);
      }

      /// <summary>
      /// phương thức này dùng để đẩy dữ liệu vào cây
      /// </summary>
      private void FillDirectoryTree(TreeView tvw, bool isSource)
      {
         // trước khi đẩy dữ liệu vào cây, ta phải xóa bỏ các nút
         // hiện đang tồn tại trên cây.
         tvw.Nodes.Clear( );

         // lấy về danh sách các ổ đĩa logic trên máy tính
         // sau đó đẩy chúng vào làm nút gốc của cây
         string[] strDrives = Environment.GetLogicalDrives( );

         // Duyệt qua các ổ đĩa, dùng khối try/catch để bắt bất
         // kỳ lỗi nào xảy ra trên đĩa, nếu đĩa hợp lệ thì ta
         // thêm vào làm nút gốc cho cây.
         // đĩa không hợp lệ sẽ không thêm vào cây : đĩa mềm hay
         // CD trống ...
         foreach (string rootDirectoryName in strDrives)
```

```
    {
      if (rootDirectoryName != @"C:\")
        continue;
      try
      {
        // nếu đĩa không hợp lệ ta sẽ quăng ra một lỗi.
        DirectoryInfo dir =
        new DirectoryInfo(rootDirectoryName);
        dir.GetDirectories( );
        TreeNode ndRoot = new TreeNode(rootDirectoryName);

        // thêm nút gốc vào cây
        tvw.Nodes.Add(ndRoot);

        // thêm các nút con vào cây, nếu là cây bên trái
        //thì thêm cả tập tin vào cây
        if (isSource)
          GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
        else
          GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
      }
      catch (Exception e)
      {
        // thông báo đĩa có lỗi
        MessageBox.Show(e.Message);
      }
    }
} // kết thúc thao tác đẩy dữ liệu vào cây

/// <summary>
/// lấy về tất cả các thư mục con của nút cha truyền vào,
/// thêm các thư mục con tìm được vào cây
/// hàm này có 3 đối số : nút cha, tên đầy đủ của nút cha,
/// và biến cờ getFileNames xác định có lấy tập tin không
/// </summary>
private void GetSubDirectoryNodes( TreeNode parentNode,
                        string fullName, bool getFileNames)
{
   DirectoryInfo dir = new DirectoryInfo(fullName);
   DirectoryInfo[] dirSubs = dir.GetDirectories( );

   // ứng với mỗi mục con ta thêm vào cây nếu nó không ở
   //trạng thái ẩn.
   foreach (DirectoryInfo dirSub in dirSubs)
   {
      // bỏ qua các thư mục ẩn
      if ( (dirSub.Attributes & FileAttributes.Hidden) != 0 )
      {
        continue;
      }

      /// <summary>
        /// ta chỉ cần tên nút để thêm vào cây, còn ta phải
        /// truyền vào tên đường dẫn đầy đủ của nút trên cây
        /// cho hàm lặp đệ qui GetSubDirectoryNodes()để nó có
        /// thể tìm được các nút con cửa nút đó
      /// </summary>
      TreeNode subNode = new TreeNode(dirSub.Name);
```

```
            parentNode.Nodes.Add(subNode);

            // lặp đệ qui hàm GetSubDirectoryNodes().
            GetSubDirectoryNodes(
            subNode,dirSub.FullName,getFileNames);
         }

         if (getFileNames)
         {
            // lấy mọi tập tin thuộc nút
            FileInfo[] files = dir.GetFiles( );

            // thêm các tập tin và nút con
            foreach (FileInfo file in files)
            {
               TreeNode fileNode = new TreeNode(file.Name);
               parentNode.Nodes.Add(fileNode);
            }
         }
      }
      /// <summary>
      /// điểm vào chính của ứng dụng.
      /// </summary>
      [STAThread]
      static void Main( )
      {
         Application.Run(new Form1( ));
      }
      /// <summary>
      /// tạo ra một danh sách có thứ tự các tập tin được chọn ,
      /// chép chúng sang cây bên phải
      /// </summary>
      private void btnCopy_Click(object sender,
         System.EventArgs e)
      {
         // lấy về danh sách tập tin
         ArrayList fileList = GetFileList( );

         // tiến hành chép tấp tin
         foreach (FileInfo file in fileList)
         {
            try
            {
               lblStatus.Text = "Copying " + txtTargetDir.Text +
                  "\\" + file.Name + "...";
               Application.DoEvents( );
               file.CopyTo(txtTargetDir.Text + "\\" +
                  file.Name,chkOverwrite.Checked);
            }
            catch // (ta không làm gì ở đây cả)
            {
            }
         }
         lblStatus.Text = "Done.";
         Application.DoEvents( );
      }

      /// <summary>
```

```
/// bắt sự kiện kết thúc ứng dụng
/// </summary>
private void btnCancel_Click(object sender, System.EventArgs e)
{
   Application.Exit( );
}

/// <summary>
/// bắt sự kiện xóa bỏ các nút được chọn trên cây bên trái
/// </summary>
private void btnClear_Click( object sender, System.EventArgs e)
{
   // lấy về nút gốc trên cây bên trái và
   // tiến hành lặp đệ qui
   foreach (TreeNode node in tvwSource.Nodes)
   {
      SetCheck(node, false);
   }
}

/// <summary>
/// đảm bảo người dùng muốn xóa nút đó
/// </summary>
private void btnDelete_Click(object sender, System.EventArgs e)
{
   // xác nhận xóa
   System.Windows.Forms.DialogResult result = MessageBox.Show(
          "Are you quite sure?", // thông điệp
          "Delete Files", // tiêu đề
          MessageBox.Buttons.OKCancel, // nút nhấn
          MessageBox.Icon.Exclamation, // biểu tượng
          MessageBoxDefaultButton.Button2); // nút mặc định

   // nếu đồng ý xóa
   if (result == System.Windows.Forms.DialogResult.OK)
   {
      // duyệt danh sách các tập tin và xoá các tập tin
      / được chọn
      ArrayList fileNames = GetFileList( );
      foreach (FileInfo file in fileNames)
      {
         try
         {
            // cập nhật nhãn
            lblStatus.Text = "Deleting " +
            txtTargetDir.Text + "\\" + file.Name + "...";
            Application.DoEvents( );

            file.Delete( );
         }
         catch (Exception ex)
         {
            // hộp thoại thông báo
            MessageBox.Show(ex.Message);
         }
      }
      lblStatus.Text = "Done.";
      Application.DoEvents( );
```

```
    }
}

/// <summary>
/// lấy đường dẫn đầy đủ của nút được chọn và gán vào
/// điều khiển TextBox txtTargetDir
/// </summary>
private void tvwTargetDir_AfterSelect( object sender,
        System.Windows.Forms.TreeViewEventArgs e)
{
   // lấy đường dẫn đầy đủ của nút
   string theFullPath = GetParentString(e.Node);

   // nếu nó không là nút lá, ta sẽ xóa 2 ký tự cuối cùng
   // đi, vì đây là dấu //
   if (theFullPath.EndsWith("\\"))
   {
      theFullPath =
        theFullPath.Substring(0,theFullPath.Length-1);
   }

   // gán đường dẫn cho điều khiển TextBox
   txtTargetDir.Text = theFullPath;
}

/// <summary>
/// đánh dấu chọn nút hiện hành và các nút con của nó
/// </summary>
private void tvwSource_AfterCheck(object sender,
   System.Windows.Forms.TreeViewEventArgs e)
{
   SetCheck(e.Node,e.Node.Checked);
}

/// <summary>
/// lập đệ qui việc đánh dấu chọn hay loại bỏ dấu chọn trên
/// nút truền vào dựa vào cờ check
/// </summary>
private void SetCheck(TreeNode node, bool check)
{
   // set this node's check mark
   node.Checked = check;
   // tìm tất cả các nút con của nút
   foreach (TreeNode n in node.Nodes)
   {
      // nếu là nút là thì ta đánh dấu chọn hoặc không chọn
      if (node.Nodes.Count == 0)
        node.Checked = check;
      // nếu không là lá thì ta lặp đệ qui
      else
        SetCheck(n,check);
   }
}

// lấy về tất cả các tập tin đã được đánh dấu chọn thuộc
// nút
private void GetCheckedFiles(TreeNode node,ArrayList fileNames)
{
```

```
    // nếu là nút lá
    if (node.Nodes.Count == 0)
    {
      // nếu nút là đánh dấu chọn
      if (node.Checked)
      {
        // lấy đường dẫn đầy đủ của nút và thêm vào cây
        string fullPath = GetParentString(node);
        fileNames.Add(fullPath);
      }
    }
    else // không là nút lá
    {
      // thực hiện trên tất cả nút con
      foreach (TreeNode n in node.Nodes)
      {
        GetCheckedFiles(n,fileNames);
      }
    }
  }

  /// <summary>
  /// lấy về đường dẫn đầy đủ của nút truyền vào
  /// </summary>
  private string GetParentString(TreeNode node)
  {
    // nếu là nút gốc thì trả về tên nút ( c:\ )
    if(node.Parent == null)
      return node.Text;
    else
      // nếu là nút cha thì thêm dấu // vào chuỗi trả về
      // nếu nút lá ta không thêm gì cả
      return GetParentString(node.Parent) + node.Text +
        (node.Nodes.Count == 0 ? "" : "\\");
  }

  /// <summary>
  /// trả về danh sách các tập tin được chọn theo thứ tự
  /// </summary>
  private ArrayList GetFileList( )
  {
    // danh sách tên tập tin đầy đủ không được sắp
    ArrayList fileNames = new ArrayList( );

    // duyệt từng nút của cây và lấy về danh sách các tập
    // tin được chọn
    foreach (TreeNode theNode in tvwSource.Nodes)
    {
      GetCheckedFiles(theNode, fileNames);
    }

    // danh sách các đối tượng FileInfo
    ArrayList fileList = new ArrayList( );

    // fileNames là tập tin thì ta thêm vào fileList
    foreach (string fileName in fileNames)
    {
      // tạo ra FileInfo tương ứng với fileName
```

```
            FileInfo file = new FileInfo(fileName);
            if (file.Exists)
               fileList.Add(file);
         }

         // tạo ra một thể hiện IComparer
         IComparer comparer = (IComparer) new FileComparer( );

         // sắp xếp danh sách tập tin
         fileList.Sort(comparer);
         return fileList;
      }
   }
}
```

13.1.3 Tạo sưu liệu XML bằng chú thích

Ngôn ngữ C# hỗ trợ kiểu chú thích mới, bằng ba dấu gạch chéo (///). Trình biên dịch C# dùng phần chú thích này để tạo thành sưu liệu XML.

Ta có thể tạo tập tin sưu liệu XML này bằng mã lệnh, ví dụ như để tạo sưu liệu cho ứng dụng FileCopier ở trên ta gõ các lệnh sau :

```
csc filecopier.cs /r:System.Windows.Forms.dll /r:mscorlib.dll
/r:system.dll /r:system.configuration.dll /r:system.data.dll
/r:system.diagnostics.dll /r:system.drawing.dll
/r:microsoft.win32.interop.dll
/doc:XMLDoc.XML
```

Ta cũng có thể tạo sưu liệu XML trực tiếp ngay trong Visual Studio .NET, bằng cách nhấn chuột phải lên biểu tượng của dự án và chọn 'Properties' để hiện lên hộp thoại thuộc tính của dự án (Property Pages), sau đó chọn mục Configuration Properties \ Build rồi gõ tên tập tin sưu liệu XML cần tạo ra vào dòng XML Document File. Khi biên dịch dự án, tập tin sưu liệu XML sẽ tự động được tạo ra trong thư mục chứa dự án. Dưới đây là một đoạn mã được trích ra từ tập tin sưu liệu XML được tạo ra từ ứng dụng FileCopier trên :

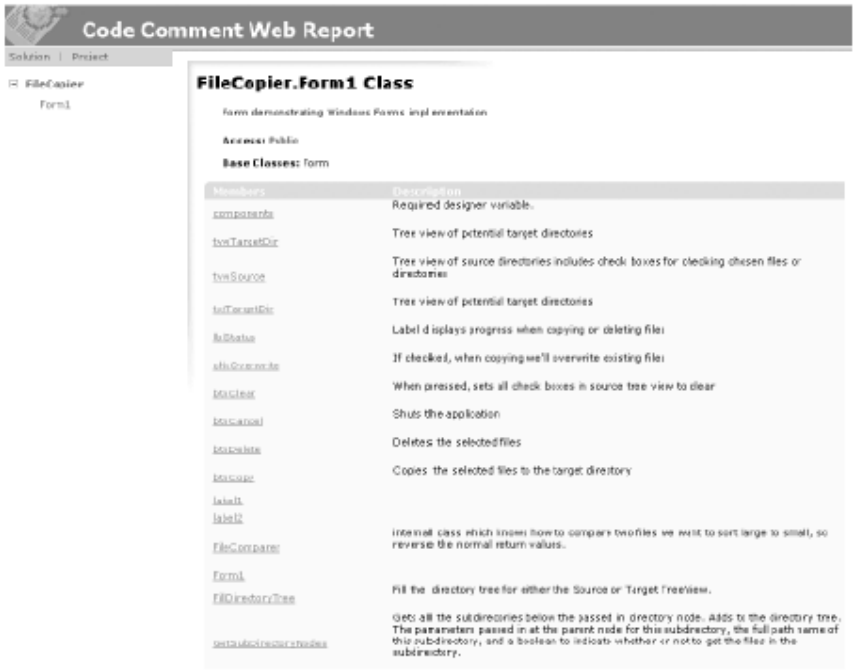
```
<?xml version="1.0"?>
<doc>
<assembly>
<name>FileCopier</name>
</assembly>
<members>
<member name="T:FileCopier.Form1">
<summary>
Form demonstrating Windows Forms implementation
</summary>
</member>
<member name="F:FileCopier.Form1.components">
<summary>
Required designer variable.
</summary>
</member>
<member name="F:FileCopier.Form1.tvwTargetDir">
<summary>
Tree view of potential target directories
```

```
</summary>
</member>
<member name="F:FileCopier.Form1.tvwSource">
<summary>
Tree view of source directories
includes check boxes for checking
chosen files or directories
</summary>
</member>
<member name="F:FileCopier.Form1.txtTargetDir">
```

Do đoạn mã trên được định dạng dưới kiểu dưới dạng XML, do đó không thuận tiện lắm khi quan sát. Ta có thể viết một tập tin theo định dạng XSLT để chuyển từ định dạng XML sang HTML.

Một cách đơn giản hơn để tạo sưu liệu XML thành các báo cáo HTML dễ đọc hơn là dùng chức năng Tool \ Build Command Web Page …, VS.NET sẽ tự động tạo ra một tập các tập tin sưu liệu HTML tương ứng với tập tin XML. Dưới đây là giao diện của màn hình sưu liệu ứng dụng FileCopier được tạo bởi VS.NET :

Hình 13-9 Sưu liệu dưới dạng Web được tạo bởi Visual Studio .NET



13.1.4 Triển khai ứng dụng

Khi ứng dụng đã thực thi hoàn chỉnh, vấn đề bây giờ là làm cách nào để có thể triển khai nó. Với các ứng dụng đơn giản, chỉ cần chép **assembly** của ứng dụng đó sang máy khác và chạy.

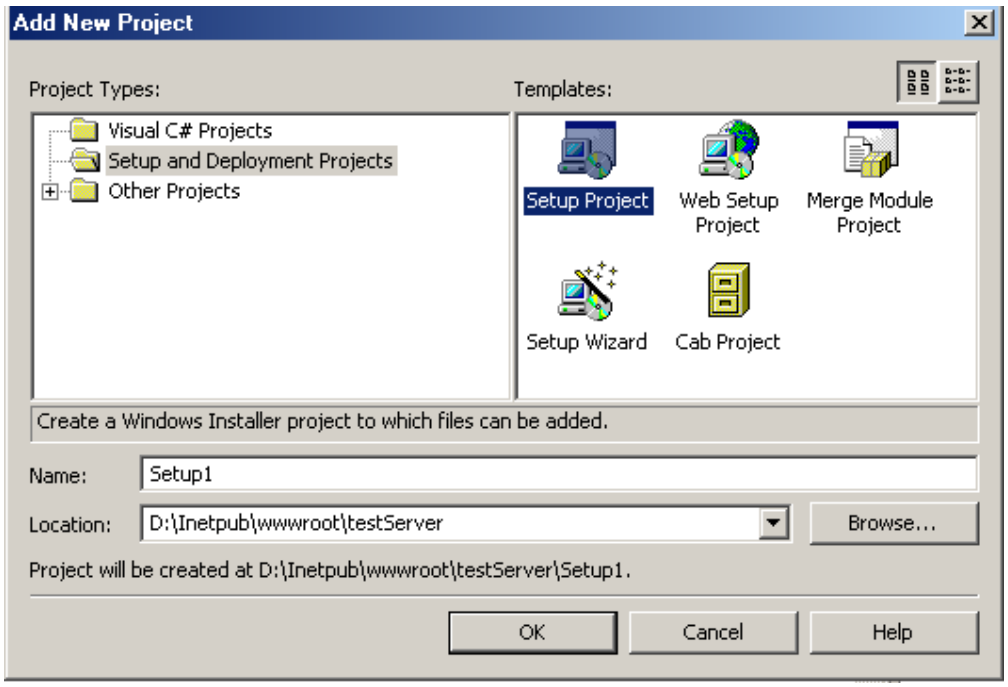
Ví dụ ta dịch ứng dụng FileCopier thành tập tin chạy FileCopier.exe, sau đó chép sang máy khác và chạy nó. Ứng dụng sẽ thực thi tốt.

13.1.4.1 Việc triển khai các dự án (Deployment Projects)

Đối với các ứng dụng thương mại lớn hơn, khách hàng muốn ứng dụng được cài đặt vào một thư mục cụ thể với biểu tượng đặc trưng của họ…, khi đó cách đơn giản trên chưa đủ. Visual Studio .NET đã cung cấp thêm một phần mở rộng khác để hỗ trợ việc cài đặt và triển khai (`Setup and Deployment Projects`) ứng dụng.

Giả sử ta đang ở trong một dự án nào đó, ta chọn File\Add Project \ New Project \ Setup and Deployment Projects. Ta sẽ thấy hộp thoại sau :

Hình 13-10 Hộp thoại tạo dự án mới.

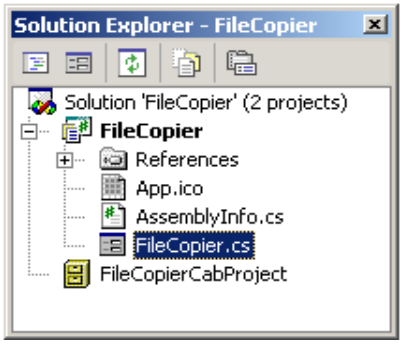


Ta có nhiều nhiều kiểu dự án triển khai khác nhau :

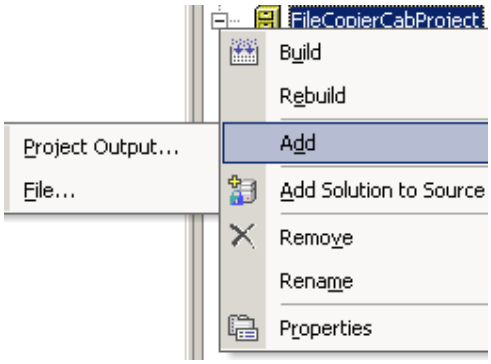
- **Setup Project** : Tạo ra tập tin cài đặt, tập tin này có thể tự cài đặt các tập tin và tài nguyên của ứng dụng.
- **Cab Project** :Giống như một tập tin ZIP, dự án loại này nén các tập tin thành một gói (Package) . Chọn lựa này có thể kết hợp với các loại khác.
- **Merge Module** : Nếu ứng dụng của ta có nhiều dự án cùng dùng chung một số tập tin, thì sự chọn lựa này giúp ta trộn chúng thành các module trung gian chung. Ta có thể tích hợp các module này vào các dự án khác.
- **Setup Wizard** : Giúp thực hiện một trong các loại dự án trên được dễ dàng.
- **Web Setup Project** : Giúp triển khai các dự án Web.

Để hiểu rõ, ta sẽ thử tạo một dự án triển khai kiểu Cab Project, thường thì khi dự án của ta có nhiều tập tin .Html, .Gif hay một số loại tài nguyên khác mà cần phải kèm theo với ứng dụng thì ta triển khai dự án theo kiểu này. Thêm dự án loại Cab Project vào dự án với tên là FileCopierCabProject.

Hình 13-11 Dự án được thêm vào ứng dụng.

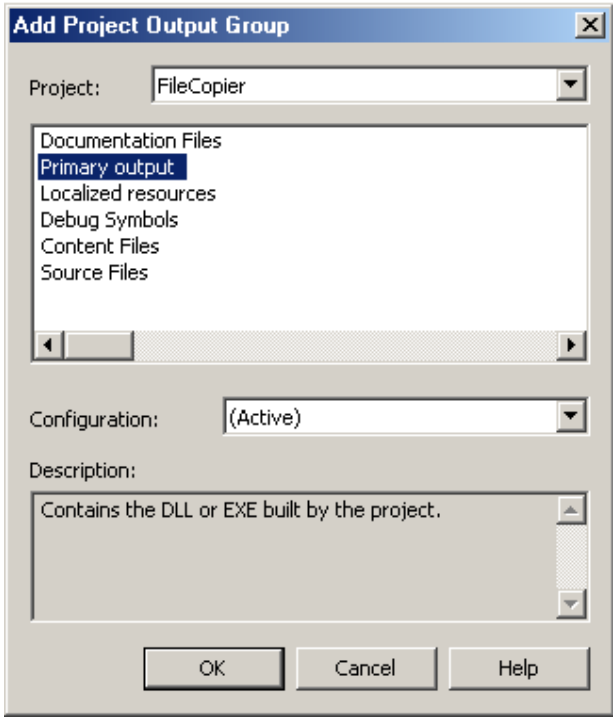


Hình 13-12 Hai kiểu thêm trong dự án loại CAB



Nhấn chuột phải trên dự án triển khai FileCopierCabProject. Có 2 dạng đóng gói tập tin CAB : Project Output… và File… . Ở đây ta chọn Add \ Project Output, hộp thoại chọn lựa kiểu kết xuất cho dự án (Add Project Output Group) xuất hiện :

Hình 13-13 Lựa chọn loại kết xuất để đóng gói.



Ở đây, ta sẽ chọn loại **Primary Output** để tạo tập tin FileCopier.exe cho ứng dụng **FileCopier** của ta. Khi chạy chương trình thì .NET sẽ tạo ra một gói có tên là **FileCopierCabProject.CAB.** Trong gói này có chứa 2 tập tin :

- FileCopier.exe : tập tin chạy của ứng dụng
- Osd8c0.osd : tập tin này mô tả gói .CAB theo dạng XML.

Mã mô tả XML tập tin .CAB

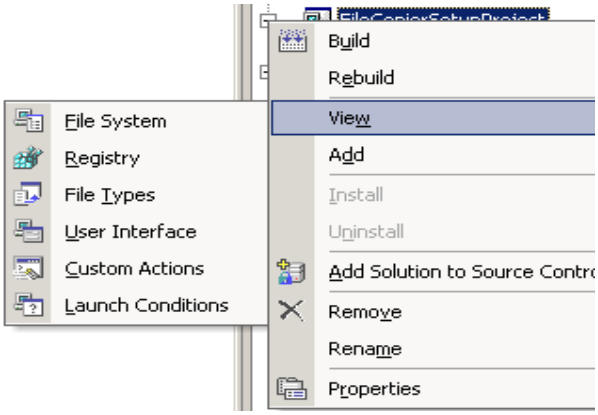
```
<?XML version="1.0" ENCODING='UTF-8'?>
<!DOCTYPE SOFTPKG SYSTEM
"http://www.microsoft.com/standards/osd/osd.dtd">
<?XML::namespace
href="http://www.microsoft.com/standards/osd/msicd.dtd"
        as="MSICD"?>
<SOFTPKG NAME="FileCopierCabProject" VERSION="1,0,0,0">
<TITLE> FileCopierCabProject </TITLE>
<MSICD::NATIVECODE>
        <CODE NAME="FileCopier">
          <IMPLEMENTATION>
           <CODEBASE FILENAME="FileCopier.exe">
           </CODEBASE>
     </IMPLEMENTATION>
        </CODE>
     </MSICD::NATIVECODE>
</SOFTPKG>
```

13.1.4.2 Việc cài đặt dự án (Setup Project)

Để tạo được chương trình tự động cài đặt cho ứng dụng, ta thêm dự án khác và chọn kiểu là **Setup Project,** dự án kiểu này khá linh động. Nó tạo thành tập tin có phần mở rộng là **MSI**, có thể tự cài đặt ứng dụng của ta lên máy tính khác.

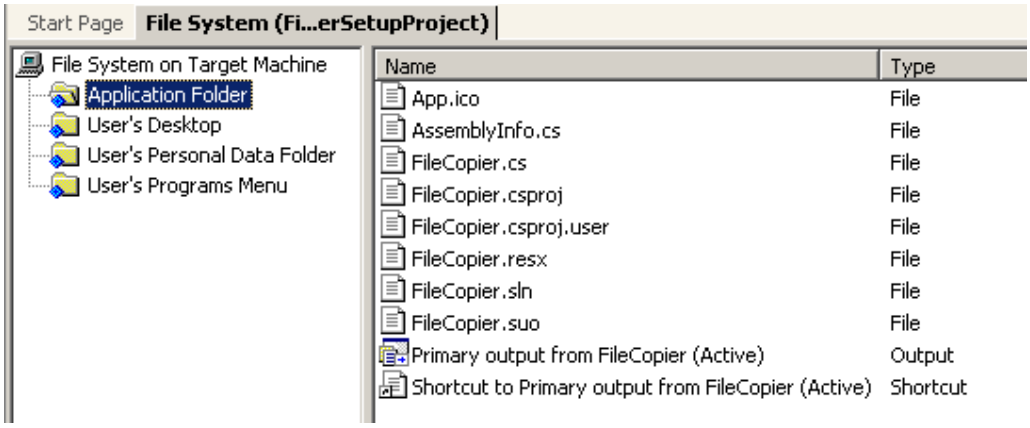
UI phục vụ chủ yếu cho việc cài đặt là cửa số **View Menu** :

Hình 13-14 Giao diện người dùng View Menu.



Ứng với mỗi chọn lựa trên View Menu, ta sẽ thấy các hiển thị tương ứng trong hai cửa sổ bên trái. Chẳng hạn như, khi ta chọn **View \ File System** thì ở bên trái sẽ hiện ra hai cửa sổ như như hình bên dưới, ta có thể thêm các tập tin hay tài nguyên liên quan đến ứng dụng theo ý muốn :

Hình 13-15 Cửa sổ File System của ứng dụng FileCopier



13.1.4.3 Triển khai trên các vị trí khác nhau

Mặc nhiên thì ứng dụng sẽ được cài đặt trên thư mục sau :

```
[ProgramFilesFolder]\[Manufacturer]\[Product Name.
  ProgramFilesFolder: thư mục Program Files trên máy người dùng
  Manufacturer: tên của nhà sản xuất
  Product Name: tên của ứng dụng
```

Ta có thể thay đổi các thông số này trong cửa sổ thuộc tính FileCopierSetupProject hoặc trong quá trình cài đặt.

Tạo biểu tượng của ứng dụng trên màn hình Desktop.

Để tạo biểu tượng cho ứng dụng, ta chỉ cần chọn **Application Folder\Primary output from FileCopier (Active)** ở cửa sổ bên trái, sau đó nhấn chuột phải và chọn **Create Shortcut to Primary output from FileCopier (Active).** Ta hiệu chỉnh đường dẫn của biểu tượng cho thích hợp.

Thêm các mục vào thư mục My Documents

Ta cũng có thể thêm các tài liệu cần thiết vào thư mục **My Documents** trên máy của người dùng khi cài đặt, bằng cách đặt chúng vào thư mục **User's Personal Data Folder**

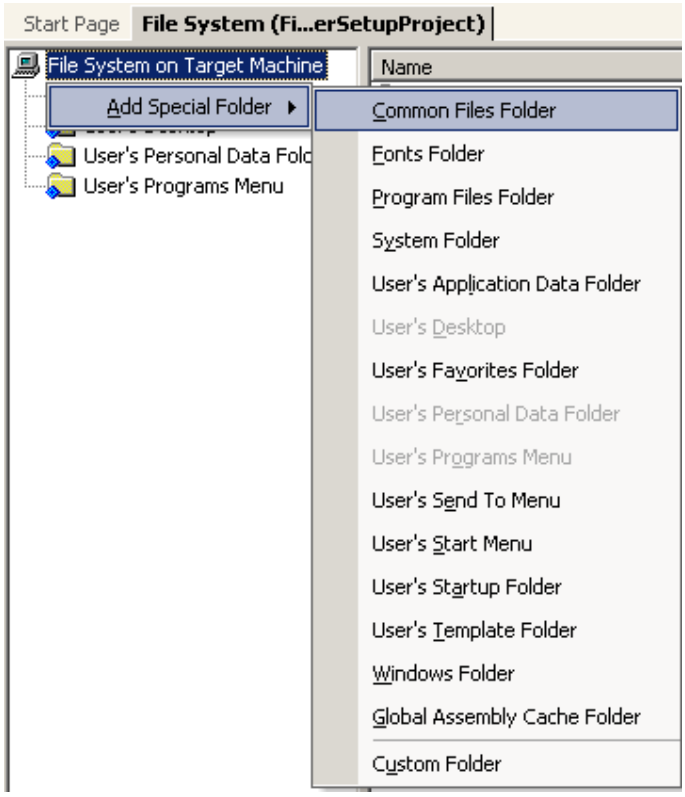
Tạo biểu trượng trong cửa sổ Start Menu

Để thêm các thành phần khác của ứng vào cửa sổ **Start / Programs**, ta thêm chúng vào thư mục **User's Program Menu** ở cửa sổ bên phải.

13.1.4.4 Thêm các chức năng cài đặt khác cho ứng dụng triển khai

Ngoài bốn mục được liệt kê trong hình trên, ta có thể bổ sung thêm các thư mục khác, như hình dưới đây :

Hình 13-16 Bổ sung các thư mục trong cửa sổ File System.



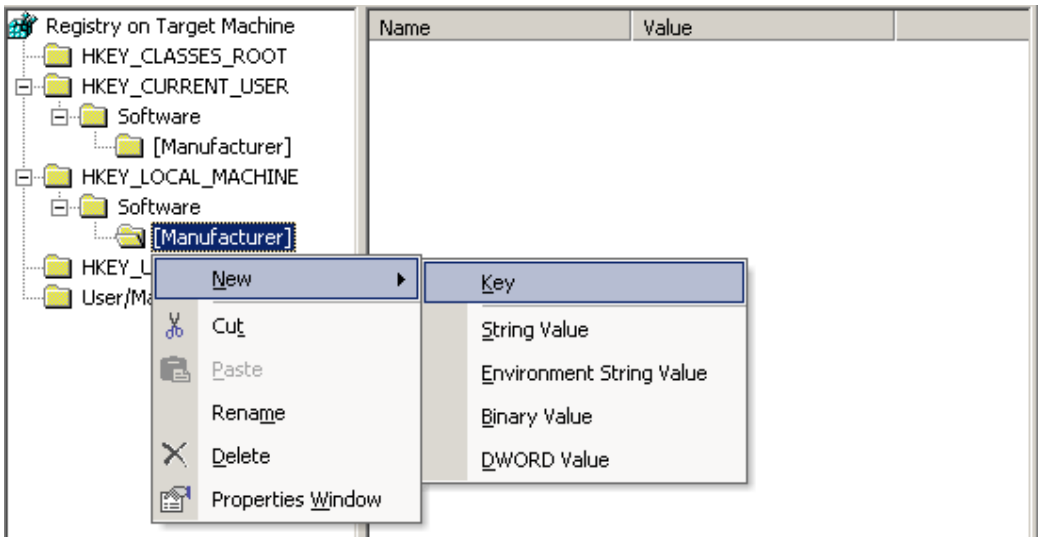
13.1.4.5 Các thành phần khác

Ta đã khảo sát qua cách cài đặt dùng **File System** trong **Menu View**, nay ta tìm hiểu thêm một số cách khác.

Tạo các thay đổi trong sổ đăng ký (Registry)

Cửa sổ đăng ký của View Menu cho phép làm cho trình cài đặt của chúng ta tạo ra các thay đổi trong sổ đăng ký chương trình trên các máy cài đặt ứng dụng. Nhấn chuột phải trên các thư mục được liệt trong hình dưới đây để hiệu chỉnh sổ đăng ký theo ý muốn :

Hình 13-17 Hiệu chỉnh sổ đăng ký.

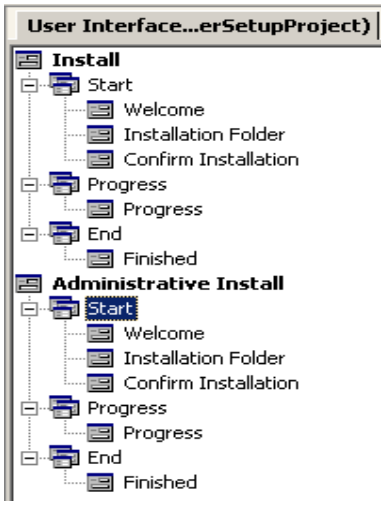


Chú ý: Phải rất thật cẩn thận khi cài đặt sổ đăng ký. Trong hầu hết các ứng dụng .NET đều không cần thiết phải liên quan đến sổ đăng ký, .NET quản lý ứng dụng không cần dùng sổ đăng ký (Registry).

Quản lý giao diện người dùng trong quá trình cài đặt

Để có thể kiểm soát các chữ hay giao diện đồ họa hiển thị trong suốt quá trình cài đặt ứng dụng, ta chọn mục User Interface trong View Menu. Hình dưới đây thể hiện luồng công việc trong quá trình cà đặt ứng dụng :

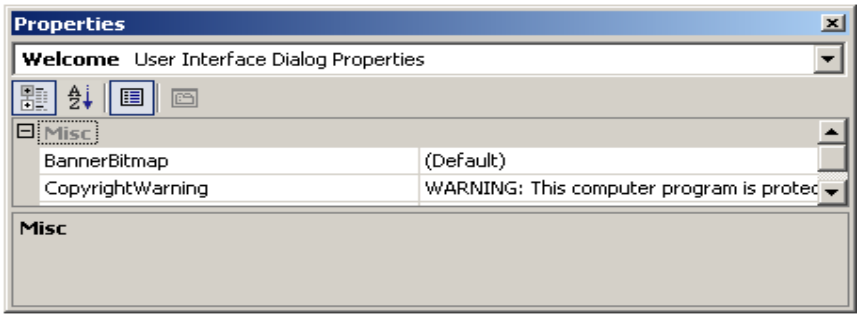
Hình 13-18 Luồng công việc trong quá trình cài đặt.



Khi ta nhấn chuột phải trên một bước nào đó trong tiến trình cài đặt và chọn **Properties** thì sẽ hiện lên một cửa sổ tương ứng với mục đó, nhờ hộp thoại thuộc tính này ta có thể hiệu chỉnh các chuỗi hay ảnh hiển thị thích hợp. Ta cũng có thể

thêm hay bớt đi một bước trong luồng công việc cài đặt. Ví dụ hộp thoại thuộc tính của mục **Welcome**.

Hình 13-19 Cửa sổ thuộc tính của mục Welcome trong quá trình cài đặt.



Hiệu chỉnh thêm cho quá trình cài đặt

Nếu quá trình cài đặt trong cửa sổ User Interface vẫn không đủ thỏa mãn nhu cầu cài đặt của ứng dụng, thì ta có thể tự hiệu chỉnh các thông số cho tiến trình cài đặt : điều kiện để chạy tiến trình cài đặt … Ta chọn mục Custom Option trong View Menu để thực hiện mục đích này.

Kết thúc việc tạo chương trình cài đặt

Sau khi hoàn tất mọi hiệu chỉnh, ta chỉ cần chạy ứng dụng và .NET sẽ tạo ra một tập tin cài đặt **MSD** để cài đặt ứng dụng của ta.

Chương 14 Truy cập dữ liệu với ADO.NET

Trong thực tế, có rất nhiều ứng dụng cần tương tác với cơ sở dữ liệu. .NET Framework cung cấp một tập các đối tượng cho phép truy cập vào cơ sở dữ liệu, tập các đối tượng này được gọi chung là ADO.NET.

ADO.NET tương tự với ADO, điểm khác biệt chính ở chỗ ADO.NET là một kiến trúc dữ liệu rời rạc, không kết nối (*Disconnected Data Architecture*). Với kiến trúc này, dữ liệu được nhận về từ cơ sở dữ liệu và được lưu trên vùng nhớ cache của máy người dùng. Người dùng có thể thao tác trên dữ liệu họ nhận về và chỉ kết nối đến cơ sở dữ liệu khi họ cần thay đổi các dòng dữ liệu hay yêu cầu dữ liệu mới.

Việc kết nối không liên tục đến cơ sở dữ liệu đã đem lại nhiều thuận lợi, trong đó điểm lợi nhất là việc giảm đi một lưu lượng lớn truy cập vào cơ sở dữ liệu cùng một lúc, tiết kiệm đáng kể tài nguyên bộ nhớ. Giảm thiểu đáng kể vấn đề hàng trăm ngàn kết nối cùng truy cập vào cơ sở dữ liệu cùng một lúc.

ADO.NET kết nối vào cơ sở dữ liệu để lấy dữ liệu và kết nối trở lại để cập nhật dữ liệu khi người dùng thay đổi chúng. Hầu hết mọi ứng dụng đều sử dụng nhiều thời gian cho việc đọc và hiển thị dữ liệu, vì thế ADO.NET đã cung cấp một tập hợp con các đối tượng dữ liệu không kết nối cho các ứng dụng để người dùng có thể đọc và hiển thị chúng mà không cần kết nối vào cơ sở dữ liệu.

Các đối tượng ngắt kết nối này làm việc tương tự đối với các ứng dụng Web.

14.1 Cơ sở dữ liệu và ngôn ngữ truy vấn SQL

Để có thể hiểu rõ được cách làm việc của ADO.NET, chúng ta cần phải nắm được một số khái niệm cơ bản về cơ sở dữ liệu quan hệ và ngôn ngữ truy vấn dữ liệu, như: khái niệm về dòng, cột, bảng, quan hệ giữa các bảng, khóa chính, khóa ngoại và cách truy vấn dữ liệu trên các bảng bằng ngôn ngữ truy vấn SQL : SELECT, UPDATE, DELETE … hay cách viết các thủ tục (Store Procedure) …. Trong phạm vi của tài liệu này, chúng ta sẽ không đề cập đến các mục trên.

Trong các ví dụ sau, chúng ta sẽ dùng cơ sở dữ liệu NorthWind, được cung cấp bởi Microsoft để minh họa cho các ví dụ của chúng ta.

14.2 Một số loại kết nối hiện đang sử dụng

1982 ra đời ODBC driver (Open Database Connectivity) của Microsoft. Chỉ truy xuất được thông tin quan hệ, không truy xuất được dữ liệu không quan hệ như : tập tin văn bản, email …Ta phải truy cập ODBC thông qua DSN.

Để truy cập được tất cả Datastore, dùng OLEDB provider thông qua ODBC. Là vỏ bọc của ODBC hoặc không. OLEDB dễ sử dụng hơn ODBC, nhưng chỉ có 1 số ít ngôn ngữ có thể hiểu được (C++), vì thế ra đời ADO. OLEDB là giao diện ở mức lập trình hệ thống để quản lý dữ liệu. OLEDB đơn giản chỉ là một tập các giao diện COM đóng gói thành các system service để quản trị các CSDL khác nhau. Gồm 4 đối tượng chính : Datasource, Session, Command, Rowset.

ADO là một COM, do đó được dùng với bất kỳ ngôn ngữ nào tương thích với COM. ADO không độc lập OS, nhưng độc lập ngôn ngữ : C++,VB, JavaScript, VBScript …Là vỏ bọc của OLEDB và ADO gồm 3 đối tượng chính : Connection, Command, Recordset.

Remote Data Services (RDS) của Microsoft cho phép dùng ADO thông qua các giao thức HTTP, HTTPS và DCOM để truy cập dữ liệu qua Web.

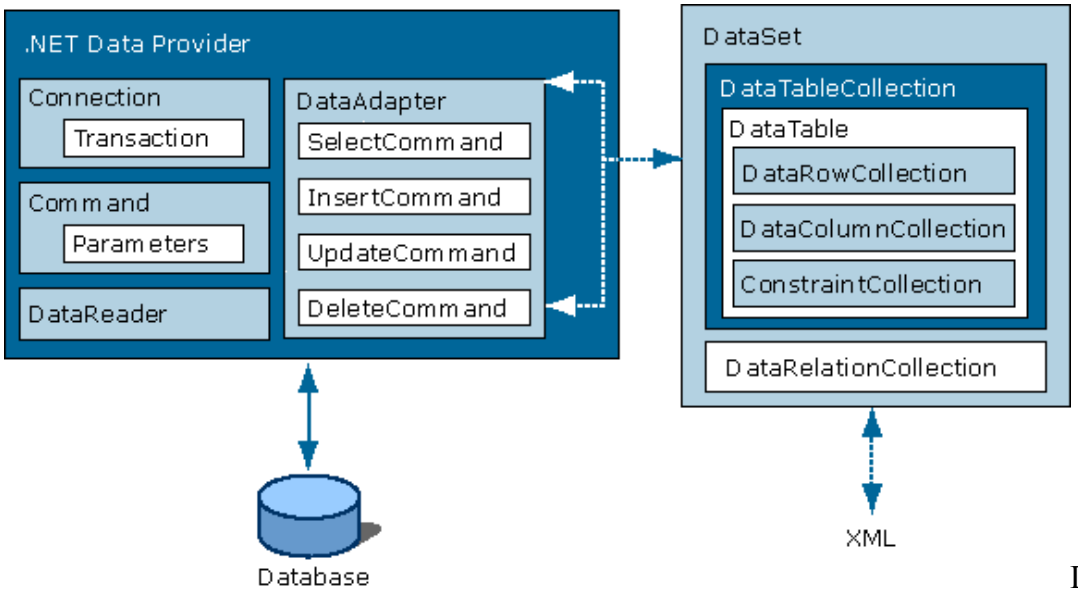
Microsoft Data Access Components (MDAC) là tổ hợp của ODBC, OLEDB, ADO và cả RDS.

Ta có thể kết nối dữ liệu bằng một trong các cách: dùng ODBC driver (DSN), dùng OLEDB thông qua ODBC hoặc OLEDB **không** thông qua ODBC.

14.3 Kiến trúc ADO.NET

ADO.NET được chia ra làm hai phần chính rõ rệt, được thể hiện qua hình

Hình 14-1 Kiến trúc ADO.NET



D

DataSet là thành phần chính cho đặc trưng kết nối không liên tục của kiến trúc ADO.NET. DataSet được thiết kế để có thể thích ứng với bất kỳ nguồn dữ liệu nào. DataSet chứa một hay nhiều đối tượng DataTable mà nó được tạo từ tập các dòng và cột dữ liệu, cùng với khoá chính, khóa ngoại, ràng buộc và các thông tin liên

quan đến đối tượng DataTable này. Bản thân DataSet được dạng như một tập tin XML.

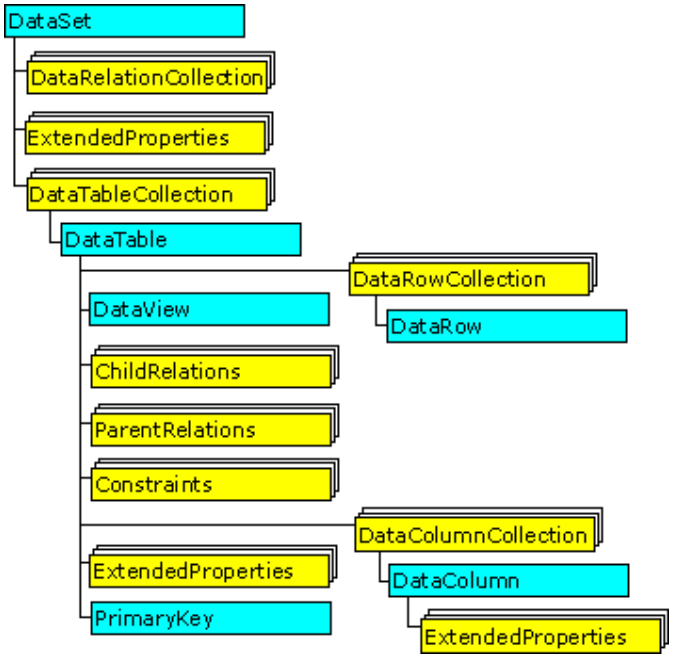
Thành phần chính thứ hai của ADO.NET chính là NET Provider Data, nó chứa các đối tượng phục vụ cho việc thao tác trên cơ sở dữ liệu được hiệu quả và nhanh chóng, nó bao gồm một tập các đối tượng Connection, Command, DataReader và DataAdapter. Đối tượng Connection cung cấp một kết nối đến cơ sở dữ liệu, Command cung cấp một thao tác đến cơ sở dữ liệu, DataReader cho phép chỉ đọc dữ liệu và DataAdapter là cấu nối trung gian giữa DataSet và nguồn dữ liệu.

14.4 Mô hình đối tượng ADO.NET

Có thể nói mô hình đối tượng của ADO.NET khá uyển chuyển, các đối tượng của nó được tạo ra dựa trên quan điểm đơn giản và dễ dùng. Đối tượng quan trọng nhất trong mô hình ADO.NET chính là **Dataset. Dataset** có thể được xem như là thể hiện của cả một cơ sở dữ liệu con, lưu trữ trên vùng nhớ cache của máy người dùng mà không kết nối đến cơ sở dữ liệu.

14.4.1 Mô hình đối tượng của Dataset

Hình 14-2 Mô hình đối tượng Dataset



DataSet bao gồm một tập các đối tượng DataRelation cũng như tập các đối tượng DataTable. Các đối tượng này đóng vai trò như các thuộc tính của DataSet.

14.4.2 Đối tượng DataTable và DataColumn

Ta có thể viết mã C# để tạo ra đối tượng **DataTable** hay nhận về từ kết quả của câu truy vấn đến cơ sở dữ liệu. **DataTable** có một số thuộc tính dùng chung (public) như thuộc tính **Columns**, từ thuộc tính này ta có thể truy cập đến đối tượng **DataColumnsCollection** thông qua chỉ mục hay tên của cột để nhận về các đối tượng **DataColumn** thích hợp, mỗi **DataColumn** tương ứng với một cột trong một bảng dữ liệu. Ví dụ :

```
DataTable dt = new DataTable("tenBang");
DataColumn dc = dt.Columns["tenCot"];
```

14.4.3 Đối tượng DataRelation

Ngoài tập các đối tượng **DataTable** được truy cập thông qua thuộc tính **Tables**, **DataSet** còn có một thuộc tính **Relations**. Thuộc tính này dùng để truy cập đến đối tượng **DataRelationCollection** thông qua chỉ mục hay tên của quan hệ và sẽ trả về đối tượng **DataRelation** tương ứng. Ví dụ :

```
DataSet ds = new DataSet("tenDataSet");
DataRelation dre = ds.Relations["tenQuanHe"];
```

14.4.4 Các bản ghi (Rows)

Tương tự như thuộc tính **Columns** của đối tượng **DataTable**, để truy cập đến các dòng ta cũng có thuộc tính **Rows**. ADO. NET không đưa ra khái niệm RecordSet, thay vào đó để duyệt qua các dòng (Row), ta có thể truy cập các dòng thông qua thuộc tính **Rows** bằng vòng lặp **foreach**.

14.4.5 Đối tượng SqlConnection và SqlCommand

Đối tượng SqlConnection đại diện cho một kết nối đến cơ sở dữ liệu, đối tượng này có thể được dùng chung cho các đối tượng SqlCommand khác nhau. Đối tượng SqlCommand cho phép thực hiện một câu lệnh truy vấn trực tiếp : như SELECT, UPDATE hay DELETE hay gọi một thủ tục (Store Procedure) từ cơ sở dữ liệu.

14.4.6 Đối tượng DataAdapter

ADO.NET dùng DataAdapter như là chiếc cầu nối trung gian giữa DataSet và DataSource (nguồn dữ liệu), nó lấy dữ liệu từ cơ sở dữ liệu sau đó dùng phương Fill() để đẩy dữ liệu cho đối tượng DataSet. Nhờ đối tượng DataAdapter này mà DataSet tồn tại tách biệt, độc lập với cơ sở dữ liệu và một DataSet có thể là thể hiện của một hay nhiều cơ sở dữ liệu. Ví dụ :

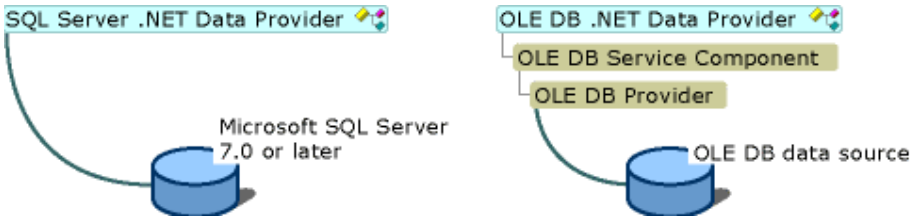
```
//Tạo đối tượng SqlDataAdapter
SqlDataAdapter sda = new SqlDataAdapter();

// cung cấp cho sda một SqlCommand và SqlConnection ...
// lấy dữ liệu ...

//tạo đối tượng DataSet mới
DataSet ds = new DataSet("tenDataSet");
```

```
//Đẩy dữ liệu trog sda vào ds bằng hàm Fill();
sda.Fill(ds);
```

14.5 Trình cung cấp dữ liệu (.NET Data Providers)

.NET Framework hỗ trợ hai trình cung cấp dữ liệu là **SQL Server .NET Data Provider** (dành cho phiên bản SQL Server 7.0 của Microsoft trở lên) và **OLE DB .NET Data Provider** (dành cho các hệ quản trị cơ sở dữ liệu khác) để truy cập vào cơ sở dữ liệu.

Hình 14-3 So sánh SQL Server .NET Data Provider và the OLE DB .NET Data Provider



SQL Server .NET Data Provider có các đặc điểm :

- Dùng nghi thức riêng để truy cập cơ sở dữ liệu
- Truy xuất dữ liệu sẽ nhanh hơn và hiệu quả hơn do không phải thông qua lớp OLE DB Provider hay ODBC
- Chỉ được dùng với hệ quản trị cơ sở dữ liệu SQL Server 7.0 trở lên.
- Được Mircrosoft hỗ trợ khá hoàn chỉnh.

OLE DB .NET Data Provider có các đặc điểm :

- Phải thông qua 2 lớp vì thế sẽ chậm hơn
- Thực hiện được các dịch vụ “Connection Pool”
- Có thể truy cập vào mọi Datasource có hỗ trợ OLE DB Provider thích hợp

14.6 Khởi sự với ADO.NET

Để có thể hiểu rõ được ADO.NET, ngoài lý thuyết ra, chúng ta sẽ khảo sát chi tiết về cách chúng hoạt động ra bằng mã lệnh cụ thể.

Ví dụ Windows Form dưới đây sẽ dùng một ListBox để lấy dữ liệu từ bảng Custommers trong cơ sở dữ liệu NorthWind.

Đầu tiên ta sẽ tạo ra đối tượng DataAdapter :

```
SqlDataAdapter DataAdapter = new SqlDataAdapter(
                commandString, connectionString);
```

Hàm khởi tạo của đối tượng này gồm hai tham số commandString và connectionString. commandString là chuỗi chứa câu lệnh truy vấn trên dữ liệu mà ta muốn nhận về :

```
string commandString =
       "Select CompanyName, ContactName from Customers";
```

Biến connectString chứa các thông số để kết nối đến cơ sở dữ liệu. Ứng dụng của ta dùng hệ quản trị cơ sở dữ liệu SQL Server, vì thế để đơn giản ta sẽ để đối số password là trống, uid là sa, máy chủ server là localhost và tên cơ sở dữ liệu là NorthWind :

```
string connectionString =
     "server=localhost; uid=sa; pwd=; database=northwind";
```

Với đối tượng DataAdapter được tạo ở trên, ta sẽ tạo ra một đối tượng DataSet mới và đẩy dữ liệu vào nó bằng phương thức Fill() của đối tương DataAdapter.

```
DataSet dataSet = new DataSet( );
DataAdapter.FillDataSet(dataSet,"Customers");
```

Đối tượng DataSet chứa một tập các DataTable, nhưng ở đây ta chỉ cần lấy dữ liệu của bảng đầu tiên là “Customers” :

```
DataTable dataTable = dataSet.Tables[0];
```

Ta sẽ duyệt qua từng dòng của bảng bằng vòng lặp foreach để lấy về từng DataRow một, sau đó sẽ truy cập đến trường cần lấy dữ liệu thông qua tên cột, rồi thêm vào ListBox.

```
foreach (DataRow dataRow in dataTable.Rows)
{
  lbCustomers.Items.Add(   dataRow["CompanyName"] +
    " (" + dataRow["ContactName"] + ")" );
}
```

Sau đây là đoạn mã đầy đủ của ứng dụng :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWinForm
{
  public class ADOForm1 : System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components;
    private System.Windows.Forms.ListBox lbCustomers;
    public ADOForm1( )
    {
      InitializeComponent( );

      // kết nối đến máy chủ, cơ sở dữ liệu northwind
      string connectionString =
      "server=localhost; uid=sa; pwd=; database=northwind";
```

```
            // lấy các dòng dữ liệu từ bảng Customers
            string commandString =
            "Select CompanyName, ContactName from Customers";

            // tạo ra đối tượng DataAdapter và DataSet
            SqlDataAdapter DataAdapter =
            new SqlDataAdapter(commandString, connectionString);
            DataSet DataSet = new DataSet( );

            // đẩy dữ liệu vào DataSet
            DataAdapter.Fill(DataSet,"Customers");

            // lấy về một bảng dữ liệu
            DataTable dataTable = DataSet.Tables[0];

            // duyệt từng dòng để lấy dữ liệu thêm vào ListBox
            foreach (DataRow dataRow in dataTable.Rows)
            {
              lbCustomers.Items.Add(dataRow["CompanyName"] +
                " (" + dataRow["ContactName"] + ")" );
            }
          }

          public override void Dispose( )
          {
            base.Dispose( );
            components.Dispose( );
          }

          private void InitializeComponent( )
          {
            this.components = new System.ComponentModel.Container();
            this.lbCustomers = new System.Windows.Forms.ListBox();
            lbCustomers.Location = new System.Drawing.Point(48, 24);
            lbCustomers.Size = new System.Drawing.Size(368, 160);
            lbCustomers.TabIndex = 0;
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(464, 273);
            this.Controls.Add(this.lbCustomers);
          }

          public static void Main(string[] args)
          {
            Application.Run(new ADOForm1( ));
          }
        }
    }
```

Chỉ với một số dòng mã ta đã có thể lấy dữ liệu và hiện thị trong hộp ListBox :

Hình 14-4 Kết xuất của ví dụ trên.

Để hoàn chỉnh giao tác trên, ta cần thực hiện tám dòng mã chính :

- Tạo ra chuỗi kết nối vào cơ sở dữ liệu

```
string connectionString =
     "server=myServer; uid=sa; pwd=; database=northwind";
```

- Tạo câu lênh truy vấn chọn dữ liệu

```
string commandString =
       "Select CompanyName, ContactName from Customers";
```

- Tạo đối tượng DataAdapter và chuyển cho nó chuỗi truy vấn và kết nối

```
SqlDataAdapter DataAdapter = new SqlDataAdapter(
                      commandString, connectionString);
```

- Tạo đối tượng DataSet mới

```
DataSet dataSet = new DataSet( );
```

- Đẩy bảng dữ liệu Customers lấy từ DataAdapter vào dataSet

```
DataAdapter.Fill(dataSet,"Customers");
```

- Trích đối tượng DataTable từ dataSet trên

```
DataTable dataTable = DataSet.Tables[0];
```

- Đẩy dữ liệu trong bảng dataTable vào ListBox

```
foreach (DataRow dataRow in dataTable.Rows)
{
  lbCustomers.Items.Add(dataRow["CompanyName"] +
  " (" + dataRow["ContactName"] + ")" );
}
```

14.7 Sử dụng trình cung cấp dữ liệu được quản lý

Ở ví dụ trên chúng ta đã khảo sát qua cách truy cập dữ liệu thông qua trình cung cấp dữ liệu **SQL Server .NET Data Provider**. Trong phần này chúng ta sẽ tiếp tục khảo sát sang trình cung cấp dữ liệu **OLE DB .NET Data Provider**, với trình cung cấp dữ liệu này ta có thể kết nối đến bất kỳ hệ quản trị cơ sở dữ liệu nào có hỗ trợ trình cung cấp dữ liệu **OLE DB Providers**, cụ thể là **Microsoft Access.**

So với ứng dụng trên, ta chỉ cần thay đổi một vào dòng mã là có thể hoạt động được. Đầu tiên là chuỗi kết nối :

```
string connectionString = "provider=Microsoft.JET.OLEDB.4.0; "
                      + "data source = c:\\northwind.mdb";
```

Chuỗi trên sẽ kết nối đến cơ sở dữ liệu northwind trên ổ đĩa C.

Kế tiếp ta thay đổi đối tượng DataAdapter từ SqlDataAdapter sang OleDbDataAdapter

```
OleDbDataAdapter DataAdapter = new OleDbDataAdapter(
                        commandString, connectionString);
```

Chúng ta phải đảm bảo là namespace OleDb được thêm vào ứng dụng :

```
using System.Data.OleDb;
```

Phần mã còn lại thì tương tự như ứng dụng trên, sau đây sẽ trích ra một đoạn mã chính phục vụ cho việc kết nối theo cách này :

```
public ADOForm1( )
{
   InitializeComponent( );

   // chuỗi kết nối đến cơ sở dữ liệu
   string connectionString = "provider=Microsoft.JET.OLEDB.4.0;"
                           + "data source = c:\\nwind.mdb";

   // chuỗi truy vấn dữ liệu
   string commandString =
            "Select CompanyName, ContactName from Customers";

   // tạo đối tượng OleDbDataAdapter và DataSet mới
   OleDbDataAdapter DataAdapter = new OleDbDataAdapter(
                        commandString, connectionString);
   DataSet dataSet = new DataSet( );

   // đẩy dữ liệu vào dataSet
   DataAdapter.Fill(DataSet,"Customers");

   // lây về bảng dữ liệu Customers
   DataTable dataTable = DataSet.Tables[0];

   // duyệt qua từng dòng dữ liệu
   foreach (DataRow dataRow in dataTable.Rows)
   {
      lbCustomers.Items.Add(dataRow["CompanyName"] +
            " (" + dataRow["ContactName"] + ")" );
   }
}
```

14.8 Làm việc với các điều khiển kết buộc dữ liệu

ADO.NET hỗ trợ khá hoàn chỉnh cho các điều khiển kết buộc dữ liệu (Data-Bound), các điều khiển này sẽ nhận vào một DataSet, sau khi gọi hàm DataBind() thì dữ liệu sẽ tự động được hiển thị lên điều khiển.

14.8.1 Đẩy dữ liệu vào điều khiển lưới DataGrid

Ví dụ sau sẽ dùng điều khiển lưới DataGrid để thực hiện kết buộc dữ liệu, điều khiển lưới này được hỗ trợ cho cả ứng dụng Windows Forms và WebForms.

Trong ứng dụng trước, ta phải duyệt qua từng dòng của đối tượng DataTable để lấy dữ liệu, sau đó hiển thị chúng lên điều khiển ListBox. Trong ứng dụng này công việc hiển thị dữ liệu lên điều khiển được thực hiện đơn giản hơn, ta chỉ cần lấy về đối tượng DataView của DataSet, sau đó gán DataView này cho thuộc tính DataSource của điều khiển lưới, sau đó gọi hàm DataBind() thì tự động dữ liệu sẽ được đẩy lên điều khiển lưới dữ liệu.

```
CustomerDataGrid.DataSource =
                                DataSet.Tables["Customers"].DefaultView;
```

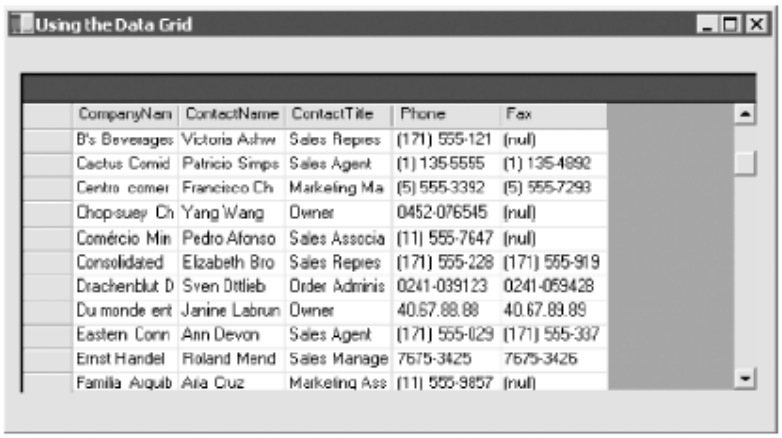
Trước tiên ta cần tạo ra đối tượng lưới trên Form bằng cách kéo thả, đặt tên lại cho điều khiển lưới là CustomerDataGrid. Sau đây là mã hoàn chỉnh của ứng dụng kết buộc dữ liệu cho điều khiển lưới :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWindows.Form
{
  public class ADOForm3 : System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components;
    private System.Windows.Forms.DataGrid CustomerDataGrid;

    public ADOForm3( )
    {
      InitializeComponent( );

      // khởi tạo chuỗi kết nối và chuỗi truy vấn dữ liệu
      string connectionString =
      "server=localComputer; uid=sa; pwd=;database=northwind";
      string commandString =
        "Select CompanyName, ContactName, ContactTitle, "
              + "Phone, Fax from Customers";

      // tạo ra một SqlDataAdapter và DataSet mới,
      // đẩy dữ liệu cho DataSet
      SqlDataAdapter DataAdapter =
        new SqlDataAdapter(commandString, connectionString);
      DataSet DataSet = new DataSet( );
      DataAdapter.Fill(DataSet,"Customers");

      // kết buộc dữ liệu của DataSet cho lưới
      CustomerDataGrid.DataSource=
      DataSet.Tables["Customers"].DefaultView;
    }

    public override void Dispose( )
    {
      base.Dispose( );
      components.Dispose( );
    }
```

```
        private void InitializeComponent( )
        {
           this.components = new System.ComponentModel.Container();
           this.CustomerDataGrid =    new DataGrid();
           CustomerDataGrid.BeginInit();
           CustomerDataGrid.Location =
                        new System.Drawing.Point (8, 24);
           CustomerDataGrid.Size = new System.Drawing.Size (656, 224);
           CustomerDataGrid.DataMember = "";
           CustomerDataGrid.TabIndex = 0;
           CustomerDataGrid.Navigate +=
                new NavigateEventHandler(this.dataGrid1_Navigate);
           this.Text = "ADOFrm3";
           this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
           this.ClientSize = new System.Drawing.Size (672, 273);
           this.Controls.Add (this.CustomerDataGrid);
           CustomerDataGrid.EndInit ( );
        }

        public static void Main(string[] args)
        {
           Application.Run(new ADOForm3());
        }
     }
  }
```

Điều khiển lưới sẽ hiển thị y hệt mọi dữ liệu hiện có trong bảng Customers, tên của các cột trên lưới cũng chính là tên của các cột trong bản dữ liệu. Giao diện của ứng dụng sau khi chạy chương trình :

Hình 14-5 Kết buộc dữ liệu cho điều khiển lưới DataGrid.



14.8.2 Tạo đối tượng DataSet

Trong ví dụ trước, tạo ra đối tượng SqlDataAdapter bằng cách gắn trực tiếp chuỗi kết nối và chuỗi truy vấn vào nó. Đối tượng Connection và Command sẽ được tạo và tích hợp vào trong đối tượng DataAdapter này. Với cách này, ta sẽ bị hạn chế trong các thao tác liên quan đến cơ sở dữ liệu.

```
SqlDataAdapter DataAdapter =
     new SqlDataAdapter(commandString, connectionString);
```

Ví dụ sau đây sẽ minh họa việc lấy về đối tượng DataSet bằng cách tạo ra các đối tượng Connection và Command một cách riêng biệt, khi ta cần dùng lại chúng hay muốn thực hiện hoàn chỉnh một thao tác thì sẽ thuận lợi hơn.

Đầu tiên ta sẽ khai báo bốn biến thành viên thuộc lớp, như sau :

```
private System.Data.SqlClient.SqlConnection myConnection;
private System.Data.DataSet myDataSet;
private System.Data.SqlClient.SqlCommand myCommand;
private System.Data.SqlClient.SqlDataAdapter DataAdapter;
```

Đối tượng Connection sẽ được tạo riêng với chuỗn kết nối :

```
string connectionString =
     "server=localhost; uid=sa; pwd=; database=northwind";
myConnection = new
          System.Data.Sql.SqlConnection(connectionString);
```

Sau đó ta sẽ mở kết nối :

```
myConnection.Open( );
```

Ta có thể thực hiện nhiều giao tác trên cơ sở dữ liệu khi kết nối được mở và sau khi dùng xong ta chỉ đơn giản đóng kết nối lại. Tiếp theo ta sẽ tạo ra đối tượng DataSet:

```
myDataSet = new System.Data.DataSet( );
```

Và tiếp tục tạo đối tượng Command, gắn cho nó đối tượng Connection đã mở và chuỗi truy vấn dữ liệu :

```
myCommand = new System.Data.SqlClient.SqlCommand( )
myCommand.Connection=myConnection;
myCommand.CommandText = "Select * from Customers";
```

Cuối cùng ta cần tạo ra đối tượng SqlDataAdapter, gắn đối tượng SqlCommand vừa tạo ở trên cho nó, đồng thời phải tiến hành ánh xạ bảng dữ liệu nó nhận được từ câu truy vấn của đối tượng Command để tạo sự đồng nhất về tên các cột khi đẩy bảng dữ liệu này vào DataSet.

```
DataAdapter = new System.Data.SqlClient.SqlDataAdapter( );
DataAdapter.SelectCommand= myCommand;
DataAdapter.TableMappings.Add("Table","Customers");
DataAdapter.Fill(myDataSet);
```

Bây giờ ta chỉ việc gắn DataSet vào thuộc tính DataSoucre của điều khiển lưới:

```
dataGrid1.DataSource=myDataSet.Tables["Customers"].DefaultView;
```

Dưới đây là mã hoàn chỉnh của ứng dụng này :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWindows.Form
{
```

```
public class ADOForm1 : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components;
   private System.Windows.Forms.DataGrid dataGrid1;
   private System.Data.SqlClient.SqlConnection myConnection;
   private System.Data.DataSet myDataSet;
   private System.Data.SqlClient.SqlCommand myCommand;
   private System.Data.SqlClient.SqlDataAdapter DataAdapter;

   public ADOForm1( )
   {
      InitializeComponent( );

      // tạo đối tượng connection và mở nó
      string connectionString =
         "server=Neptune; uid=sa; pwd=oWenmEany;" +
         "database=northwind";
      myConnection = new SqlConnection(connectionString);
      myConnection.Open();

      // tạo đối tượng DataSet mới
      myDataSet = new DataSet( );

      // tạo đối tượng command mới và gắn cho đối tượng
      //connectio và chuỗi truy vấn cho nó
      myCommand = new System.Data.SqlClient.SqlCommand( );
      myCommand.Connection=myConnection;
      myCommand.CommandText = "Select * from Customers";

      // tạo đối tượng DataAdapter với đối tượng Command vừa
      // tạo ở trên, đồng thời thực hiện ánh xạ bảng dữ liệu
      DataAdapter = new SqlDataAdapter( );
      DataAdapter.SelectCommand= myCommand;
      DataAdapter.TableMappings.Add("Table","Customers");

      // đẩy dữ liệu vào DataSet
      DataAdapter.Fill(myDataSet);

      // gắn dữ liệu vào lưới
      dataGrid1.DataSource =
                myDataSet.Tables["Customers"].DefaultView;
   }

   public override void Dispose()
   {
      base.Dispose();
      components.Dispose();
   }

   private void InitializeComponent( )
   {
      this.components = new System.ComponentModel.Container();
      this.dataGrid1 = new System.Windows.Forms.DataGrid();
      dataGrid1.BeginInit();
      dataGrid1.Location = new System.Drawing.Point(24, 32);
      dataGrid1.Size = new System.Drawing.Size(480, 408);
      dataGrid1.DataMember = "";
      dataGrid1.TabIndex = 0;
```

```
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(536, 501);
            this.Controls.Add(this.dataGrid1);
            dataGrid1.EndInit( );
        }

        public static void Main(string[] args)
        {
            Application.Run(new ADOForm1());
        }
    }
}
```

Giao diện của ví dụ này cũng tương tự như các ví dụ trên.

14.8.3 Kết hợp giữa nhiều bảng

Các ví dụ ở trên chỉ đơn thuần lấy dữ liệu từ trong một bảng. Ở ví dụ này ta sẽ tìm hiểu về cách lấy dữ liệu trên hai bảng. Trong cơ sở dữ liệu của ta, một khách hàng có thể có nhiều hóa đơn khác nhau, vì thế ta sẽ có quan hệ một nhiều giữa bảng khách hàng (Customers)và bảng hóa đơn (Orders). Bảng Orders sẽ chứa thuộc tính CustomersId của bảng Customers, thuộc tính này đóng vai trò là khóa chính đối bảng Customers và khóa ngoại đối với bảng Orders.

Ứng dụng của ta sẽ hiển thị dữ liệu của hai bảng Customers và Orders trên cùng một lưới và thể hiện quan hệ một nhiều của hai bảng ngay trên lưới. Để làm được điều này ta chỉ cần dùng chung một đối tượng Connetion, hai đối tượng tượng SqlDataAdapter và hai đối tượng SqlCommand.

Sau khi tạo đối tượng SqlDataAdapter cho bảng Customers tương tự như ví dụ trên, ta tiến tạo tiếp đối tượng SqlDataAdapter cho bảng Orders :

```
myCommand2 = new System.Data.SqlClient.SqlCommand();
DataAdapter2 = new System.Data.SqlClient.SqlDataAdapter();
myCommand2.Connection = myConnection;
myCommand2.CommandText = "SELECT * FROM Orders";
```

Lưu ý là ở đây đối tượng DataAdapter2 có thể dùng chung đối tượng Connection ở trên, nhưng đối tượng Command thì khác. Sau đó gắn đối tượng Command2 cho DataAdapter2, ánh xạ bảng dữ liệu và đẩy dữ liệu vào DataSet ở trên.

```
DataAdapter2.SelectCommand = myCommand2;
DataAdapter2.TableMappings.Add ("Table", "Orders");
DataAdapter2.Fill(myDataSet);
```

Tại thời điểm này, ta có một đối tượng DataSet nhưng chứa hai bảng dữ liệu : Customers và Orders. Do ta cần thể hiện cả quan hệ của hai bảng ngay trên điều khiển lưới, cho nên ta cần phải định nghĩa quan hệ này cho đối tượng DataSet của chúng ta. Nếu không làm điều này thì đối tượng DataSet sẽ bỏ qua quan hệ giữa 2 bảng này.

Do đó ta cần khai báo thêm đối tương DataRelation :

```
System.Data.DataRelation dataRelation;
```

Do mỗi bảng Customers và Orders đều có chứa một thuộc tính CustomersId, nên ta cũng cần khái báo thêm hai đối tượng DataColumn tương ứng với hai thuộc tính này.

```
System.Data.DataColumn dataColumn1;
System.Data.DataColumn dataColumn2;
```

Mỗi một DataColumn sẽ giữ giá trị của một cột trong bảng của đối tượng DataSet :

```
dataColumn1 = myDataSet.Tables["Customers"].Columns["CustomerID"];
dataColumn2 = myDataSet.Tables["Orders"].Columns["CustomerID"];
```

Ta tiến hành tạo quan hệ cho hai bảng bằng cách gọi hàm khởi tạo của đối tượng DataRelation, truyền vào cho nó tên quan hệ và hai cột cần tạo quan hệ :

```
dataRelation =new System.Data.DataRelation("CustomersToOrders",
                              dataColumn1, dataColumn2);
```

Sau khi tạo được đối tượng DataRelation, ta thêm vào DataSet của ta. Sau đó ta cần tạo một đối tượng quản lý khung nhìn DataViewManager cho DataSet, đối tượng khung nhìn này sẽ được gán cho lưới điều khiển để hiển thị:

```
myDataSet.Relations.Add(dataRelation);
DataViewManager DataSetView = myDataSet.DefaultViewManager;
dataGrid1.DataSource = DataSetView;
```

Do điều khiển lưới phải hiển thị quan hệ của hai bảng dữ liệu, nên ta phải chỉ cho nó biết là bảng nào sẽ là bảng cha. Ở đây bảng cha là bảng Customers :

```
dataGrid1.DataMember= "Customers";
```

Sau đây là mã hoàn chỉnh của toàn bộ ứng dụng :

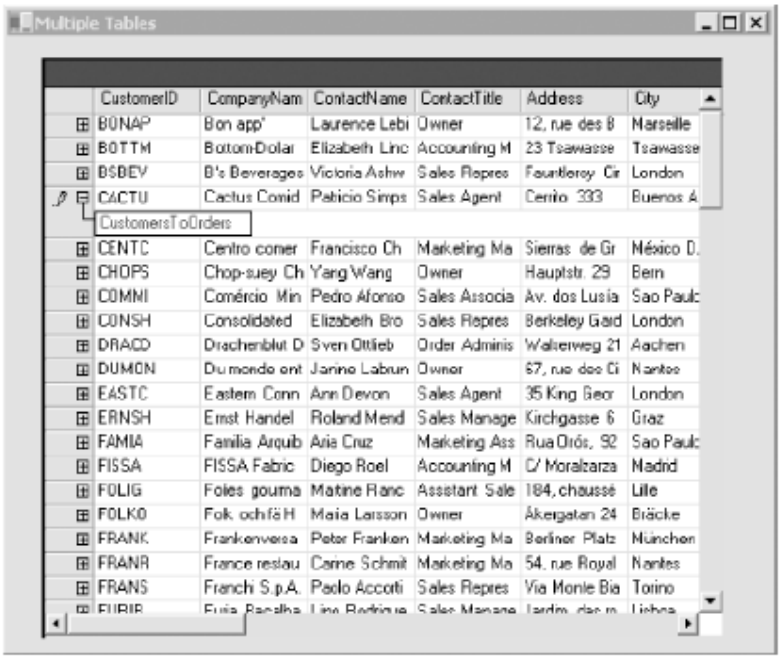
```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWindows.Form
{
  public class ADOForm1 : System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components;
    private System.Windows.Forms.DataGrid dataGrid1;
    private System.Data.SqlClient.SqlConnection myConnection;
    private System.Data.DataSet myDataSet;
    private System.Data.SqlClient.SqlCommand myCommand;
    private System.Data.SqlClient.SqlCommand myCommand2;
    private System.Data.SqlClient.SqlDataAdapter DataAdapter;
    private System.Data.SqlClient.SqlDataAdapter DataAdapter2;

    public ADOForm1( )
    {
      InitializeComponent( );

      // tạo kết nối
      string connectionString = "server=Neptune; uid=sa;" +
                    " pwd=oWenmEany; database=northwind";
```

```
    myConnection = new SqlConnection(connectionString);
    myConnection.Open( );

    // tạo DataSet
    myDataSet = new System.Data.DataSet( );

    // tạo đối tượng Command và DataSet cho bảng Customers
    myCommand = new System.Data.SqlClient.SqlCommand( );
    myCommand.Connection=myConnection;
    myCommand.CommandText = "Select * from Customers";
    DataAdapter =new System.Data.SqlClient.SqlDataAdapter();
    DataAdapter.SelectCommand= myCommand;
    DataAdapter.TableMappings.Add("Table","Customers");
    DataAdapter.Fill(myDataSet);

    // tạo đối tượng Command và DataSet cho bảng Orders
    myCommand2 = new System.Data.SqlClient.SqlCommand( );
    DataAdapter2=new System.Data.SqlClient.SqlDataAdapter();
    myCommand2.Connection = myConnection;
    myCommand2.CommandText = "SELECT * FROM Orders";
    DataAdapter2.SelectCommand = myCommand2;
    DataAdapter2.TableMappings.Add ("Table", "Orders");
    DataAdapter2.Fill(myDataSet);

    // thiết lập quan hệ giữa 2 bảng
    System.Data.DataRelation dataRelation;
    System.Data.DataColumn dataColumn1;
    System.Data.DataColumn dataColumn2;
    dataColumn1 =
      myDataSet.Tables["Customers"].Columns["CustomerID"];
    dataColumn2 =
      myDataSet.Tables["Orders"].Columns["CustomerID"];
    dataRelation = new System.Data.DataRelation(
      "CustomersToOrders", dataColumn1, dataColumn2);

    // thêm quan hệ trên vào DataSet
    myDataSet.Relations.Add(dataRelation);

    // Đặt khung nhìn và bảng hiển thị trước cho lưới
    DataViewManager DataSetView =
                          myDataSet.DefaultViewManager;
    dataGrid1.DataSource = DataSetView;
    dataGrid1.DataMember= "Customers";
}

public override void Dispose( )
{
    base.Dispose( );
    components.Dispose( );
}

private void InitializeComponent( )
{
    this.components = new System.ComponentModel.Container();
    this.dataGrid1 = new System.Windows.Forms.DataGrid();
    dataGrid1.BeginInit( );
    dataGrid1.Location = new System.Drawing.Point(24, 32);
    dataGrid1.Size = new System.Drawing.Size(480, 408);
```

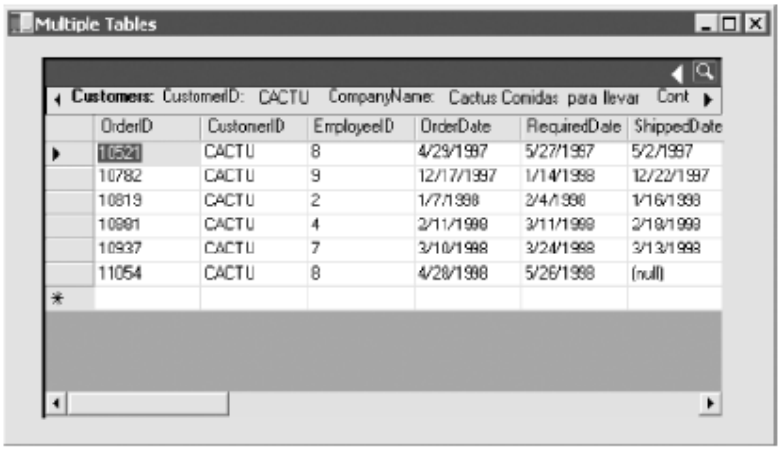
```
            dataGrid1.DataMember = "";
            dataGrid1.TabIndex = 0;
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size (536, 501);
            this.Controls.Add (this.dataGrid1);
            dataGrid1.EndInit ( );
         }

         public static void Main(string[] args)
         {
            Application.Run(new ADOForm1( ));
         }
      }
   }
```

Khi chạy ứng dụng sẽ có giao diện nhu sau :

Hình 14-6 Quan hệ một nhiều giữa hai bảng Customers và Orders



Hình 14-7 Danh sách các hóa đơn tương ứng với khách hàng được chọn



14.9 Thay đổi các bản ghi của cơ sở dữ liệu

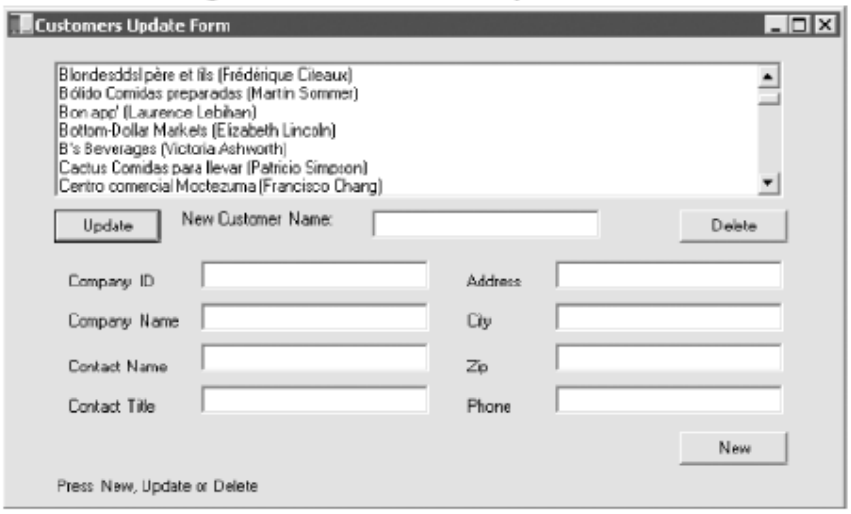
Tới lúc này, chúng ta đã học cách lấy dữ liệu từ cơ sở dữ liệu sau đó hiển thị chúng ra màn hình dựa vào các điều khiển có hay không kết buộc dữ liệu. Phần này chúng ta sẽ tìm hiểu cách cập nhật vào cơ sở dữ liệu. Các thao tác trên cơ sở dữ liệu như : Thêm, xóa và sửa một dòng trong các bảng dữ liệu. Sau đây là luồng công việc hoàn chỉnh khi ta có một thao tác cập nhật cơ sở dữ liệu :

1. Đẩy dữ liệu của bảng vào DataSet bằng câu truy vấn SQL hay gọi thủ tục từ cơ sở dữ liệu
2. Hiển thị dữ liệu trong các bảng có trong DataSet bằng cách kết buộc hay duyệt qua các dòng dữ liệu.
3. Hiệu chỉnh dữ liệu trong các bảng DataTable với các thao tác thêm, xóa hay sửa trên dòng DataRow.
4. Gọi phương thúc GetChanges() để lấy về một DataSet khác chứa tất cả các thay đổi trên dữ liệu.
5. Kiểm tra lỗi trên DataSet mới được tạo này bằng thuộc tính HasErrors. Nếu có lỗi thì ta sẽ tiến hành kiểm tra trên từng bảng DataTable của DataSet, khi gặp một bảng có lỗi thì ta tiếp tục dùng hàm GetErrors()để lấy về các dòng DataRow có lỗi, ứng với từng dòng ta sẽ dùng thuộc tính RowError trên dòng để xác định xem dòng đó có lỗi hay không để có thể đưa ra xử lý thích hợp.
6. Trộn hai DataSet lại thành một.
7. Gọi phương thức Update() của đối tượng DataAdapter với đối số truyền vào là DataSet vừa có trong thao tác trộn ở trên để cập nhật các thay đổi vào cơ sở dữ liệu.

8. Gọi phương thức AcceptChanges() của DataSet để cập nhật các thay đổi vào DataSet này hay phương thức RejectChanges() nếu từ chối cập nhật thay đổi cho DataSet hiện hành.

Với luồng công việc trên, cho phép ta có thể kiểm soát tốt được việc thay đổi trên cơ sở dữ liệu hay việc gỡ lỗi cũng thuận tiện hơn. Trong ví dụ dưới đây , ta sẽ cho hiện thị dữ liệu trong bảng Customers lên một ListBox, sau đó ta tiến hành các thao tác thêm, xóa hay sửa trên cơ sở dữ liệu. Để dễ hiểu, ta giảm bớt một số thao tác quản lý ngoại lệ hay lỗi, chỉ tập trung vào mục đích chính của ta. Giao diện chính của ứng dụng sau khi hoàn chỉnh :

Hình 14-8 Hiệu chỉnh dữ liệu trên bảng Customers.



Trong Form này, ta có một ListBox lbCustomers liệt kê các khách hàng, một Button btnUpdate cho việc cập nhật dữ liệu, một Button Xóa, ứng với nút thêm mới ta có tám hộp thoại TextBox để nhận dữ liệu gõ vào từ người dùng. Đồng thời ta có thêm một lblMessage để hiển thị các thông báo ứng với các thao tác trên.

14.9.1 Truy cập và hiển thị dữ liệu

Ta sẽ tạo ra ba biến thành viên : DataAdapter, DataSet và Command :

```
private SqlDataAdapter DataAdapter;
private DataSet DataSet;
private DataTable dataTable;
```

Việc khai báo các biến thành viên như vậy sẽ giúp ta có thể dùng lại cho các phương thức khác nhau. T khai báo chuỗi kết nối và truy vấn :

```
string connectionString =
  "server=localhost; uid=sa; pwd=; database=northwind";
string commandString = "Select * from Customers";
```

Các chuỗi được dùng làm đối số để tạo đối tượng DataAdapter :

```
DataAdapter=new SqlDataAdapter(commandString,ConnectionString);
```

Tạo ra đối tượng DataSet mới, sau đó đẩy dữ liệu từ DataAdapter vào cho nó:

```
DataSet = new DataSet();
DataAdapter.Fill(DataSet,"Customers");
```

Để hiển thị dữ liệu, ta sẽ gọi hàm PopulateDB()để đẩy dữ liệu vào ListBox:

```
dataTable = DataSet.Tables[0];
lbCustomers.Items.Clear( );
foreach (DataRow dataRow in dataTable.Rows)
{
  lbCustomers.Items.Add(dataRow["CompanyName"] +
        " (" + dataRow["ContactName"] + ")" );
}
```

14.9.2 Cập nhật một dòng dữ liệu

Khi người dùng nhấn Button Update (cập nhật), ta sẽ lấy chỉ mục được chọn trên ListBox, và lấy ra dòng dữ liệu DataRow trong bảng ứng với chỉ mục trên. Sau đó cập nhật DataSet với dòng dữ liệu mới này nếu sau khi kiểm tra thấy chúng không có lỗi nào cả. Chi tiết về quá trình thực hiện cập nhật :

Đầu tiên ta sẽ lấy về dòng dữ liệu người dùng muốn thay đổi từ đối tượng dataTable mà ta đã khai báo làm biến thành viên ngay từ đầu :

```
DataRow targetRow = dataTable.Rows[lbCustomers.SelectedIndex];
```

Hiển thị chuỗi thông báo cập nhật dòng dữ liệu đó cho người dùng biết. Để làm điều này ta sẽ gọi phương thức tình DoEvents() của đối tượng Application, hàm này sẽ giúp sơn mới lại màn hình với thông điệp hay các thay đổi khác.

```
lblMessage.Text = "Updating " + targetRow["CompanyName"];
Application.DoEvents();
```

Gọi hàm BeginEdit() của đối tượng DataRow, để chuyển dòng dữ liệu sang chế độ hiệu chỉnh (Edit) và EndEdit()để kết thúc chế độ hiệu chỉnh dòng.

```
targetRow.BeginEdit();
targetRow["CompanyName"] = txtCustomerName.Text;
targetRow.EndEdit();
```

Lấy về các thay đổi trên đối tượng DataSet để kiểm tra xem các thay đổi có xảy ra bất kỳ lỗi nào không. Ở đây ta sẽ dùng một biến cờ có kiểu `true/false` để xác định là có lỗi là `true`, không có lỗi là `false`.Kiểm tra lỗi bằng cách dùng hai vòng lặp tuần tự trên bảng và dòng của DataSet mới lấy về ở trên, ta dùng thuộc tính HasErrors để kiểm tra lỗi trên bảng, phương thức GetErrors()để lấy về các dòng có lỗi trong bảng.

```
DataSet DataSetChanged;
DataSetChanged = DataSet.GetChanges(DataRowState.Modified);
bool okayFlag = true;
if (DataSetChanged.HasErrors)
{
  okayFlag = false;
  string msg = "Error in row with customer ID ";
  foreach (DataTable theTable in DataSetChanged.Tables)
  {
    if (theTable.HasErrors)
```

```
        {
          DataRow[] errorRows = theTable.GetErrors( );
          foreach (DataRow theRow in errorRows)
            msg = msg + theRow["CustomerID"];
        }
      }
      lblMessage.Text = msg;
  }
```

Nếu biến cờ okagFlag là `true`, thì ta sẽ trộn DataSet ban đầu với DataSet thay đổi thành một, sau đó cập nhật DataSet sau khi trộn này vào cơ sở dữ liệu.

```
if (okayFlag)
{
  DataSet.Merge(DataSetChanged);
  DataAdapter.Update(DataSet,"Customers");
```

Tiếp theo hiển thị câu lệnh truy vấn cho người dùng biết, và cập nhật những thay đổi cho DataSet đầu tiên, rồi hiển thị dữ liệu mới lên đối tượng ListBox.

```
lblMessage.Text = DataAdapter.UpdateCommand.CommandText;
Application.DoEvents( );
DataSet.AcceptChanges( );
PopulateLB( );
```

Nếu cờ okayFlag là `false`, có nghĩa là có lỗi trong quá trình hiệu chỉnh dữ liệu, ta sẽ từ chối các thay đổi trên DataSet.

```
else
  DataSet.RejectChanges( );
```

14.9.3 Xóa một dòng dữ liệu

Mã thực thi của sự kiện xóa thì đơn giản hơn một chút, ta nhận về dòng cần xóa :

```
DataRow targetRow = dataTable.Rows[lbCustomers.SelectedIndex];
```

Giữ lại dòng cần xóa để dùng làm thông điệp hiển thị cho người dùng biết trước khi xóa dòng này khỏi cơ sở dữ liệu.

```
string msg = targetRow["CompanyName"] + " deleted. ";,
```

Bắt đầu thực hiện xóa trên bảng dữ liệu, cập nhật thay đổi vào DataSet và cập nhật luôn vào cơ sở dữ liệu :

```
dataTable.Rows[lbCustomers.SelectedIndex].Delete( );
DataSet.AcceptChanges( );
DataAdapter.Update(DataSet,"Customers");
```

Khi gọi hàm AccceptChanges()để cập nhật thay đổi cho DataSet thì nó sẽ lần lượt gọi hàm này cho các DataTable, sau đó cho các DataRow để cập nhật chúng. Ta cũng cần chú ý khi gọi hàm xóa trên bảng Customers, dòng dữ liệu DataRow của khách hàng này chỉ được xóa nếu nó không vi phạm ràng buộc trên các bảng khác, ở đây khách hàng chỉ được xóa nếu nếu khách hàng không có một hóa đơn nào trên bảng Orders. Nếu có ta phải tiến hành xóa trên bảng hóa đơn trước, sau đó mới xóa trên bảng Customers.

14.9.4 Tạo một dòng dữ liệu mới

Sau khi người dùng cung cấp các thông tin về khách hàng cần tạo mới và nhấn Button tạo mới (New), ta sẽ viết mã thực thi trong hàm bắt sự kiện nhấn nút tạo mới này. Đầu tiên ta sẽ tạo ra một dòng mới trên đối tượng DataTable, sau đó gán dữ liệu trên các TextBox cho các cột của dòng mới này :

```
DataRow newRow = dataTable.NewRow( );
newRow["CustomerID"] = txtCompanyID.Text;
newRow["CompanyName"] = txtCompanyName.Text;
newRow["ContactName"] = txtContactName.Text;
newRow["ContactTitle"] = txtContactTitle.Text;
newRow["Address"] = txtAddress.Text;
newRow["City"] = txtCity.Text;
newRow["PostalCode"] = txtZip.Text;
newRow["Phone"] = txtPhone.Text;
```

Thêm dòng mới với dữ liệu vào bảng DataTable, cập nhật vào cơ sở dữ liệu, hiển thị câu truy vấn, cập nhật DataSet, hiển thị dữ liệu mới lên hộp ListBox. Làm trắng các điều khiển TextBox bằng hàm thành viên ClearFields().

```
dataTable.Rows.Add(newRow);
DataAdapter.Update(DataSet,"Customers");
lblMessage.Text = DataAdapter.UpdateCommand.CommandText;
Application.DoEvents( );
DataSet.AcceptChanges( );
PopulateLB( );
ClearFields( );
```

Để hiểu rõ hoàn chỉnh ứng, ta sẽ xem mã hoàn chỉnh của toàn ứng dụng :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWindows.Form
{
  public class ADOForm1 : System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components;
    private System.Windows.Forms.Label label9;
    private System.Windows.Forms.TextBox txtPhone;
    private System.Windows.Forms.Label label8;
    private System.Windows.Forms.TextBox txtContactTitle;
    private System.Windows.Forms.Label label7;
    private System.Windows.Forms.TextBox txtZip;
    private System.Windows.Forms.Label label6;
    private System.Windows.Forms.TextBox txtCity;
    private System.Windows.Forms.Label label5;
    private System.Windows.Forms.TextBox txtAddress;
    private System.Windows.Forms.Label label4;
    private System.Windows.Forms.TextBox txtContactName;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.TextBox txtCompanyName;
```

```
private System.Windows.Forms.Label label2;
private System.Windows.Forms.TextBox txtCompanyID;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Button btnNew;
private System.Windows.Forms.TextBox txtCustomerName;
private System.Windows.Forms.Button btnUpdate;
private System.Windows.Forms.Label lblMessage;
private System.Windows.Forms.Button btnDelete;
private System.Windows.Forms.ListBox lbCustomers;
private SqlDataAdapter DataAdapter;

// biết thành viên DataSet và dataTable cho phép ta sử
//dụng trên nhiều hàm khác nhau
private DataSet DataSet;
private DataTable dataTable;

public ADOForm1( )
{
   InitializeComponent( );
   string connectionString = "server=Neptune; uid=sa;" +
               " pwd=oWenmEany; database=northwind";
   string commandString = "Select * from Customers";
   DataAdapter =
      new SqlDataAdapter(commandString, connectionString);
   DataSet = new DataSet( );
   DataAdapter.Fill(DataSet,"Customers");
   PopulateLB( );
}

// Đẩy dữ liệu vào điều khiển ListBox
private void PopulateLB( )
{
   dataTable = DataSet.Tables[0];
   lbCustomers.Items.Clear( );
   foreach (DataRow dataRow in dataTable.Rows)
   {
      lbCustomers.Items.Add( dataRow["CompanyName"] + " (" +
                        dataRow["ContactName"] + ")" );
   }
}

public override void Dispose( )
{
   base.Dispose( );
   components.Dispose( );
}

private void InitializeComponent( )
{
   this.components = new System.ComponentModel.Container();
   this.txtCustomerName=new System.Windows.Forms.TextBox();
   this.txtCity = new System.Windows.Forms.TextBox();
   this.txtCompanyID = new System.Windows.Forms.TextBox();
   this.lblMessage = new System.Windows.Forms.Label();
   this.btnUpdate = new System.Windows.Forms.Button();
   this.txtContactName= new System.Windows.Forms.TextBox();
   this.txtZip = new System.Windows.Forms.TextBox();
   this.btnDelete = new System.Windows.Forms.Button();
```

```
this.txtContactTitle=new System.Windows.Forms.TextBox();
this.txtAddress = new System.Windows.Forms.TextBox();
this.txtCompanyName=new System.Windows.Forms.TextBox( );
this.label5 = new System.Windows.Forms.Label( );
this.label6 = new System.Windows.Forms.Label( );
this.label7 = new System.Windows.Forms.Label( );
this.label8 = new System.Windows.Forms.Label( );
this.label9 = new System.Windows.Forms.Label( );
this.label4 = new System.Windows.Forms.Label( );
this.lbCustomers = new System.Windows.Forms.ListBox( );
this.txtPhone = new System.Windows.Forms.TextBox( );
this.btnNew = new System.Windows.Forms.Button( );
this.label1 = new System.Windows.Forms.Label( );
this.label2 = new System.Windows.Forms.Label( );
this.label3 = new System.Windows.Forms.Label( );
txtCustomerName.Location =
            new System.Drawing.Point(256, 120);
txtCustomerName.TabIndex = 4;
txtCustomerName.Size = new System.Drawing.Size(160, 20);
txtCity.Location = new System.Drawing.Point(384, 245);
txtCity.TabIndex = 15;
txtCity.Size = new System.Drawing.Size (160, 20);
txtCompanyID.Location =
            new System.Drawing.Point (136, 216);
txtCompanyID.TabIndex = 7;
txtCompanyID.Size = new System.Drawing.Size (160, 20);
lblMessage.Location = new System.Drawing.Point(32, 368);
lblMessage.Text = "Press New, Update or Delete";
lblMessage.Size = new System.Drawing.Size (416, 48);
lblMessage.TabIndex = 1;
btnUpdate.Location = new System.Drawing.Point (32, 120);
btnUpdate.Size = new System.Drawing.Size (75, 23);
btnUpdate.TabIndex = 0;
btnUpdate.Text = "Update";
btnUpdate.Click +=
new System.EventHandler (this.btnUpdate_Click);
txtContactName.Location =
            new System.Drawing.Point(136, 274);
txtContactName.TabIndex = 11;
txtContactName.Size = new System.Drawing.Size (160, 20);
txtZip.Location = new System.Drawing.Point (384, 274);
txtZip.TabIndex = 17;
txtZip.Size = new System.Drawing.Size (160, 20);
btnDelete.Location = new System.Drawing.Point(472, 120);
btnDelete.Size = new System.Drawing.Size(75, 23);
btnDelete.TabIndex = 2;
btnDelete.Text = "Delete";
btnDelete.Click +=
new System.EventHandler (this.btnDelete_Click);
txtContactTitle.Location =
            new System.Drawing.Point(136, 303);
txtContactTitle.TabIndex = 19;
txtContactTitle.Size = new System.Drawing.Size(160, 20);
txtAddress.Location = new System.Drawing.Point(384, 216);
txtAddress.TabIndex = 13;
txtAddress.Size = new System.Drawing.Size (160, 20);
txtCompanyName.Location= new System.Drawing.Point (136, 245);
txtCompanyName.TabIndex = 9;
```

```
txtCompanyName.Size = new System.Drawing.Size (160, 20);
label5.Location = new System.Drawing.Point (320, 252);
label5.Text = "City";
label5.Size = new System.Drawing.Size (48, 16);
label5.TabIndex = 14;
label6.Location = new System.Drawing.Point (320, 284);
label6.Text = "Zip";
label6.Size = new System.Drawing.Size (40, 16);
label6.TabIndex = 16;
label7.Location = new System.Drawing.Point (40, 312);
label7.Text = "Contact Title";
label7.Size = new System.Drawing.Size (88, 16);
label7.TabIndex = 18;
label8.Location = new System.Drawing.Point (320, 312);
label8.Text = "Phone";
label8.Size = new System.Drawing.Size (56, 16);
label8.TabIndex = 20;
label9.Location = new System.Drawing.Point (120, 120);
label9.Text = "New Customer Name:";
label9.Size = new System.Drawing.Size (120, 24);
label9.TabIndex = 22;
label4.Location = new System.Drawing.Point (320, 224);
label4.Text = "Address";
label4.Size = new System.Drawing.Size (56, 16);
label4.TabIndex = 12;
lbCustomers.Location = new System.Drawing.Point(32, 16);
lbCustomers.Size = new System.Drawing.Size (512, 95);
lbCustomers.TabIndex = 3;
txtPhone.Location = new System.Drawing.Point (384, 303);
txtPhone.TabIndex = 21;
txtPhone.Size = new System.Drawing.Size (160, 20);
btnNew.Location = new System.Drawing.Point (472, 336);
btnNew.Size = new System.Drawing.Size (75, 23);
btnNew.TabIndex = 5;
btnNew.Text = "New";
btnNew.Click += new System.EventHandler(this.btnNew_Click);
label1.Location = new System.Drawing.Point (40, 224);
label1.Text = "Company ID";
label1.Size = new System.Drawing.Size (88, 16);
label1.TabIndex = 6;
label2.Location = new System.Drawing.Point (40, 252);
label2.Text = "Company Name";
label2.Size = new System.Drawing.Size (88, 16);
label2.TabIndex = 8;
label3.Location = new System.Drawing.Point (40, 284);
label3.Text = "Contact Name";
label3.Size = new System.Drawing.Size (88, 16);
label3.TabIndex = 10;
this.Text = "Customers Update Form";
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size (584, 421);
this.Controls.Add (this.label9);
this.Controls.Add (this.txtPhone);
this.Controls.Add (this.label8);
this.Controls.Add (this.txtContactTitle);
this.Controls.Add (this.label7);
this.Controls.Add (this.txtZip);
this.Controls.Add (this.label6);
```

```
    this.Controls.Add (this.txtCity);
    this.Controls.Add (this.label5);
    this.Controls.Add (this.txtAddress);
    this.Controls.Add (this.label4);
    this.Controls.Add (this.txtContactName);
    this.Controls.Add (this.label3);
    this.Controls.Add (this.txtCompanyName);
    this.Controls.Add (this.label2);
    this.Controls.Add (this.txtCompanyID);
    this.Controls.Add (this.label1);
    this.Controls.Add (this.btnNew);
    this.Controls.Add (this.txtCustomerName);
    this.Controls.Add (this.btnUpdate);
    this.Controls.Add (this.lblMessage);
    this.Controls.Add (this.btnDelete);
    this.Controls.Add (this.lbCustomers);
}

// Quản lý sự kiện nhấn nút tạo mới (New)
protected void btnNew_Click( object sender, System.EventArgs e)
{
    // tạo một dòng mới
    DataRow newRow = dataTable.NewRow( );
    newRow["CustomerID"] = txtCompanyID.Text;
    newRow["CompanyName"] = txtCompanyName.Text;
    newRow["ContactName"] = txtContactName.Text;
    newRow["ContactTitle"] = txtContactTitle.Text;
    newRow["Address"] = txtAddress.Text;
    newRow["City"] = txtCity.Text;
    newRow["PostalCode"] = txtZip.Text;
    newRow["Phone"] = txtPhone.Text;

    // thêm một dòng mới vào bảng
    dataTable.Rows.Add(newRow);

    // cập nhật vào cơ sở dữ liệu
    DataAdapter.Update(DataSet,"Customers");

    // thông báo cho người dùng biết câu truy vấn thay đổi
    lblMessage.Text = DataAdapter.UpdateCommand.CommandText;
    Application.DoEvents( );
    DataSet.AcceptChanges( );

    // hiển thị lại dữ liệu cho điều khiển ListBox
    PopulateLB( );

    // Xoá trằng các TextBox
    ClearFields( );
}

// Xóa trắng các TextBox
private void ClearFields( )
{
    txtCompanyID.Text = "";
    txtCompanyName.Text = "";
    txtContactName.Text = "";
    txtContactTitle.Text = "";
    txtAddress.Text = "";
```

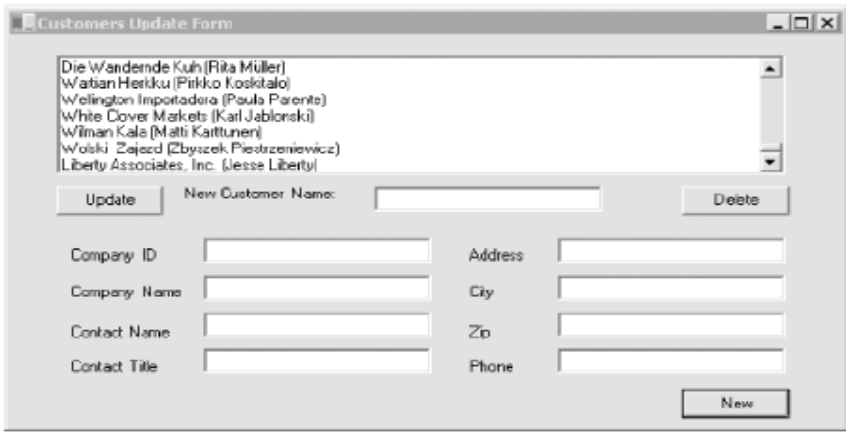
```
    txtCity.Text = "";
    txtZip.Text = "";
    txtPhone.Text = "";
}

// quản lý sự kiện nhất nút chọn cập nhật (Update)
protected void btnUpdate_Click( object sender, EventArgs e)
{
    // lấy về dòng được chọn trên ListBox
    DataRow targetRow =
              dataTable.Rows[lbCustomers.SelectedIndex];

    // thông báo cho người biết dòng cập nhật
    lblMessage.Text = "Updating " +  targetRow["CompanyName"];
    Application.DoEvents( );

    // hiệu chỉnh dòng
    targetRow.BeginEdit( );
    targetRow["CompanyName"] = txtCustomerName.Text;
    targetRow.EndEdit( );

    // lấy về các dòng thay đổi
    DataSet DataSetChanged =
    DataSet.GetChanges(DataRowState.Modified);

    // đảm bảo không có dòng nào có lỗi
    bool okayFlag = true;
    if (DataSetChanged.HasErrors)
    {
      okayFlag = false;
      string msg = "Error in row with customer ID ";

      // kiểm tra lỗi trên từng bảng
      foreach (DataTable theTable in DataSetChanged.Tables)
      {
        // nếu bảng có lỗi thì tìm lỗi trên dòng cụ thể
        if (theTable.HasErrors)
        {
          // lấy các dòng có lỗi
          DataRow[] errorRows = theTable.GetErrors( );

          // duyệt qua từng dòng có lỗi để thống báo.
          foreach (DataRow theRow in errorRows)
          {
            msg = msg + theRow["CustomerID"];
          }
        }
      }
      lblMessage.Text = msg;
    }

    // nếu không có lỗi
    if (okayFlag)
    {
      // trộn các thay đổi trong 2 DataSet thành một
      DataSet.Merge(DataSetChanged);

      // cập nhật cơ sở dữ liệu
```

```
            DataAdapter.Update(DataSet,"Customers");

            // thông báo câu truy vấn cho người dùng
            lblMessage.Text = DataAdapter.UpdateCommand.CommandText;
            Application.DoEvents( );

            // cập nhật DataSet và
            // hiển thị dữ liệu mới cho ListBox
            DataSet.AcceptChanges( );
            PopulateLB( );
          }
          else // nếu có lỗi
            DataSet.RejectChanges( );
        }

        // quản lý sự kiện xóa
        protected void btnDelete_Click( object sender, EventArgs e)
        {
          // lấy về dòng được chọn trên ListBox
          DataRow targetRow =
                      dataTable.Rows[lbCustomers.SelectedIndex];

          // chuẩn bị thông báo cho người dùng
          string msg = targetRow["CompanyName"] + " deleted. ";

          // xóa dòng được chọn
          dataTable.Rows[lbCustomers.SelectedIndex].Delete( );

          // cập nhật thay đổi cho DataSet
          DataSet.AcceptChanges( );

          // cập nhật cơ sở dữ liệu
          DataAdapter.Update(DataSet,"Customers");
          // hiển thị lại ListBox với dữ liệu thay đổi
          PopulateLB( );

          // thông báo cho người dùng biết
          lblMessage.Text = msg;
          Application.DoEvents( );
        }
        public static void Main(string[] args)
        {
          Application.Run(new ADOForm1( ));
        }
      }
    }
```

Giao diện thực thi của ứng dụng :

Hình 14-9 Cung cấp dữ liệu cho các TextBox để thêm mới một dòng

Hình 14-10 Sau khi thêm một dòng vào cuối ListBox



Chương 15 Ứng dụng Web với Web Forms

Công nghệ .NET được dùng để xây dựng các ứng dụng Web là ASP.NET, nó cung cấp hai vùng tên khá mạnh và đầy đủ phục vụ cho việc tạo các ứng dụng Web là **System.Web** và **System.Web.UI**. Trong phần này chúng ta sẽ tập trung chủ yếu vào việc dùng ngôn ngữ C# để lập trình với ASP.NET.

Bộ công cụ Web Form cũng được thiết kế để hỗ trợ mô hình phát triển nhanh (RAD). Với Web Form, ta có thể kéo thả các điều khiển trên Form thiết kế cũng như có thể viết mã trực tiếp trong tập tin .**aspx** hay .**aspx.cs**. Ứng dụng Web sẽ được triển khai trên máy chủ, còn người dùng sẽ tương tác với ứng dụng thông qua trình duyệt. .NET còn hỗ trợ ta bộ cung cụ để tạo ra các ứng dụng tuân theo mô hình n - lớp (tầng - n tier), giúp ta có thể quản lý được ứng dụng được dễ dàng hơn và nhờ thế nâng cao hiệu suất phát triển phần mềm.

1.1 Tìm hiểu về Web Forms

Web Form là bộ công cụ cho phép thực thi các ứng dụng mà các trang Web do nó tạo động ra được phân phối đến trình duyệt thông qua mạng Internet.

Với Web Forms, ta tạo ra các trang HTML với nội dung tĩnh và dùng mã C# chạy trên Server để xử lý dữ liệu tĩnh này rồi tạo ra trang Web động, gửi trang này về trình duyệt dưới mã HTML chuẩn.

Web Forms được thiết để chạy trên bất kỳ trình duyệt nào, trang HTML gửi về sẽ được gọt giũa sao cho thích hợp với phiên bản của trình duyệt. Ngoài dùng C#, ta cũng có thể dùng ngôn ngữ VB.NET để tạo ra các ứng dụng Web tương tự.

Web Forms chia giao diện người dùng thành hai phần : phần thấy trực quan (hay *UI*) và phần trang mã phía sau của UI. Quan điểm này thì tương tự với Windows Form, nhưng với Web Forms, hai phần này nằm trên hai tập tin riêng biệt. Phần giao diện UI được lưu trữ trong tập tin có phần mở rộng là .aspx, còn mã được lưu trữ trong tập tin có phần mở rộng là .aspx.cs.

Với môi trường làm việc được cung cấp bởi bộ Visual Studio .NET, tạo các ứng dụng Web đơn giản chỉ là mở Form mới, kéo thả và viết mả quản lý sự kiện thích hợp. Web Forms được tích hợp thêm một loạt các điều khiển thực thi trên Server, có thể tự kiểm tra sự hợp lệ của dữ liệu ngay trên máy khách mà ta không phải viết mã mô tả gì cả.

15.1 Các sự kiện của Web Forms

Một sự kiện (Events) được tạo ra khi người dùng nhấn chọn một Button, chọn một mục trong ListBox hay thực hiện một thao tác nào đó trên UI. Các sự kiện cũng có thể được phát sinh hệ thống bắt đầu hay kết thúc.

Phương thức đáp ứng sự kiện gọi là trình quản lý sự kiện, các trình quản lý sự kiện này được viết bằng mã C# trong trang mã (code-behind) và kết hợp với các thuộc tính của các điều khiển thuộc trang.

Trình quản lý sự kiện là một "Delegate", phương thức này sẽ trả về kiểu void, và có hai đối số. Đối số đầu tiên là thể hiện của đối tượng phát sinh ra sự kiện, đối số thứ hai là đối tượng EventArg hay một đối tượng khác được dẫn xuất từ đối tượng EventArgs. Các sự kiện này được quản lý trên Server.

15.1.1 Sự kiện PostBack và Non-PostBack

PostBack là sự kiện sẽ khiến Form được gửi về Server ngay lập tức, chẳng hạn sự kiện đệ trình một Form với phương thức Post. Đối lập với PostBack là Non-PostBack, sự kiện này không gửi Form nên Server mà nó lưu sự kiện trên vùng nhớ Cache cho tới khi có một sự kiện PostBack nữa xảy ra. Khi một điều khiển có thuộc tính AutoPostBack là `true` thì sự kiện PostBack sẽ có tác dụng trên điều khiển đó : mặc nhiên thuộc tính AutoPostBach của điều khiển DropDownList là `false`,ta phải đặt lại là `true` thì sự kiện chọn một mục khác trong DropDownList này mới có tác dụng.

15.1.2 Trạng thái của ứng dụng Web (State)

Trạng thái của ứng dụng Web là giá trị hiện hành của các điều khiển và mọi biến trong phiên làm việc hiện hành của người dùng. Web là môi trường không trạng thái, nghĩa là mỗi sự kiện Post lên Server đều làm mất đi mọi thông tin về phiên làm việc trước đó. Tuy nhiên ASP.NET đã cung cấp cơ chế hỗ trợ việc duy trì trạng thái về phiên của người dùng.

Bất kỳ trang nào khi được gửi lên máy chủ Server đều được máy chủ tổng hợp thông tin và tái tạo lại sau đó mới gửi xuống trình duyệt cho máy khách. ASP.NET cung cấp một cơ chế giúp duy trì trạng thái của các điều khiển phía máy chủ (Server Control) một cách tự động. Vì thế nếu ta cung cấp cho người dùng một danh sách dữ liệu ListBox, và người dùng thực hiện việc chọn lựa trên ListBox này, sự kiện chọn lựa này sẽ vẫn được duy trì sau khi trang được gửi lên máy chủ và gửi về cho trình duyệt cho máy khách.

15.1.3 Chu trình sống của một Web-Form

Khi có yêu cầu một trang Web trên máy chủ Web sẽ tạo ra một chuỗi các sự kiện ở máy chủ đó, từ lúc bắt đầu cho đến lúc kết thúc một yêu cầu sẽ hình thành một chu trình sống (Life-Cycle) cho trang Web và các thành phần thuộc nó. Khi một trang

Web được yêu cầu, máy chủ sẽ tiến hành mở (Load) nó và khi hoàn tất yêu cầu máy chủ sẽ đóng trang này lại, kết xuất của yêu cầu này là một trang HTML tương ứng sẽ được gửi về cho trình duyệt. Dưới đây sẽ liệt kê một số sự kiện, ta có thể bắt các sự kiện để xử lý thích hợp hay bỏ qua để ASP.NET xử lý mặc định.

Khởi tạo (*Initialize*) Là sự kiện đầu tiên trong chu trình sống của trang, ta có thể khởi bất kỳ các thông số cho trang hay các điều khiển thuộc trang.

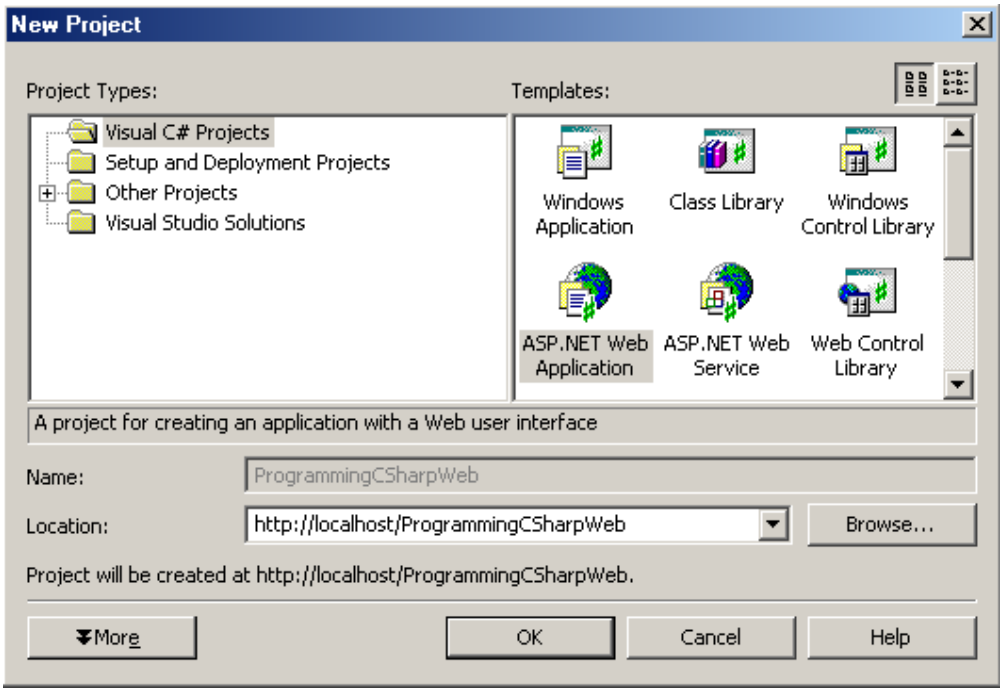
Mở trạng thái vùng quan sát (*Load View State*) Được gọi khi thuộc tính **ViewState** của điều khiển được công bố hay gọ. Các giá trị trong ViewState sẽ được lưu trữ trong một biến ẩn (Hidden Field), ta có thể lấy giá trị này thông qua hàm LoadViewState() hay lấy trực tiếp.

Kết thúc (Dispose) Ta có thể dùng sự kiện này để giải phóng bất kỳ tài nguyên nguyên nào : bộ nhớ hay hủy bỏ các kết nối đến cơ sở dữ liệu.

15.2 Hiển thị chuỗi lên trang

Đầu tiên ta cần chạy Visual Studio .NET, sau đó tạo một dự án mới kiểu Web Application, ngôn ngữ chọn là C# và ứng dụng sẽ có tên là *ProgrammingCSharpWeb*. Url mặc nhiên của ứng dụng sẽ có tên là *http://localhost/ ProgrammingCSharpWeb.*

Hình 15-1 Cửa sổ tạo ứng dụng Web mới



Visual Studio .NET sẽ đặt hầu hết các tập tin nó tạo ra cho ứng dụng trong thư mục Web mặc định trên máy người dùng, ví dụ : *D:\Inetpub\wwwroot\ProgrammingCSharpWeb*. Trong .NET, một giải pháp

(Solution) có một hay hiều dự án (Project), mỗi dự án sẽ tạo ra một thư viện liên kết động (DLL) hay tập tin thực thi (EXE). Để có thể chạy được ứng dụng Web Form, ta cần phải cài đặt IIS và FrontPage Server Extension trên máy tính.

Khi ứng dụng Web Form được tạo, .NET tạo sẵn một số tập tin và một trang Web có tên mặc định là WebForm1.aspx chỉ chứa mã HTML và **WebForm1.cs** chứa mã quản lý trang. Trang mã **.cs** không nằm trong cửa sổ Solution Explorer, để hiển thị nó ta chọn Project\Show All Files, ta chỉ cần nhấn đúp chuột trái trên trang Web là cửa sổ soạn thảo mã (Editor) sẽ hiện nên, cho phép ta viết mã quản lý trang. Để chuyển từ cửa số thiết kế kéo thả sang cửa sổ mã HTML của trang, ta chọn hai Tab ở góc bên trái phía dưới màn hình.

Đặt tên lại cho trang Web bằng cách nhấn chuột phải lên trang và chọn mục **Rename** để đổi tên trang thành **HelloWeb.aspx**, .NET cũng sẽ tự động đổi tên trang mã của trang thành **HelloWeb.cs**. NET cũng tạo ra một số mã HTML cho trang :

Hình 15-2

```
4  <%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
5      Inherits="ProgrammingCSharpWeb.WebForm1" %>
6  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
7  <html>
8      <head>
9          <title>WebForm1</title>
10         <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
11         <meta name="CODE_LANGUAGE" Content="C#">
12         <meta name="vs_defaultClientScript"content="JavaScript">
13         <meta name="vs_targetSchema" content="http://schemas.microsoft.com/intelli
14     </head>
15     <body MS_POSITIONING="GridLayout">
16         <form id="Form1" method="post" runat="server">
17         </form>
18     </body>
19 </html>
```

.NET đã phát sinh ra một số mã ASP.NET :

```
<%@ Page language="c#"
Codebehind="HelloWeb.cs"
AutoEventWireup="false"
Inherits="ProgrammingCSharpWeb.WebForm1" %>
```

Thuộc tính **language** chỉ ra ngôn ngữ lập trình được dùng trong trang mã để quản lý trang, ở đây là C#. **Codebehide** xác định trang mã quản lý có tên **HelloWeb.cs** và thuộc tính **Inherits** chỉ trang Web được thừa kế từ lớp **WebForm1** được viết trong **HelloWeb.cs :**

```
public class WebForm1 : System.Web.UI.Page
```

Ta thấy trang này được thừa kế từ lớp **System.Web.UI.Page**, lớp này do ASP.NET cung cấp, xác định các thuộc tính, phương thức và các sự kiện chung cho các trang phía máy chủ. Mã HTML phát sinh định dạng thuộc tính của Form :

```
<form id="Form1" method="post" runat="server">
```

Thuộc tính **id** làm định danh cho Form, thuộc tính **method** có giá trị là "**POST**" nghĩa là Form sẽ được gởi lên máy chủ ngay lập tức khi nhận một sự kiện do người dùng phát ra (như sự kiện nhấn nút) và cờ **IsPostBack** trên máy chủ khi đó sẽ có giá trị là `true.` Biến cờ này có giá trị là `false` nếu Form được đệ trình với phương thức "**GET**" hay lần đầu tiên trang được gọi. Bất kỳ điều khiển nào hay Form có thuộc tính **runat="server"** thì điều khiển hay Form này sẽ được xử lý bởi ASP.NET Framework trên máy chủ. Thuộc tính MS_POSITIONING = "GridLayout" trong thẻ <Body>, cho biết cách bố trí các điều khiển trên Form theo dạng lưới, ngoài ra ta còn có thể bố trí các điều khiển trôi lối trên trang, bằng cách gán thuộc tính MS_POSITIONING thành "FlowLayout".

Hiện giờ Form của ta là trống, để hiển thị một chuỗi gì đó lên màn hình, ta gõ dòng mã sau trong thẻ **<body>** :

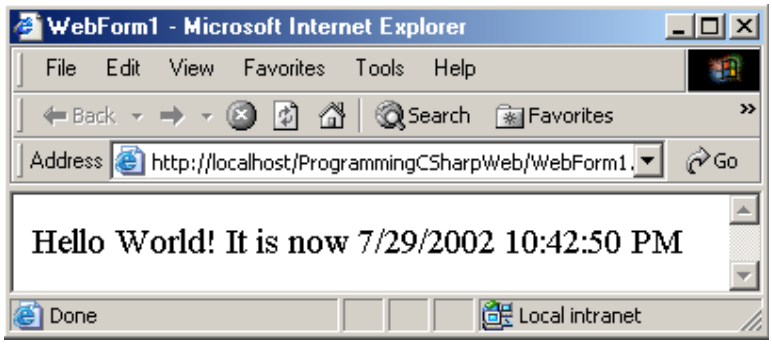
```
Hello World! It is now <% = DateTime.Now.ToString( ) %>
```

Giống với ASP, phần nằm trong dấu **<% %>** được xem như là mã quản lý cho trang, ở đây là mã C#. Dấu = chỉ ra một giá trị nhận được từ một biến hay một đối tượng nào đó, ta cũng có thể viết mã trên lại như sau với cùng chức năng :

```
Hello World! It is now
<% Response.Write(DateTime.Now.ToString( )); %>
```

Thực thi trang này (Ctrl-F5), kết quả sẽ hiện trên trình duyệt như sau :

Hình 15-3 Hiển thị chuỗi thời gian



Để thêm các điều khiển cho trang, hoặc là ta có thể viết mã trong của sổ HTML hoặc là kéo thả các điều khiển trên bộ công của Web Form vào cửa sổ thiết kế trang. ASP.NET sẽ tự động phát sinh ra kết quả từ mã HTML thành các điều khiển cũng như từ các điều khiển trên trang thiết thành mã HTML tương ứng. Ví dụ, kéo hai RadioButton vào trang và gán cùng một giá trị nào đó cho thuộc tính **GroupName** của cả hai điều khiển, thuộc tính này sẽ làm cho các nút chọn loại trừ lẫn nhau. Mã HTML của trang trong thẻ <Form> do ASP.NET phát sinh sẽ như sau :

Hình 15-4

```
<form id="Form2" method="post" runat="server">
<asp:RadioButton GroupName = "Radio" id="Radiobutton3"
    style="Z-INDEX: 101; LEFT: 328px; POSITION: absolute; TOP: 264px"
    runat="server"></asp:RadioButton>
<asp:RadioButton GroupName = "Radio" id="Radiobutton4"
    style="Z-INDEX: 102; LEFT: 328px; POSITION: absolute; TOP: 360px"
    runat="server"></asp:RadioButton>
</form>
```

Các điều khiển của ASP.NET, có thêm chữ "asp:" phía trước tên của điều khiển đó, được thiết kế mang tính hướng đối tượng nhiều hơn.

```
<asp:RadioButton>
<asp:CheckBox>
<asp:Button>
<asp:TextBox rows="1">
<asp:TextBox rows="5">
```

Ngoài các điều khiển của ASP.NET, các điều khiển HTML chuẩn cũng được ASP.NET hỗ trợ. Tuy nhiên các điều khiển không tạo sự dễ đọc trong mã nguồn do tính đối tượng trên chúng không rõ ràng, các điều khiển HTML chuẩn ứng với năm điều khiển trên là :

```
<input type = "radio">
<input type="checkbox">
<input type="button">
<input type="text">
<textarea>
```

15.3 Điều khiển xác nhận hợp

ASP.NET cung cấp một tập các điều khiển xác nhận hợp lệ dữ liệu nhập phía máy chủ cũng như ở dưới trình duyệt của máy khách. Tuy nhiên việc xác nhận hợp lệ dưới máy khách chỉ là một chọn lựa, ta có thể tắt nó đi, nhưng việc xác nhận hợp lệ trên máy chủ thông qua các điều khiển này là bắt buộc, nhằm phòng ngừa một số trường hợp dữ liệu nhập là giả mạo. Việc kiểm tra hợp lệ của mã trên máy chủ là đề phòng các trường hợp. Một số loại xác nhận hợp lệ : dữ liệu không được rỗng, thỏa một định dạng dữ liệu nào đó …

Các điều khiển xác nhận hợp lệ phải được gắn liền với một điều khiển nhận dữ liệu HTML nào đó, các điều khiển nhập được liệt trong bảng sau :

Bảng 15-1 Các điều khiển nhập HTML dùng để xác nhận hợp lệ

Các điều khiển nhập	Thuộc tính xác nhận hợp lệ

Các điều khiển nhập	Thuộc tính xác nhận hợp lệ
HtmlInputText	Value
HtmlTextArea	Value
HtmlSelect	Value
HtmlInputFile	Value
TextBox	Text
ListBox	SelectedItem.Value
DropDownList	SelectedItem.Value
RadioButtonList	SelectedItem.Value

Ứng với một điều khiển nhập HTML, ta có thể gắn nhiều điều khiển xác nhận hợp lệ cho nó, bảng dưới đây sẽ liệt kê các điều khiển nhập hiện có :

Bảng 15-2 Các điều khiển xác nhận hợp lệ

Điều khiển	Mục đích
CompareValidator	So sánh các giá trị của hai điều khiển để xem có bằng nhau hay không
CustomValidator	Gọi một hàm do người dùng định nghĩa để thi hành việc kiểm tra
RangeValidator	Kiểm tra xem một mục có nằm trong một miền đã cho hay không
RegularExpressionvalidator	Kiểm tra người dùng có sửa đổi một mục (mà giá trị của nó khác với một giá trị đã chỉ định ban đầu, ngầm định giá trị ban đầu là một chuỗi trống) hay không
ValidationSummary	Thông báo sự hợp lệ trên các điều khiển

15.4 Một số ví dụ mẫu minh họa

Một cách thuận tiện nhất để học một công nghệ mới chính là dựa vào các ví dụ, vì vậy trong phần này chúng ta sẽ khảo sát một vài ví dụ để minh họa cho phần lý thuyết của chúng ta. Như ta đã biết, ta có thể viết mã quản lý theo hai cách : hoặc là viết trong tập tin **.cs** hoặc là viết trực tiếp trong trang chứa mã HTML. Ở đây để dễ tập trung vào các ví dụ của chúng ta, ta sẽ viết mã quản lý trực tiếp trên trang HTML.

15.4.1 Kết buộc dữ liệu

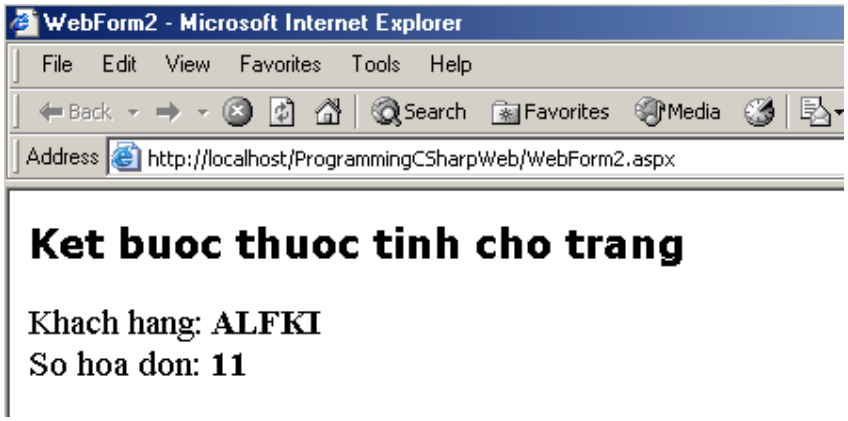
15.4.1.1 Không thông qua thuộc tính DataSource

Ứng dụng của chúng ta đơn giản chỉ hiện lên trang tên khách hàng và số hóa đơn bằng cách dùng hàm DataBind(). Hàm này sẽ kết buộc dữ liệu của mọi thuộc tính hay của bất kỳ đối tượng.

```
<html>
<head>
// mã quản lý C# sẽ được viết trong thẻ <script> này
```

```
    <script language="C#" runat="server">

    // trang sẽ gọi hàm này đầu tiên, ta sẽ thực hiện kết buộc
    // trực tiếp trong hàm này
    void Page_Load(Object sender, EventArgs e) {
              Page.DataBind();
    }

    // lấy giá trị của thuộc tính thông qua thuộc tính // get
    string custID{
       get {
          return "ALFKI";
       }
    }
    int orderCount{
       get {
          return 11;
       }
     }
    </script>

</head>
<body>
<h3><font face="Verdana"> Ket buoc khong dung DataSource
</font></h3>
     <form runat=server>
         Khach hang: <b><%# custID %></b><br>
         So hoa don: <b><%# orderCount %></b>
     </form>
</body>
</html>
```

Hình 15-5 Giao diện của ví dụ



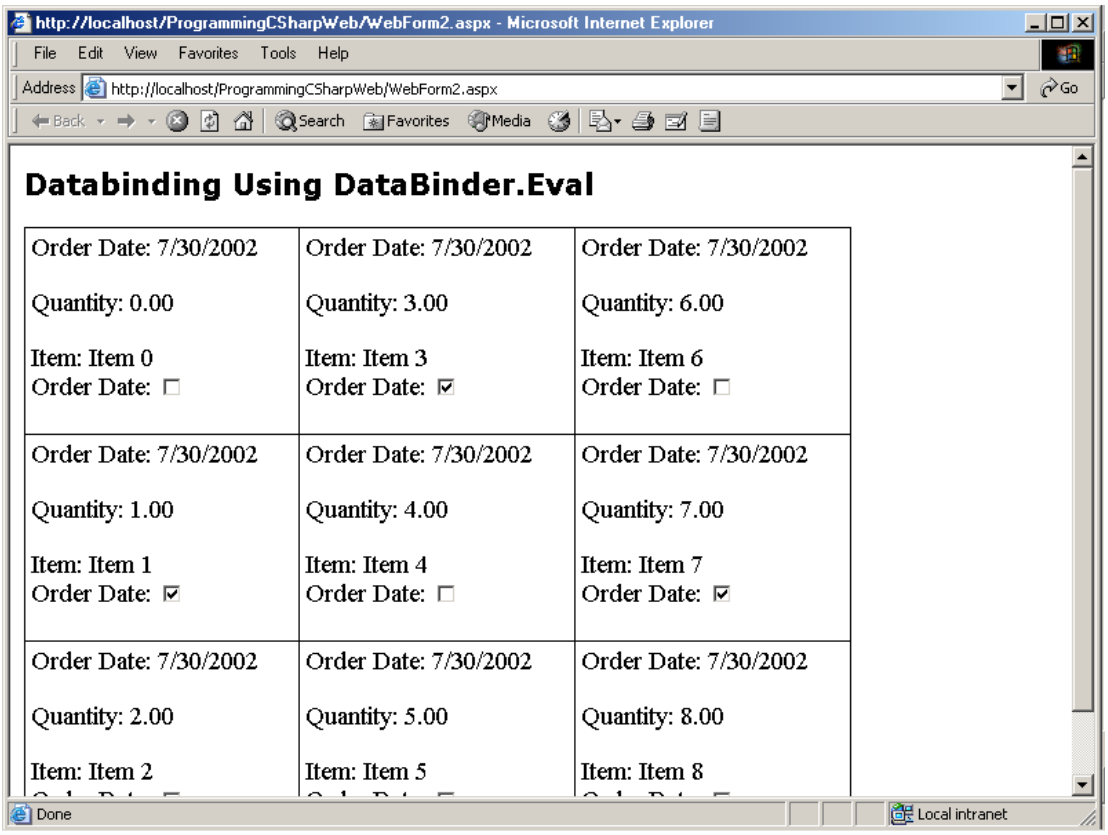
15.4.1.2 Điều khiển DataList với DataSource

Trong ví dụ này, ta sẽ dùng thuộc tính DataSource của điều khiển <asp:DataList> để kết buộc dữ liệu, ta sẽ cung cấp cho thuộc tính DataSource này một bảng dữ liệu giả, sau đó dùng hàm DataBinder.Eval()để kết buộc dữ liệu trong DataSource theo một định dạng (Format) thích hợp mong muốn. Dữ liệu sẽ được hiển thị lên màn hình dưới dạng một bảng các hóa đơn sau khi ta gọi hàm DataBind().

```
//Không gian tên chứa các đối tượng của ADO.NET
<%@ Import namespace="System.Data" %>
<html>
  <head>
  <script language="C#" runat="server">
    void Page_Load(Object sender, EventArgs e) {

       // nếu trang được gọi lần đầu tiên
       if (!Page.IsPostBack) {

       // tạo ra một bảng dữ liệu mới gồm 4 cột , sau đó thêm dữ
       // liệu giả cho bảng
          DataTable dt = new DataTable();
          DataRow dr;

       // thêm 4 cột DataColumn vào bảng, mỗi cột có các
       // kiểu dữ liệu riêng
dt.Columns.Add(new DataColumn("IntegerValue", typeof(Int32)));
dt.Columns.Add(new DataColumn("StringValue", typeof(string)));
dt.Columns.Add(new DataColumn("DateTimeValue", typeof(DateTime)));
dt.Columns.Add(new DataColumn("BoolValue", typeof(bool)));

       // thêm 9 dòng dữ liệu cho bảng bằng cách tạo ra
       // một dòng mới dùng phương thức NewRow() của đối
       // tượng DataTable, sau đó gán dữ liệu giả cho
       // dòng này và thêm dòng dữ liệu này vào bảng
       for (int i = 0; i < 9; i++) {
          dr = dt.NewRow();
          dr[0] = i;
          dr[1] = "Item " + i.ToString();
          dr[2] = DateTime.Now;
          dr[3] = (i % 2 != 0) ? true : false;
          dt.Rows.Add(dr);
       }

     // gán bảng dữ liệu cho thuộc tính DataSource của điều
     // khiển DataList, sau đó thực hiện kết buộc bằng hàm
     // DataBind()
     dataList1.DataSource = new DataView(dt);
     dataList1.DataBind();
   }
  }
  </script>
  </head>

  <body>

<h3><font face="Verdana">Ket buoc du lieu dung DataSource thong qua
ham DataBind.Eval() </font></h3>
<form runat=server>

// điều khiển danh sách cho phép ta kết buộc dữ liệu khá
// linh động, ta chỉ cần cung cấp cho nó một DataSource
// thích hợp, sau đó gọi hàm DataBind()để hiển thị dữ liệu // lên
trang
<asp:DataList id="dataList1" runat="server"
                RepeatColumns="3"
                Width="80%"
```

```
                BorderColor="black"
                BorderWidth="1"
                GridLines="Both"
                CellPadding="4"
                CellSpacing="0">
// đây là một thuộc tính của lưới, khi gọi hàm
// DabaBind(), dữ liệu trong DataSource sẽ được trích ra
// (nếu là danh các đối tượng thì mỗi lần trích sẽ lấy ra
// một phần tử kiểu đối tượng đó, sau đó dùng hàm
// DataBinder.Eval()để gán giá trị, còn nếu là một bảng
// dữ liệu thì mỗi lần kết buộc sẽ lấy ra một dòng dữ
// liệu, hàm DataBind.Eval() sẽ lấy dữ liệu của từng
// trường) để kết buộc lên trang. Nó sẽ lặp lại thao tác
// này cho tới khi dữ liệu được kết buộc hết.
<ItemTemplate>
  //lấy dữ liệu trên cột đầu tiên để kết buộc
  Ngay hoa don: <%# DataBinder.Eval(Container.DataItem,
"DateTimeValue", "{0:d}") %>

  //lấy dữ liệu trên cốt thứ 2
  So luong: <%# DataBinder.Eval(Container.DataItem, "IntegerValue",
"{0:N2}") %>

  //cột thứ 3
Muc: <%# DataBinder.Eval(Container.DataItem, "StringValue") %>
  //cột thứ 4
Ngay hoa don: <asp:CheckBox id=chk1 Checked='<%#
(bool)DataBinder.Eval(Container.DataItem, "BoolValue") %>'
runat=server/><p>
   </ItemTemplate>
</asp:Datalist>
</form>
</body>
</html>
```

Hình 15-6 Giao diện của ví dụ sau khi thực thi



15.4.1.3 Kết buộc với điều khiển DataGrid

Trong ví trước, ta đã tìm hiểu sơ qua về cách đẩy dữ liệu vào thuộc tính DataSource của điều khiển DataList thông qua hàm kết buộc DataBind().Ví dụ này chúng ta sẽ khảo sát thêm về cách kết buộc trên điều khiển lưới DataGrid và cách dùng điều khiển xác nhận hợp lệ trên dữ liệu. Khi ứng dụng chạy sẽ hiển thị một bảng dữ liệu lên trang, người dùng có thể hiệu chỉnh bất kỳ một dòng nào trên bảng dữ liệu bằng cách nhấn vào chuỗi lệnh hiệu chỉnh (Edit) trên lưới, gõ vào các dữ liệu cần hiệu chỉnh, khi muốn hủy bỏ thao tác hiệu chỉnh ta nhấn chọn chuỗi bỏ qua (Cancel). Để tập trung vào mục đích của ví dụ, chúng ta sẽ dùng bảng dữ liệu giả, cách làm sẽ tương tự trên bảng dữ liệu lấy ra từ cơ sở dữ liệu. Sau đây là mã của ví dụ :

```
//không gian tên cần thiết để truy cập đến các đối tượng ADO.NET
<%@ Import Namespace="System.Data" %>
<html>

<script language="C#" runat="server">

   //khai báo đối tượng bảng và khung nhìn
   DataTable Cart;
   DataView CartView;

// lấy dữ liệu trong Session, nếu không có thì ta sẽ tạo ra một
// bảng dữ liệu khác
```

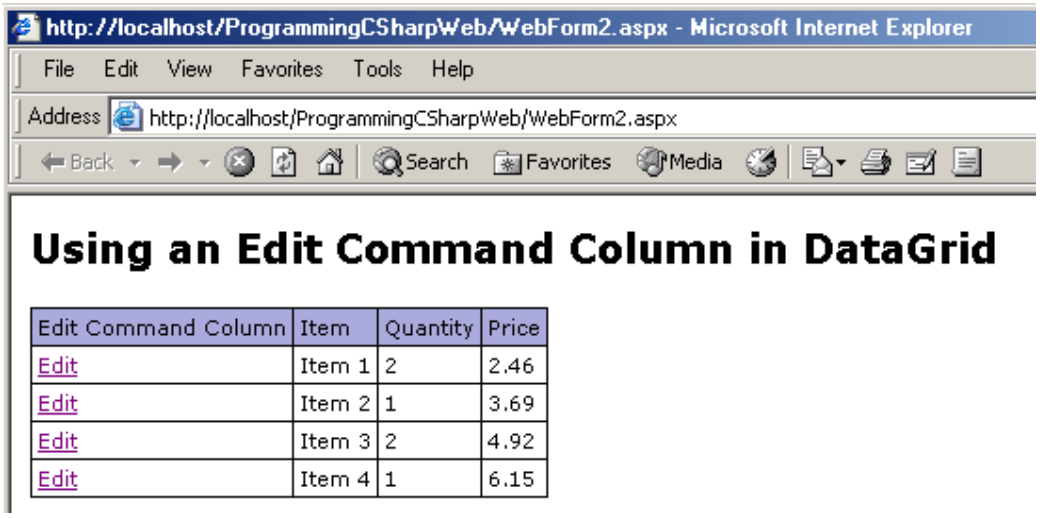
```
  void Page_Load(Object sender, EventArgs e) {
    if (Session["DG6_ShoppingCart"] == null)     {
      Cart = new DataTable();

      //bảng sẽ có 3 cột đều có kiểu là chuỗi
        Cart.Columns.Add(new DataColumn("Qty", typeof(string)));
        Cart.Columns.Add(new DataColumn("Item", typeof(string)));
        Cart.Columns.Add(new DataColumn("Price", typeof(string)));

      //đẩy định danh của bảng vào phiên làm việc hiện thời
        Session["DG6_ShoppingCart"] = Cart;

        // tạo dữ liệu mẫu cho bảng
        for (int i=1; i<5; i++) {
         DataRow dr = Cart.NewRow();
           dr[0] = ((int)(i%2)+1).ToString();
           dr[1] = "Item " + i.ToString();
           dr[2] = ((double)(1.23 * (i+1))).ToString();
           Cart.Rows.Add(dr);
      }
    }
     else {
      //nếu bảng đã có sẵn trong Session, ta sẽ lấy ra dùng
      Cart = (DataTable)Session["DG6_ShoppingCart"];
    }

// tạo ra khung nhìn cho bảng, sau đó sắp xếp khung nhìn theo // cột
Item
      CartView = new DataView(Cart);
      CartView.Sort = "Item";

// nếu trang được gọi lần đầu tiên thì kết buộc dữ liệu thông // qua
hàm BindGrid()của ta
      if (!IsPostBack) {
       BindGrid();
    }
  }

// sự kiện nhấn chuỗi hiệu chỉnh (Edit) trên lưới, ta sẽ lấy chỉ //
mục của dòng cần hiệu chỉnh thông qua đối tượng
// DataGridCommandEventArgs, sau đó truyền chỉ mục này cho điều //
khiển lưới của ta và gọi hàm kết buộc của ta để đẩy dữ liệu
// lên lưới
public void MyDataGrid_Edit(Object sender, DataGridCommandEventArgs
e) {
    MyDataGrid.EditItemIndex = (int)e.Item.ItemIndex;
     BindGrid();
  }

  //sự kiện nhấn bỏ qua trên lưới (Cancel)
public void MyDataGrid_Cancel(Object sender,
DataGridCommandEventArgs e) {
    MyDataGrid.EditItemIndex = -1;
     BindGrid();
  }

//sau khi hiệu chỉnh dữ liệu, người dùng tiến hành cập nhật public
void MyDataGrid_Update(Object sender, DataGridCommandEventArgs e) {
```

```
      // lấy dữ liệu trên TextBox
      string item = e.Item.Cells[1].Text;
       string qty = ((TextBox)e.Item.Cells[2].Controls[0]).Text;
       string price = ((TextBox)e.Item.Cells[3].Controls[0]).Text;

// Ở đây, do chúng ta dùng dữ liệu giả lưu trên bộ nhớ chính, // nếu
dùng cơ sở dữ liệu thì chúng ta sẽ tiến hành hiệu chỉnh // trực tiếp
trong cơ sở dữ liệu bằng các câu truy vấn :
// UPDATE, SELECT, DELETE

      //xóa dòng cũ
       CartView.RowFilter = "Item='"+item+"'";
       if (CartView.Count > 0) {
        CartView.Delete(0);
      }
       CartView.RowFilter = "";

       //tạo dòng mới và thêm vào bảng
       DataRow dr = Cart.NewRow();
       dr[0] = qty;
       dr[1] = item;
       dr[2] = price;
       Cart.Rows.Add(dr);
       MyDataGrid.EditItemIndex = -1;
       BindGrid();
   }

   //kết buộc dữ liệu thông qua thuộc tính DataSource của lưới
   public void BindGrid() {
     MyDataGrid.DataSource = CartView;
     MyDataGrid.DataBind();
    }

</script>

<body style="font: 10pt verdana">

   <form runat="server">
<h3><font face="Verdana">Using an Edit Command Column in
DataGrid</font></h3>

      //Khai báo các thông số cho lưới, các sự kiện trên lưới :
        OnEditCommand : khi người dùng nhấn chuỗi hiệu chỉnh (Edit)
        OnCancelCommand : nhấn chuỗi bỏ qua hiệu chỉnh (Cancel)
        OnUpdateCommand : nhấn chuỗi cập nhật hiệu chỉnh (Update)
      <asp:DataGrid id="MyDataGrid" runat="server"
          BorderColor="black"
          BorderWidth="1"
          CellPadding="3"
          Font-Name="Verdana"
          Font-Size="8pt"
          HeaderStyle-BackColor="#aaaadd"
          OnEditCommand="MyDataGrid_Edit"
          OnCancelCommand="MyDataGrid_Cancel"
          OnUpdateCommand="MyDataGrid_Update"
          AutoGenerateColumns="false"
          >
```

```
// các thông số hiệu chỉnh trên cột, ở đây ta chỉ cho người
// dùng hiệu chỉnh trên cột số lượng và giá hóa đơn
      <Columns>
       <asp:EditCommandColumn
          EditText="Edit"
             CancelText="Cancel"
             UpdateText="Update"
             ItemStyle-Wrap="false"
             HeaderText="Edit Command Column"
             HeaderStyle-Wrap="false"
          />
<asp:BoundColumn HeaderText="Item" ReadOnly="true"
DataField="Item"/>
             <asp:BoundColumn HeaderText="Quantity" DataField="Qty"/>
             <asp:BoundColumn HeaderText="Price" DataField="Price"/>
     </Columns>
    </asp:DataGrid>

   </form>
   </body>
</html>
```

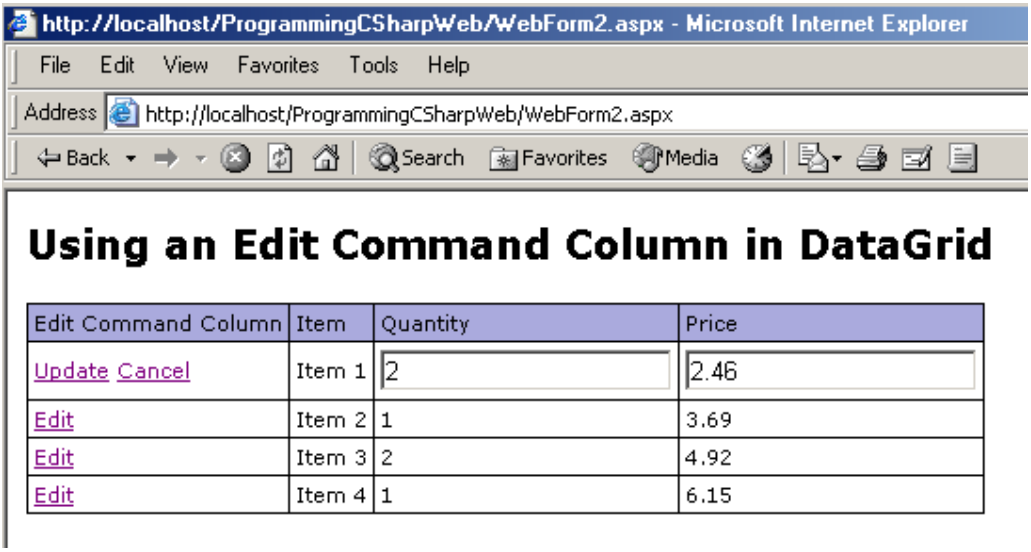
Giao diện của ví dụ khi chạy :

Hình 15-7 Hiệu chỉnh trực tiếp trên lưới dữ liệu



Sau khi người dùng chọn nút Edit trên lưới, màn hình ứng dụng sẽ như sau :

Hình 15-8 Người dùng sau khi nhấn chuỗi Edit



15.4.2 Điều khiển xác nhận hợp lệ

Việc xác nhận hợp lệ là cần thiết với các ứng dụng cần yêu cầu nhập liệu, việc đưa ra các điều khiển có khả năng xác nhận hợp lệ trực tiếp dưới máy khách lẫn ở trên máy chủ, đây có thể là một tính năng mới của ASP.NET, ta không cần phải viết mã kiểm tra gì cả, mã kiểm tra dưới trình duyệt (chẳng hạn như Java Script) sẽ được ASP.NET tự động phát sinh. Để gắn một điều khiển bắt lỗi vào một điều khiển cần bắt lỗi ta chỉ cần gán thuộc tính ControlToValidate của điều khiển bắt lỗi bằng giá trị định danh **id** của điều khiển cần bắt lỗi, ví dụ : Để bắt lỗi điều khiển TextBox không được trống, ta viết má như sau :

```
      //điều khiển cần bắt lỗi
<ASP:TextBox id=TextBox1 runat=server />

//điều khiển bắt lỗi hộp nhập liệu TextBox1
     <asp:RequiredFieldValidator id="RequiredFieldValidator2"
   ControlToValidate="TextBox1"
    ErrorMessage="Card Number. "
    Display="Static"
    Width="100%" runat=server>
    *
    </asp:RequiredFieldValidator>
```

Ví dụ của chúng ta sẽ cho hiển thị 2 hộp thoại DropDownList, 2 nút chọn RadioButton và một hộp thoại nhập TextBox, nếu tồn tại mục nhập nào trống khi nhấn nút xác nhận *Validate*, thì các điều khiển xác nhận hợp lệ sẽ hiển thị lỗi tương ứng. Thông điệp lỗi có thể được hiển thị theo ba cách khác nhau : liệt kê theo danh sách (List), liệt kê trên cùng một dòng (Single Paragraph), liệt kê danh sách với dấu chấm tròn ở đầu (Bullet List). Mã hoàn chỉnh của ví dụ được liệt kê như sau :

```
// không cho phép điều khiển xác nhận hợp lệ dưới máy khách bằng
// cách gán thuộc tính  clienttarget = downlevel
<%@ Page clienttarget=downlevel %>
```

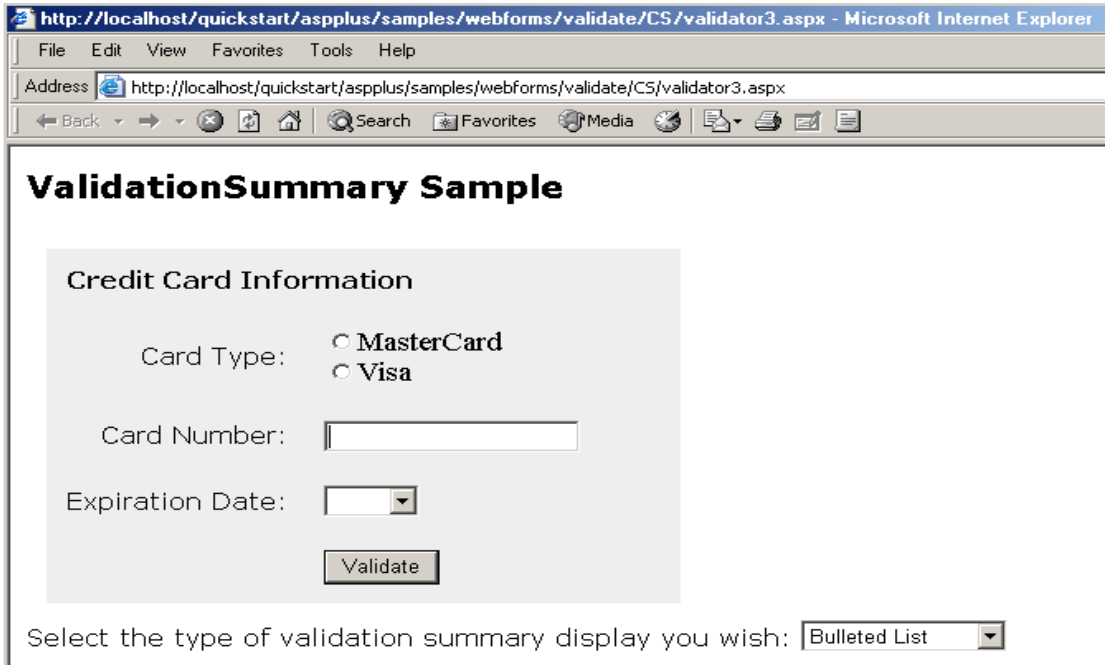
```
<html>
<head>
  <script language="C#" runat=server>

// thay đổi chế độ hiển thị lỗi bằng cách chọn 1 trong 3 mục
// trong hộp thoại ListBox
void ListFormat_SelectedIndexChanged(Object Sender, EventArgs E )
{
valSum.DisplayMode = (ValidationSummaryDisplayMode)
ListFormat.SelectedIndex;
     }
   </script>
</head>

<body>
<h3><font face="Verdana">Ví dụ về xác nhận điều khiển hợp lệ
ValidationSummary</font></h3>
<p>
  <form runat="server">
<table cellpadding=10><tr> <td>
  <table bgcolor="#eeeeee" cellpadding=10><tr><td colspan=3>

<font face=Verdana size=2><b>Credit Card
Information</b></font></td></tr>
       <tr>
         <td align=right>
<font face=Verdana size=2>Card Type:</font></td>

             <td>
// danh sách các nút chọn được bắt lỗi bởi điều //khiển xác nhận hợp
lệ RequireFieldValidator1
<ASP:RadioButtonList id=RadioButtonList1 RepeatLayout="Flow"
runat=server>
                <asp:ListItem>MasterCard</asp:ListItem>
              <asp:ListItem>Visa</asp:ListItem>
           </ASP:RadioButtonList>
         </td>

//điều khiển xác nhận hợp lệ cho các nút chọn //RadioButtonList1
         <td align=middle rowspan=1>
<asp:RequiredFieldValidator id="RequiredFieldValidator1"
                ControlToValidate="RadioButtonList1"
                  ErrorMessage="Card Type. "
                  Display="Static"
                  InitialValue="" Width="100%" runat=server>
                    *
                </asp:RequiredFieldValidator>
         </td></tr>

           <tr>
             <td align=right>
               <font face=Verdana size=2>Card Number:</font>
             </td>
             <td>
               <ASP:TextBox id=TextBox1 runat=server />
             </td>
             <td>
```

```
//điều khiển xác nhận hợp lệ trên hộp thoại //nhập liệu TextBox, nếu
chuỗi là trống khi //nhấn nút Validate thì sẽ bị bắt lỗi.
<asp:RequiredFieldValidator id="RequiredFieldValidator2"
                    ControlToValidate="TextBox1"
                    ErrorMessage="Card Number. "
                    Display="Static"
                    Width="100%" runat=server>
                    *
                  </asp:RequiredFieldValidator>
                </td>
              </tr>
              <tr>
                <td align=right>
<font face=Verdana size=2>Expiration Date:</font>
                </td>
                <td>

//hộp thoại DropDownList dùng để hiển thị //danh sách các ngày, nếu
người dùng chọn //mục trống trong DropDownList này thì sẽ bị //điều
khiển xác nhận hợp lệ //RequireFieldValidator3 bắt lỗi
<ASP:DropDownList id=DropDownList1 runat=server>
                    <asp:ListItem></asp:ListItem>
                    <asp:ListItem >06/00</asp:ListItem>
                    <asp:ListItem >07/00</asp:ListItem>
                    <asp:ListItem >08/00</asp:ListItem>
                    <asp:ListItem >09/00</asp:ListItem>
                    <asp:ListItem >10/00</asp:ListItem>
                    <asp:ListItem >11/00</asp:ListItem>
                    <asp:ListItem >01/01</asp:ListItem>
                    <asp:ListItem >02/01</asp:ListItem>
                    <asp:ListItem >03/01</asp:ListItem>
                    <asp:ListItem >04/01</asp:ListItem>
                    <asp:ListItem >05/01</asp:ListItem>
                    <asp:ListItem >06/01</asp:ListItem>
                    <asp:ListItem >07/01</asp:ListItem>
                    <asp:ListItem >08/01</asp:ListItem>
                    <asp:ListItem >09/01</asp:ListItem>
                    <asp:ListItem >10/01</asp:ListItem>
                    <asp:ListItem >11/01</asp:ListItem>
                    <asp:ListItem >12/01</asp:ListItem>
                  </ASP:DropDownList>
                </td>
                <td>

//điều khiển xác nhận hợp lệ trên //DropDownList1 hiển thị ngày hết
hạn, nếu //người dùng chọn một mục trống trên //DropDownList thì
điều khiển này sẽ phát //sinh ra lỗi
<asp:RequiredFieldValidator id="RequiredFieldValidator3"
                    ControlToValidate="DropDownList1"
                    ErrorMessage="Expiration Date. "
                    Display="Static"
                    InitialValue=""
                    Width="100%"
                    runat=server>
                    *
                  </asp:RequiredFieldValidator></td>
            </tr>
              <tr>
```

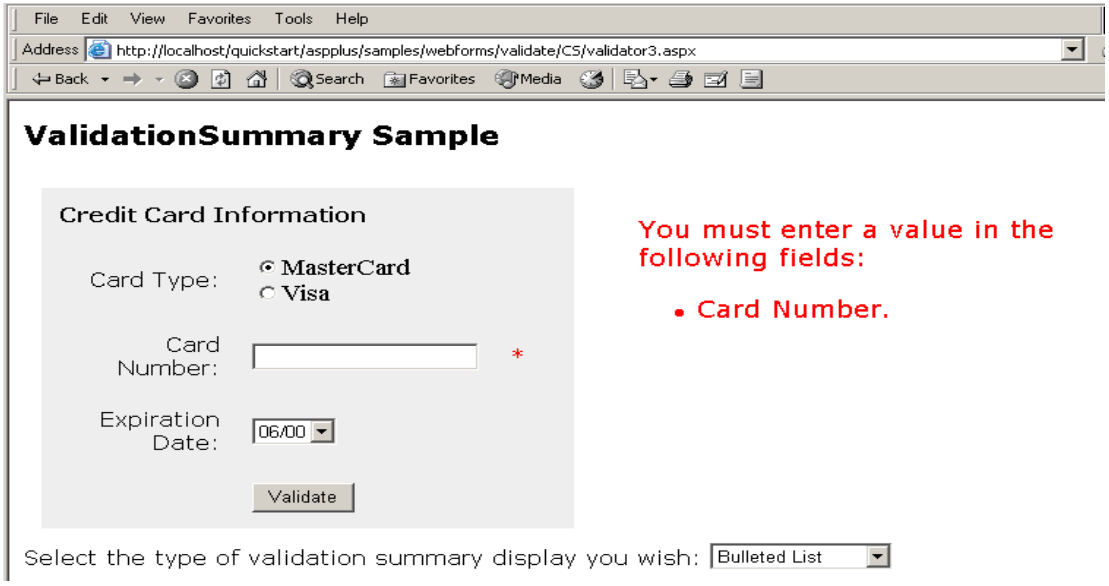
```
                <td>
                //nút nhấn để xác định hợp lệ
<ASP:Button id=Button1 text="Validate" runat=server /></td></tr>

        </table>
         </td>
         <td valign=top>
             <table cellpadding=20><tr><td>

//điều khiển dùng để hiện thị các lỗi lên trang, //nó sẽ bắt bất kỳ
lỗi nào được phát sinh bởi các //điều khiển DropDownList để hiển thị
            <asp:ValidationSummary ID="valSum" runat="server"
HeaderText="You must enter a value in the following fields:"
               Font-Name="verdana"
               Font-Size="12"
            />
            </td></tr></table>
         </td>
    </tr>
</table>

<font face="verdana" size="-1">Select the type of validation summary
display you wish: </font>

//Danh sách liệt kê 3 cách hiển thị lỗi
<asp:DropDownList id="ListFormat" AutoPostBack=true
OnSelectedIndexChanged="ListFormat_SelectedIndexChanged"
runat=server >
    <asp:ListItem>List</asp:ListItem>
    <asp:ListItem selected>Bulleted List</asp:ListItem>
    <asp:ListItem>Single Paragraph</asp:ListItem>
</asp:DropDownList>

  </form>
  </body>
  </html>
```

Giao diện của ví dụ khi chạy :

Hình 15-9 Khi chưa nhấn nút xác nhận Validate



Hình 15-10 Hiển thị lỗi do bỏ trống trên TextBox theo dạng dấu chấm tròn Bullet



Chương 16 Các dịch vụ Web

Hiện nay, vẫn còn một số hạn chế lớn trong các ứng dụng Web. Người dùng bị giới hạn chỉ thực hiện được những nội dung đã được cấu trúc cho một trang cụ thể và xem dữ liệu thông qua một số giao diện cụ thể nào đó đã được thiết kế trên máy chủ. Do đó người dùng muốn lấy được thông tin được linh động và hiệu quả hơn. Hơn nữa, thay vì ta hiển thị thông tin thông qua trình duyệt Web, ta muốn chạy một phần mềm trực tiếp trên máy khách mà có thể trao đổi dữ liệu trên máy chủ tuỳ ý. Công nghệ .NET cho phép xây dụng cách dịch vụ Web (Web Services) đáp ứng được các yêu cầu trên. Ý tưởng chính là : thay vì liệt kê các thông tin theo dạng HTML, trang tạo sẵn một loạt các lệnh gọi hàm. Các lệnh gọi hàm này có thể trao đổi thông tin qua lại giữa các hệ cơ sở dữ liệu trên máy chủ. Các hàm này có thể chấp nhận các tham số và có thể trả về một giá trị tùy ý.

Các dịch vụ Web vẫn dựa trên giao thức HTTP để truyền dữ liệu, đồng thời nó cần phải sử dụng thêm một loại giao thức để phục vụ cho việc gọi hàm. Hiện nay có hai giao thức được dùng chủ yếu là : **SOAP** (Simple Object Access Protocol) và **SDL** (Service Description Language, đây là giao thức riêng của Microsoft). Cả hai giao thức này đều được xây dụng dựa trên XML, mục đích chung của chúng là giúp định nghĩa các lệnh gọi hàm, tham số và giá trị.

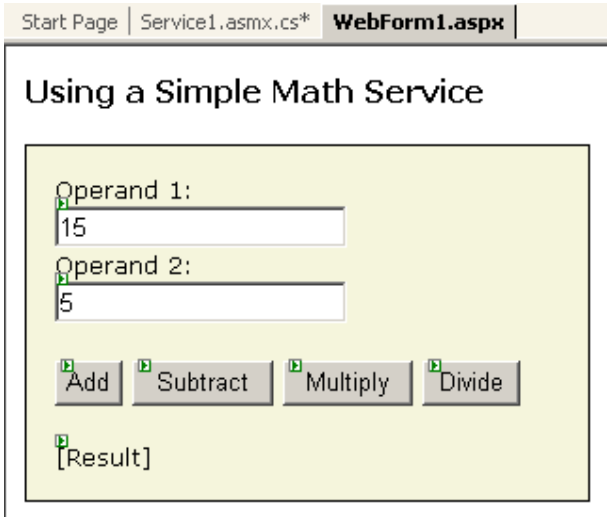
Ngoài ra, Microsoft cũng đưa ra thêm một ý tưởng mới về tập tin **Discovery File,** có phần mở rộng là .disco. Tập tin dạng này dùng để cung cấp các thông tin cho các trình duyệt để các trình duyệt này có thể xác định được các trang trên các máy chủ mà có chứa các dịch vụ Web.

Sau đây, ta sẽ tìm hiểu một ví dụ nhằm minh họa việc tạo ra một dịch vụ Web, đóng vai trò là một thư viện chứa một tập các hàm tiện ích. Trang Web của chúng ta sẽ sử dụng các hàm của dịch vụ này. Dịch vụ Web của chúng sẽ có tên **MathService**, đơn giản là định nghĩa bốn phương thức cộng, trừ, nhân, chia trên hai số thực bất kỳ. Mỗi phương thức đều nhận vào hai đối số kiểu số thực và trả về kết quả cũng có kiểu số thực.

Đầu tiên ta cần tạo một dự án kiểu Web Service bằng cách chọn : New Project\Visual C# Project\ASP.NET Web Service và đặt tên cho dự án là **MathService** và đổi tên dịch vụ thành MathService.asmx. NET có tạo sẵn cho chúng ta một số tập tin như :

- **Service1.asmx** : được trình duyệt yêu cầu, tương tự với tập tin .aspx.
- **WebService1.cs**: trang chứa mã C# quản lý.
- **DiscoFile1.disco**: tập tin khám phá.

Trong ví dụ này, chúng ta sẽ tạo ra một Web Form mới và thiết giao diện như sau :



Web Form sẽ gọi thực thi các hàm của dịch vụ Web.

Dự án của ta sẽ thừa kế namespace là `System.Web.Services.WebService`,nơi chứa các thuộc tính và phương thức cần thiết để tạo dịch vụ Web.

```
public class MathService : System.Web.Services.WebService
```

Trên mỗi phương thức ta cần khai báo thuộc tính `[WebMethod]`, để chỉ ra đây là phương thức sẽ được sử dụng cho dịch vụ Web. Mã của tập tin dịch vụ sẽ như sau :

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace MathService
{
  public class MathService :System.Web.Services.WebService
  {
    public MathService()
    {
      InitializeComponent();
    }

    #region Component Designer generated code

    private IContainer components = null;

    private void InitializeComponent()
    {

    }
    protected override void Dispose( bool disposing )
```

```
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #endregion

        //4 hàm toán học của dịch vụ Web, trên mỗi phương thức
        //ta cần khai báo thuộc tính [WebMethod] để chỉ đây là
        //phương thức dành cho dịch vụ Web.
        [WebMethod]
        public float Add(float a, float b)
        {
            return a + b;
        }

        [WebMethod]
        public float Subtract(float a, float b)
        {
            return a - b;
        }

        [WebMethod]
        public float Multiply(float a, float b)
        {
            return a * b;
        }

        [WebMethod]
        public float Divide(float a, float b)
        {
            if (b==0) return -1;
            return a / b;
        }
    }
}
```

Bây giờ chúng ta sẽ viết mã thực thi cho trang Web. Trang Web của chúng ta sẽ gọi các hàm của dịch vụ tương ứng với các phép cộng, trừ, nhân, chia . Sau đây là mã của trang Web:

```
<%@ Import Namespace="MathService" %>
<html>
<script language="C#" runat="server">
    float operand1 = 0;
    float operand2 = 0;
    public void Submit_Click(Object sender, EventArgs E)
    {
        try
        {
            operand1 = float.Parse(Operand1.Text);
            operand2 = float.Parse(Operand2.Text);
    }
    catch (Exception) { /* bỏ qua lỗi nếu có */ }
```

```
        //tạo ra một đối tượng dịch vụ MathService để có thể truy cập đến
        //các hàm thành viên của chúng.
        MathService service = new MathService();
          switch (((Control)sender).ID)
          {
              case "Add"          : Result.Text = "<b>Result</b> = " +
              service.Add(operand1, operand2).ToString(); break;

              case "Subtract"  : Result.Text = "<b>Result</b> = " +
              service.Subtract(operand1, operand2).ToString(); break;

              case "Multiply"  : Result.Text = "<b>Result</b> = " +
              service.Multiply(operand1, operand2).ToString(); break;

              case "Divide"      : Result.Text = "<b>Result</b> = " +
              service.Divide(operand1, operand2).ToString(); break;
          }
         }

        </script>

        <body style="font: 10pt verdana">
              <h4>Using a Simple Math Service
              </h4>
              <form runat="server">
                    <div             style="padding:15,15,15,15;background-
                    color:beige;width:300;border-color:black;border-
                    width:1;border-style:solid">
                    Operand 1:<br>
                    <asp:TextBox  id="Operand1"  Text="15"  runat="server"
                    /><br>

                    Operand 2:<br>
                    <asp:TextBox  id="Operand2"  Text="5"  runat="server"
                    /><p>

                    <input      type="submit"      id="Add"      value="Add"
                    OnServerClick="Submit_Click" runat="server">

                    <input  type="submit"  id="Subtract"  value="Subtract"
                    OnServerClick="Submit_Click" runat="server">

                    <input  type="submit"  id="Multiply"  value="Multiply"
                    OnServerClick="Submit_Click" runat="server">

                    <input    type="submit"    id="Divide"    value="Divide"
                    OnServerClick="Submit_Click" runat="server">
                    <p>
                    <asp:Label id="Result" runat="server" />
                    </div>
              </form>
        </body>
</html>
```

Chương 17 Assemblies và Versioning

Đơn vị cơ bản trong lập trình .NET là **Assembly**. Một Assembly là một tập hợp các tập tin mà đối với người sử dụng, họ chỉ thấy đó là một tập tin DLL hay EXE.

.NET định nghĩa Assembly là một đơn vị có khả năng tái sử dụng (re-use), mang số hiệu phiên bản (versioning), bảo mật (security) và cuối cùng là khả năng triển khai (deployment)

Asssembly có thể chứa đựng nhiều thành phần khác ngoài mã chương trình ứng dụng như tài nguyên (resource, ví dụ tập tin .GIF), thông tin mô tả kiểu (type definition), siêu dữ liệu (metadata) về mã và dữ liệu.

17.1 Tập tin PE

Assembly được lưu trữ trên dĩa từ theo dạng thức tập tin Portable Executable (PE). Dạng thức tập tin PE của .NET cũng giống như tập tin PE bình thường của Windows NT. Dạng thức PE được cài đặt thành dạng thức tập tin DLL và EXE.

Về mặt logic, assembly chứa đựng một hay nhiều module. Mỗi module được tổ chức thành một DLL và đồng thời mỗi module là một cấu thành của assembly. Các module tự bản thân chúng không thể chạy được, các module phải kết hợp với nhau thành assembly thì mới có thể làm được việc gì đó hữu ích.

17.2 Metadata

Metadata là thông tin được lưu trữ bên trong assembly với mục đích là để mô tả các kiểu dữ liệu, các phương thức và các thông tin khác về assembly. Do có chứa metadata nên assembly có khả năng **tự mô tả**.

17.3 Ranh giới an ninh

Assembly tạo ra một ranh giới an ninh (security boundary). Các kiểu dữ liệu định nghĩa bên trong assembly bị giới hạn phạm vi tại ranh giới assembly. Để có thể sử dụng chung một kiểu dữ liệu giữa 2 assembly, cần phải chỉ định rõ bằng tham chiếu (reference) trong IDE hoặc dòng lệnh.

17.4 Số hiệu phiên bản (Versioning)

Mỗi assembly có số hiệu phiên bản riêng. Một "phiên bản" ám chỉ toàn bộ nội dung của một assembly bao gồm cả kiểu dữ liệu và resource.

17.5 Manifest

Manifest chính là một thành phần của metadata. Manifest mô tả một assembly chứa những gì, ví dụ như: thông tin nhận dạng (tên, phiên bản), danh sách các kiểu dữ

liệu, danh sách các resource, danh sách các assembly khác được assembly này tham chiếu đến, …

17.5.1 Các module trong manifest

Một assembly có thể chứa nhiều module, do đó manifest trong assembly còn có thể chứa mã băm (hash code) của mỗi module lắp ghép thành assembly để bảo đảm rằng khi thực thi, chỉ có thể nạp các module đúng phiên bản.

Chỉ cần một sự thay đổi rất rất nhỏ trong module là mã băm sẽ thay đổi.

17.5.2 Manifest trong module

Mỗi module cũng chứa riêng phần manifest mô tả cho chính nó giống như assembly chứa manifest mô tả cho assembly vậy.

17.5.3 Các assembly cần tham chiếu

Manifest của assembly cũng có thể chứa tham chiếu đến các assembly khác. Mỗi tham chiếu chứa đựng tên, phiên bản, văn hóa (culture), nguồn gốc (originator),…

Thông tin về nguồn gốc chính là chữ ký số (digital signature) của lập trình viên hay của công ty nơi cung cấp assembly mà assembly hiện tại đang tham chiếu đến.

Văn hóa là một đối tượng chứa thông tin về ngôn ngữ, cách trình bày của mỗi quốc gia. Ví dụ như cách thể hiện ngày tháng: D/M/Y hay M-D-Y

17.6 Đa Module Assembly

Một assembly đơn module là một assembly chỉ gồm một module, module này có thể là một tập tin EXE hoặc DLL. Manifest cho assembly đơn module được nhúng vào trong module.

Một assembly đa module là một assembly bao gồm nhiều tập tin (ít nhất một tập tin EXE hoặc DLL). Manifest cho assembly đa module có thể được lưu trữ thành một tập tin riêng biệt hoặc được nhúng vào một module nào đó bất kỳ.

17.6.1 Lợi ích của đa module assembly

Nếu một dự án có nhiều lập trình viên mà dự án đó chỉ xây dựng bằng một assembly, việc kiểm lỗi, biên dịch dự án,… là một "ác mộng" vì tất cả các lập trình viên phải hợp tác với nhau, phải kiểm tra phiên bản, phải đồng bộ hóa mã nguồn,…

Nếu một ứng dụng lớn được xây dựng bằng nhiều assembly, khi cần cập nhật (update) để sửa lỗi chẳng hạn, thì chỉ cần cập nhật một / vài assembly mà thôi.

Nếu một ứng dụng lớn được tổ chức từ nhiều assembly, chỉ có những phần mã chương trình thường sử dụng / quan trọng thuộc một vài assembly là được nạp vào bộ nhớ, do đó làm giảm bớt chi phí bộ nhớ, tăng hiệu suất hệ thống.

17.7 Assembly nội bộ (private assembly)

Có 2 loại Assembly: nội bộ (private) và chia sẻ (shared). Assembly nội bộ được dự định là chỉ dùng cho một ứng dụng, còn assembly chia sẻ thì ngược lại, dùng cho nhiều ứng dụng.

Các assembly nội bộ được ghi trên dĩa từ thành một tập tin EXE hoặc DLL trong cùng thư mục với assembly chính hoặc trong các thư mục con của thư mục chứa assembly chính. Để thực thi trên máy khác chỉ cần sao chép đúng cấu trúc thư mục là đủ, không cần phải đăng ký với Registry.

17.8 Assembly chia sẻ (shared assembly)

Khi viết ra một assembly đại loại như một control chẳng hạn, nếu tác giả của control đó có ý định chia sẻ cho các lập trình viên khác thì anh / chị ta phải xây dựng assembly đó đáp ứng các yêu cầu sau:

- Assembly đó phải có tên "mạnh" (strong name). Tên mạnh có nghĩa là chuỗi biểu diễn tên đó phải là duy nhất (globally unique)
- Phải có thông tin về phiên bản để tránh hiện tượng các phiên bản "dẫm chân lên nhau"
- Để có thể chia sẻ assembly, assembly đó phải được đặt vào nơi gọi là **Global Assembly Cache (GAC)**. Đây là nơi được quy định bởi Common Language Runtime (CLR) dùng để chứa assembly chia sẻ.

17.8.1 Chấm dứt "địa ngục DLL"

Giả sử bạn cài đặt một ứng dụng A lên một máy và nó chạy tốt. Sau đó bạn cài đặt ứng dụng B, bỗng nhiên ứng dụng A không chịu hoạt động. Sau quá trình tìm hiểu, cuối cùng nguyên nhân là do ứng dụng B đã cài một phiên bản khác đè lên một tập tin DLL mà ứng dụng A sử dụng. Tình huống trên gọi là "địa ngục DLL"

Sự ra đời của assembly đã chấm dứt tình trạng trên.

17.8.2 Phiên bản

Assembly chia sẻ trong .NET được định vị bằng tên duy nhất (unique) và phiên bản. Phiên bản được biểu diễn bởi 4 số phân cách bằng dấu ':' ví dụ như 1:2:6:1246

```
Số đầu tiên mô tả phiên bản chính (major version)
Số thứ 2 mô tả phiên bản phụ (minor version)
Số thứ 3 mô tả thứ tự bản xây dựng (build)
Số cuối cùng mô tả lần xem xét cập nhật (revision) để sửa lỗi
```

17.8.3 Tên mạnh

Một tên mạnh là một chuỗi các ký tự hexa mang thuộc tính là duy nhất trong toàn cầu (globally unique). Ngoài ra chuỗi đó còn được mã hóa bằng thuật toán khóa công khai [1] để bảo đảm rằng assembly không bị thay đổi vô tình hay cố ý.

Để tạo ra tên mạnh, một cặp khóa công khai-bí mật được tạo ra cho assembly. Một mã băm (hash code) được tạo ra từ tên, nội dung của các tập tin bên trong assembly và chuỗi biểu diễn khóa công khai. Sau đó mã băm này được mã hóa bằng khóa bí mật, kết quả mã hóa được ghi vào manifest. Quá trình trên được gọi là ký xác nhận vào assembly (signing the assembly).

Khi assembly được CLR nạp vào bộ nhớ, CLR sẽ dùng khóa công khai trong manifest giải mã mã băm để xác định xem assembly có bị thay đổi không.

17.8.4 Global Assembly Cache (GAC)

Sau khi đã tạo tên mạnh và ghi vào assembly, việc còn lại để thực hiện chia sẻ assembly là đặt assembly đó vào thư mục GAC. Đó là một thư mục đặc biệt dùng để chứa assembly chia sẻ. Trên Windows, đó là thư mục \WinNT\assembly.

[1] Mã hóa khóa công khai – bí mật: đó là một thuật toán mã hóa đặc biệt, đầu tiên dùng một thuật toán riêng tạo ra 2 khóa, một khóa phổ biến rộng rãi nên gọi là khóa công khai, khóa còn lại do chủ nhân của nó cất nơi an toàn nên gọi là bí mật. Sau đó dùng thuật toán mã hóa để mã hóa dữ liệu. Một khi dữ liệu bị mã hóa bằng một khóa thì dữ liệu đó chỉ có thể được giải mã bằng khóa kia và ngược lại.

Chương 18 Attributes và Reflection

Xin được nhắc lại rằng một ứng dụng .NET bao gồm mã chương trình, dữ liệu, metadata. Metadata chính là thông tin về dữ liệu mà ứng dụng sử dụng như kiểu dữ liệu, mã thực thi, assembly,…

Attributes là cơ chế để tạo ra metadata. Ví dụ như chỉ thị cho trình biên dịch, những dữ liệu khác liên quan đến dữ liệu, phương thức, lớp, …

Reflection là quá trình một ứng dụng đọc lại metadata của chính nó để có cách thể hiện, ứng xử thích hợp cho từng người dùng.

18.1 Attributes

Một attribute là một đối tượng, trong đối tượng đó chứa một mẩu dữ liệu, mà lập trình viên muốn đính kèm với một phần tử (element) nào đó trong ứng dụng. Phần tử (element) mà lập trình viên muốn đính kèm attribute gọi là mục tiêu (target) của attribute. Ví dụ attribute:

[NoIDispatch]

được đính kèm với một lớp hay một giao diện để nói rằng lớp đích (target class) nên được thừa kế từ giao diện IUnknown chứ không phải thừa kế từ IDispatch.

18.2 Attribute mặc định (intrinsic attributes)

Có 2 loại attribute:

- Attribute mặc định: là attribute được CLR cung cấp sẵn.
- Attribute do lập trình viên định nghĩa (custom attribute)

18.2.1 Đích của Attribute

Mỗi attribute chỉ ảng hưởng đến một đích (target) mà nó khai báo. Đích có thể là lớp, giao diện, phương thức … Bảng sau liệt kê tất cả các đích

Bảng 18-1 Các đích của attribute

Loại	Ý nghĩa
All	Áp dụng cho tất cà các loại bên dưới
Assembly	Áp dụng cho chính assembly
Class	Áp dụng cho một thể hiện của lớp
ClassMembers	Áp dụng cho các loại từ sau hàng này trở đi

Constructor	Áp dụng với hàm dựng
Delegate	Áp dụng cho delegate
Enum	Áp dụng cho kiểu liệt kê
Event	Áp dụng cho sự kiện
Field	Áp dụng cho biến thành viên (tĩnh lẫn không tĩnh)
Interface	Áp dụng cho giao diện
Method	Áp dụng cho phương thức
Module	Áp dụng cho module
Parameter	Áp dụng cho tham số
Property	Áp dụng cho property
ReturnValue	Áp dụng cho trị trả về
Struct	Áp dụng cho cấu trúc

18.2.2 Áp dụng Attribute

Lập trình viên áp dụng attribute lên mục tiêu bằng cách đặt attribute trong ngoặc vuông [] liền trước mục tiêu. Ví dụ attribute "Assembly" được áp dụng:

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("keyfile.snk")]
```

Cách sau cũng tương đương với cách trên:

```
[assembly: AssemblyDelaySign(false), assembly:
AssemblyKeyFile("keyfile.snk")]
```

Attribute thường dùng trong lập trình C# là **"Serializable"**

```
[serializable]
class MySerClass
```

Attribute trên báo cho compiler biết rằng lớp **MySerClass** cần được bảo đảm trong việc ghi nội dung, trạng thát xuống dĩa từ hay truyền qua mạng.

18.3 Attribute do lập trình viên tạo ra

Lập trình viên hoàn toàn tự do trong việc tạo ra các attribute riêng và đem sử dụng chúng vào nơi nào cảm thấy thích hợp.

18.3.1 Khai báo Attribute tự tạo

Đầu tiên là thừa kế một lớp từ lớp **System.Attribute**:

```
Public class XYZ : System.Attribute
```

Sau đó là báo cho compiler biết attribute này có thể đem áp dụng lên mục tiêu nào.

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

Attribute **"AttributeUsage"** trên có mục tiêu áp dụng là Attribute khác: gọi là meta-attribute.

18.3.2 Đặt tên một attribute

Lập trình viên hoàn toàn tự do trong việc đặt tên cho attribute. Tuy nhiên, compiler của .NET còn có thêm khả năng tự nối thêm chuỗi "Attribute" vào tên. Điều đó có nghĩa là nếu lập trình viên định nghĩa một attribute có tên là **"MyBugFix"** thì khi tìm kiếm hoặc truy xuất attribute trên, lập trình viên có thể viết tên attribute: **"MyBugFix"** hoặc **"MyBugFixAttribute".** Nếu một attribute có tên là **"MyBugFixAttribute"** thì lập trình viên cũng có thể ghi tên attribute là **"MyBugFix"** hoặc **"MyBugFixAttribute"**.

Ví dụ:

```
[MyBugFix(123, "Jesse Liberty", "01/01/05", Comment="Off by one")]
```

18.3.3 Khởi tạo Attribute

Mỗi attribute phải có ít nhất một contructor. Attribute nhận 2 kiểu đối số: kiểu vị trí (positional) và kiểu tên (named).

Trong ví dụ **MyBugFix** ở phần trước, phần tên và ngày tháng là kiểu vị trí, phần ghi chú (comment) là kiểu tên.

Các đối số kiểu vị trí phải được truyền vào contructor đúng theo thứ tự khai báo. Ví dụ:

```
public BugFixAttribute(int bugID, string programmer,
                   string date)
{
  this.bugID = bugID;
  this.programmer = programmer;
  this.date = date;
}
```

Đối số kiểu tên thì được cài đặt như là properties:

```
public string Comment
{
  get { return comment; }
  set { comment = value; }
}
```

18.3.4 Sử dụng Attribute

Một khi đã định nghĩa attribute, lập trình viên sử dụng nó bằng cách đặt nó ngay trước mục tiêu (target) của nó. Ví dụ:

```
[BugFixAttribute(121,"Jesse Liberty","01/03/05")]
BugFixAttribute(107,"Jesse Liberty","01/04/05",
Comment="Fixed off by one errors")]
public class MyMath
```

Ví dụ trên áp dụng attribute MyBugFix vào lớp MyMath.

18.4 Reflection

Để cho việc lưu attribute trong metadata có ích, cần phải có cơ chế truy xuất chúng vào lúc chạy. Các lớp trong vùng tên (namespace) Reflection, cùng với các lớp trong System.Type và System.TypeReference, cung cấp sự hỗ trợ truy xuất metadata.

Reflection là một khái niệm chung cho bất kỳ thao tác nào trong các thao tác sau đây:

- Xem xét metadata
- Tìm hiểu kiểu dữ liệu (type discovery): lớp, interface, phương thức, đối số của phương thức, properties, event, field, …
- Nối kết trễ các phương thức và properties (late binding to methods and properties)
- Tạo ra một kiểu dữ liệu mới ngay trong lúc thực thi. Lập trình viên có thể định nghĩa một assembly mới ngay lúc chạy, có thể lưu xuống dĩa từ để dùng lại.

Chương 19 Marshaling và Remoting

Ngày nay, các ứng dụng không còn đơn thuần chỉ gồm một module, khi thực thi thì chỉ cần chạy trong một process mà là một tập hợp nhiều thành phần (component) phức tạp. Các thành phần đó không chỉ phân cách với nhau bằng ranh giới giữa các process mà còn có thể phân cách với nhau qua ranh giới máy - mạng - máy.

Tiến trình **di chuyển** một đối tượng vượt qua một ranh giới (process, máy, …) được gọi là **Remoting**.

Tiến trình **chuẩn bị** để một đối tượng thực hiện remoting được gọi là **Marshaling**.

Giả sử đối tượng A nằm trên máy X muốn sử dụng dịch vụ của đối tượng B nằm trên máy Y. Để phục vụ đối tượng A, đối tượng B chuyển cho A một đối tượng **proxy**. Những yêu cầu từ đối tượng A sẽ được proxy chuyển về cho B, những kết quả trả lời của B được gởi đến proxy, proxy sẽ gởi lại cho đối tượng A. Giữa đối tượng A và đối tượng B có nhiều đối tượng **sink**, công việc của các đối tượng sink là áp đặt an ninh lên kênh liên lạc giữa 2 đối tượng. Các thông điệp được chuyển tải giữa A và B trên một đối tượng **channel**. Đối tượng channel lại yêu cầu sự giúp đỡ của đối tượng **formatter**. Công việc của formatter là định dạng lại thông điệp để 2 phía có thể hiểu nhau (ví dụ chuyển mã hóa endian của dãy byte).

19.1 Miền Ứng Dụng (Application Domains)

Theo lý thuyết, một process là một ứng dụng đang thực thi (đang chạy). Mỗi một application thực thi trong một process riêng của nó. Nếu trên máy hiện có Word, Excel, Visual Studio thì tương ứng trên máy đang có 3 process.

Bên trong mỗi process, .NET chia nhỏ ra thành các phần nhỏ hơn gọi là **miền ứng dụng (Application Domains viết tắt là app domains)**. Có thể xem mỗi miền ứng dụng là một process "nhẹ cân", miền ứng dụng hành xử y như là một process nhưng điểm khác biệt là nó sử dụng ít tài nguyên hơn process.

Các miền ứng dụng trong một process có thể khởi động (started) hay bị treo (halted) độc lập với nhau. Miền ứng dụng cung cấp khả năng chịu lỗi (fault tolerance); nếu khởi động một đối tượng trong một miền ứng dụng khác với miền ứng dụng chính và đối tượng vừa khởi động gây lỗi, nó chỉ làm crash miền ứng dụng của nó chứ không làm crash toàn bộ ứng dụng.

Mỗi process lúc bắt đầu thực thi có một miền ứng dụng ban đầu (initial app domain) và có thể tạo thêm nhiều miền ứng dụng khác nếu lập trình viên muốn. Thông thường, ứng dụng chỉ cần một miền ứng dụng là đủ. Tuy nhiên, trong những ứng dụng lớn cần sử dụng những thư viện do người khác viết mà thư viện đó không

được tin cậy lắm thì cần tạo ra một miền ứng dụng khác dùng để chứa thư viện không tin cập đó, tách thư viện đó khỏi miền ứng dụng chính để cô lập lỗi, nếu lỗi xảy ra thì không làm crash ứng dụng.

Miền ứng dụng khác với thread. Một thread luôn chạy bên trong một miền ứng dụng. Trong một miền ứng dụng có thể tồn tại nhiều thread.

19.1.1 Marshaling vượt qua biên miền ứng dụng

Marshaling là quá trình chuẩn bị một đối tượng để di chuyển qua một ranh giới nào đó. Marshaling có thể được tiến hành theo 2 cách: bằng giá trị (by value) và bằng tham chiếu (by reference).

- Khi một đối tượng được marshaling bằng giá trị, một bản sao của đối tượng được tạo ra và truyền đến nơi nhận. Những thay đổi trên bản sao này không làm thay đổi đối tượng gốc ban đầu.
- Khi một đối tượng được marshaling bằng tham chiếu, một đối tượng đặc biệt gọi là proxy được gởi đến nơi nhận. Những thay đổi, những lời gọi hàm trên đối tượng proxy sẽ được chuyển về cho đối tượng ban đầu xử lý.

19.1.1.1 Tìm hiểu marshaling và proxy

Khi marshaling đối tượng bằng reference, .NET CLR cung cấp cho đối tượng đang thực hiện lời gọi từ xa một đối tượng proxy "trong suốt" (transparent proxy - TP).

Công việc của đối tượng TP là nhận tất cả những thông tin gì liên quan đến việc gọi hàm (giá trị trả về, thông số nhập, …) từ trong stack rồi đóng gói vào một đối tượng riêng mà đối tượng đó đã cài đặt giao diện **IMessage**. Sau đó đối tượng IMessage đó được trao cho đối tượng **RealProxy**.

RealProxy là một lớp cơ sở trừu tượng (abstract base class) mà từ đó các đối tượng proxy thừa kế. Lập trình viên có thể tự tạo ra các đối tượng proxy mới thừa kế từ RealProxy. Đối tượng proxy mặc định của hệ thống sẽ trao IMessage cho một chuỗi các đối tượng **sink**. Số lượng các sink phụ thuộc vào số lượng chính sách bảo an (policy) mà nhà quản trị muốn duy trì, tuy nhiên đối tượng sink cuối cùng là đối tượng đặt IMessage vào Channel.

Channel được chia ra thành channel phía client và channel phía server, công việc chính của channel là di chuyển thông điệp (message) vượt qua một ranh giới (boundary). Channel chịu trách nhiệm tìm hiểu nghi thức truyền thông (transport protocol). Định dạng thật sự của thông điệp di chuyển qua ranh giới được quản lý bởi **formatter**. Khung ứng dụng (framework) .NET cung cấp 2 formatter:

- Simple Object Access Protocol (SOAP) dùng cho HTTP channel
- Binary dùng cho TCP/IP channel

Lập trình viên cũng có thể tạo đối tượng formatter riêng và nếu muốn cũng có thể tạo ra channel riêng.

Một khi message vượt qua ranh giới, nó được nhận bởi channel và formatter phía server. Channel phía server sẽ tái tạo lại đối tượng IMessage, sau đó channel phía server trao đối tượng IMessage cho một hay nhiều đối tượng sink phía server. Đối tượng sink cuối cùng trong chuỗi sink là một đối tượng **StackBuilder**, công việc của StackBuilder là nhận IMessage rồi tái tạo lại stack frame để có thể thực hiện lời gọi hàm.

19.1.1.2 Chỉ định phương pháp Marshaling

Một đối tượng bình thường hoàn toàn không có khả năng marshaling.

Để marshaling một đối tượng bằng giá trị (by value), chỉ cần đánh dấu attribute Serializable lúc định nghĩa đối tượng.

```
[Serializable]
public class Point
```

Để marshaling một đối tượng bằng tham chiếu (by reference), đối tượng đó cần thừa kế từ **MarshalByRefObject**.

```
public class Shape : MarshalByRefObject
```

19.2 Context

Miền ứng dụng (app domain) đến lượt nó lại được chia ra thành các **context**. Context có thể xem như một ranh giới mà các đối tượng bên trong context có cùng quy tắc sử dụng (usage rules). Các quy tắc sử dụng như: đồng bộ hóa giao dịch (synchronization transaction), …

19.2.1 Đối tượng loại Context-Bound và Context-Agile

Các đối tượng có thể là Context-Bound hoặc Context-Agile.

Nếu các đối tượng là Context-Bound, chúng tồn tại trong một context riêng, khi giao tiếp lẫn nhau, các thông điệp của chúng được marshaling để vượt qua biên context.

Nếu các đối tượng là Context-Agile, chúng hoạt động bên trong context của đối tượng yêu cầu (calling). Do đó, khi một đối tượng A triệu gọi phương thức của đối tượng B, phương thức của B được thực thi bên trong context của A. Vì vậy việc marshaling là không cần thiết.

Giả sử đối tượng A cần giao tiếp với cơ sở dữ liệu, giả sử đối tượng A có thiết lập về giao dịch (transaction). Do đó A cần tạo một context. Tất cả các phương thức của A sẽ được thực thi trong context của transaction.

Giả sử có một đối tượng B khác thuộc loại context-agile. Giả sử rằng đối tượng A trao một tham chiếu cơ sở dữ liệu cho đối tượng B và triệu gọi một phương thức X của B. Lại giả sử rằng phương thức X của B mà A triệu gọi lại gọi một phương thức Y khác của A. Bởi vì B thuộc loại context-agile do đó phương thức X đó của B được thực thi trong context của đối tượng A. Vì context của A có sự bảo vệ của giao

dịch nên nếu A có roll-back cơ sở dữ liệu thì những thay đổi mà phương thức X của B lên cơ sở dữ liệu cũng sẽ được roll-back.

Giả sử đối tượng C triệu gọi một phương thức Z của B, phương thức Z có thực hiện thay đổi cơ sở dữ liệu, do B thuộc loại Context-Agile nên Z được thực thi trong context của C. Vì vậy những thay đổi mà Z thực hiện lên cơ sở dữ liệu sẽ không thể được roll-back.

Nếu B thuộc loại Context-Bound, khi A tạo ra B, context của B sẽ được thừa kế từ context của A. Khi C triệu gọi phương thức Z của B, lời triệu gọi đó được marshaling tử context của C đến context của B rồi được thực hiện trong context của B (thừa kế từ A). Do đó phương thức Z thực thi trong sự bảo vệ của transaction. Những thay đổi mà Z thực hiện lên cơ sở dữ liệu có thể được rooll-back.

Một đối tượng có thể có 3 lựa chọn về Context:

- Context-Agile
- Context-Bound không chỉ định attribute. Thực hiện bằng cách thừa kế từ ContextBoundObject. Phương thức của đối tượng thuộc loại này thực thi trong Context thừa kế từ Context của đối tượng tạo ra nó
- Context-Bound có chỉ định attribute Context. Các phương thức hoạt động trong Context do nó tạo riêng.

19.2.2 Marshaling vượt qua ranh giới Context

Khi truy cập đối tượng Context-Agile trong cùng miền ứng dụng thì không cần proxy. Khi một đối tượng A trong một context truy cập một đối tượng Context-Bound B trong một context khác, đối tượng A đó truy cập đối tượng B thông qua proxy của B.

Đối tượng được marshaling khác nhau vượt qua ranh giới context phụ thuộc vào cách nó được tạo ra:

- Đối tượng bình thường không có marshaling; bên trong miền ứng dụng các đối tượng thuộc loại context-agile
- Đối tượng có đánh dấu attribute Serializable thì thuộc loại context-agile và được marshaling bởi giá trị (by value) vượt qua ranh giới miền ứng dụng
- Đối tượng thừa kế từ MarshalByRefObject thì thuộc loại context-agile và được marshaling bởi tham chiếu (by reference) vượt qua ranh giới miền ứng dụng
- Đối tượng thừa kế từ ContextBoundObject được marshaling bởi tham chiếu vượt qua ranh gới miền ứng dụng và ranh giới context

19.3 Remoting

Cùng với việc marshaling đối tượng vượt qua ranh giới context và miền ứng dụng, đối tượng có thể được marshaling vượt qua ranh giới process và thậm chí qua ranh giới máy - mạng - máy. Khi một đối tượng được marshaling vượt qua ranh giới process hay máy - mạng – máy, nó được gọi là **Remoted**.

19.3.1 Tìm hiểu các kiểu dữ liệu phía Server

Có 2 kiểu đối tượng phía server phục vụ cho việc **Remoting** trong .NET: **nổi tiếng (well-known)** và **client kích hoạt (client activated)**. Kênh liên lạc với đối tượng nổi tiếng được thiết lập mỗi khi client gởi thông điệp (message). Kênh liên lạc đó không được giữ thường trực như trong trường hợp của đối tượng client kích hoạt.

Đối tượng nổi tiếng chia thành 2 loại nhỏ: **singleton** và **single-call**.

- Đối tượng nổi tiếng kiểu singleton: tất cả các thông điệp từ client gởi đến đối tượng được phân phối (dispatch) cho **một** đối tượng đang chạy trên server. Đối tượng này được tạo ra khi server khởi động và nằm chờ trên server để phục vụ cho bất kỳ client nào. Vì vậy đối tượng loại này phải có contructor không tham số.
- Đối tượng nổi tiếng kiểu single-call: mỗi thông điệp mới từ client gởi đi được giải quyết bởi một đối tượng mới. Mô hình này thường dùng để cân bằng tải hệ thống.

Đối tượng client kích hoạt thường được sử dụng bởi các lập trình viên có công việc chính là tạo ra các server riêng phục vụ cho việc lập trình, đối tượng client kích hoạt duy trì kết nối với client cho đến khi nào toàn bộ yêu cầu của client được đáp ứng.

19.3.2 Mô tả một server bằng Interface

Sau đây là ví dụ xây dựng một lớp máy tính (calculator) với 4 chức năng.

Tạo một tập tin ICalc.cs với nội dung

```
namespace Programming_CSharp
using System;
public interface ICalc
{
   double Add(double x, double y);
   double Sub(double x, double y);
   double Mult(double x, double y);
   double Div(double x, double y);
}
```

Tạo một project mới kiểu C# Class Library, mở menu Build, ra lệnh Build. Kết quả là một tập tin Icalc.DLL

19.3.3 Xây dựng một Server

Tạo một project kiểu C# Console Application, tạo một tập tin mới CalcServer.cs, sau đó ra lệnh Build.

Lớp Calculator thừa kế từ MarshalByRefObect nên nó sẽ trao cho client một proxy khi được triệu gọi.

```
public class Calculator : MarshalByRefObject, ICalc
```

Công việc đầu tiên là tạo channel và đăng ký, sử dụng HTTPChannel cung cấp bởi .NET:

```
HTTPChannel chan = new HTTPChannel(65100);
```

Đăng ký channel với .NET Channel Services:

```
ChannelServices.RegisterChannel(chan);
```

Đăng ký đối tượng (cần cung cấp **endpoint** cho hàm đăng ký; endpoint chính là tên liên kết với đối tượng)

```
Type calcType = Type.GetType("Programming_CSharp.Calculator");
```

Đăng ký kiểu Singleton

```
RemotingConfiguration.RegisterWellKnownServiceType
   ( calcType, "theEndPoint",WellKnownObjectMode.Singleton );
```

Sau đây là toàn bộ nội dung:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Programming_CSharp
{
   // implement the calculator class
   public class Calculator : MarshalByRefObject, ICalc
   {
      public Calculator( )
      {
         Console.WriteLine("Calculator constructor");
      }

      // implement the four functions
      public double Add(double x, double y)
      {
         Console.WriteLine("Add {0} + {1}", x, y);
         return x+y;
      }

      public double Sub(double x, double y)
      {
         Console.WriteLine("Sub {0} - {1}", x, y);
         return x-y;
      }

      public double Mult(double x, double y)
```

```
      {
        Console.WriteLine("Mult {0} * {1}", x, y);
        return x*y;
      }

      public double Div(double x, double y)
      {
        Console.WriteLine("Div {0} / {1}", x, y);
        return x/y;
      }
    }

    public class ServerTest
    {
      public static void Main( )
      {
        // tạo một channel và đăng ký nó
        HttpChannel chan = new HttpChannel(65100);
        ChannelServices.RegisterChannel(chan);
        Type calcType =
          Type.GetType("Programming_CSharp.Calculator");
        // register our well-known type and tell the server
        // to connect the type to the endpoint "theEndPoint"
        RemotingConfiguration.RegisterWellKnownServiceType
          ( calcType, "theEndPoint",
            WellKnownObjectMode.Singleton );
        // "They also serve who only stand and wait."); (Milton)
        Console.WriteLine("Press [enter] to exit...");
        Console.ReadLine( );
      }
    }
}
```

19.3.4 Xây dựng Client

Client cũng phải đăng ký channel, tuy nhiên client không lắng nghe trên channel đó nên client sử dụng channel 0 (zero)

```
HTTPChannel chan = new HTTPChannel(0);
ChannelServices.RegisterChannel(chan);
```

Client kết nối với dịch vụ từ xa, trao cho hàm kết nối kiểu đối tượng mà nó cần cộng với URI của lớp cài đặt dịch vụ

```
MarshalByRefObject obj = RemotingServices.Connect(
       typeof(Programming_CSharp.ICalc),
       "http://localhost:65100/theEndPoint");
```

Vì đối tượng từ xa có thể không sẵn sàng (mạng đứt, máy chứa đối tượng không tồn tại,…) nên để gọi hàm cần khối thử

```
try
{
  Programming_CSharp.ICalc calc = obj as
            Programming_CSharp.ICalc;
  double sum = calc.Add(3,4);
```

Bây giờ client đã có đối tượng proxy của đối tượng Calculator. Sau đây là toàn bộ mã nguồn

```
namespace Programming_CSharp
{
  using System;
  using System.Runtime.Remoting;
  using System.Runtime.Remoting.Channels;
  using System.Runtime.Remoting.Channels.Http;
  public class CalcClient
  {
    public static void Main( )
    {
      int[] myIntArray = new int[3];
      Console.WriteLine("Watson, come here I need you...");
      // create an Http channel and register it
      // uses port 0 to indicate won't be listening
      HttpChannel chan = new HttpChannel(0);
      ChannelServices.RegisterChannel(chan);
      // get my object from across the http channel
      MarshalByRefObject obj =
        (MarshalByRefObject) RemotingServices.Connect
        (typeof(Programming_CSharp.ICalc),
        "http://localhost:65100/theEndPoint");
      try
      {
        // cast the object to our interface
        Programming_CSharp.ICalc calc =
          obj as Programming_CSharp.ICalc;
        // use the interface to call methods
        double sum = calc.Add(3.0,4.0);
        double difference = calc.Sub(3,4);
        double product = calc.Mult(3,4);
        double quotient = calc.Div(3,4);
        // print the results
        Console.WriteLine("3+4 = {0}", sum);
        Console.WriteLine("3-4 = {0}", difference);
        Console.WriteLine("3*4 = {0}", product);
        Console.WriteLine("3/4 = {0}", quotient);
      }
      catch( System.Exception ex )
      {
        Console.WriteLine("Exception caught: ");
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

```
Kết xuất phía client
Watson, come here I need you...
3+4 = 7
3-4 = -1
3*4 = 12
3/4 = 0.75
Kết xuất phía server:
Calculator constructor
Press [enter] to exit...
Add 3 + 4
Sub 3 - 4
Mult 3 * 4
```

```
Div 3 / 4
```

19.3.5 Sử dụng SingleCall

Để thấy sự khác biệt giữa Singleton và Single Call, thay đổi một dòng trong mã nguồn Calculator.cs

```
RemotingServices. RegisterWellKnownServiceType(
   "CalcServerApp","Programming_CSharp.Calculator",
   "theEndPoint",WellKnownObjectMode.Singleton );
```

thành

```
RemotingServices. RegisterWellKnownServiceType(
   "CalcServerApp","Programming_CSharp.Calculator",
   "theEndPoint",WellKnownObjectMode.SingleCall );
```

Và đây là kết quả kết xuất phía server

```
Calculator constructor
Press [enter] to exit...
Calculator constructor
Add 3 + 4
Calculator constructor
Sub 3 - 4
Calculator constructor
Mult 3 * 4
Calculator constructor
Div 3 / 4
```

19.3.6 Tìm hiểu RegisterWellKnownServiceType

Khi gọi phương thức RegisterWellKnownServiceType, điều gì đã xảy ra ?

Xin nhớ lại rằng bạn đã tạo một đối tượng **Type** cho lớp **Calculator**

```
Type.GetType("Programming_CSharp.Calculator");
```

Sau đó bạn gọi RegisterWellKnownServiceType(), trao cho phương thức đó đối tượng Type, endpoint và Singleton. Điều đó báo cho CLR biết phải tạo một thể hiện của Calculator và liên kết endpoint với thể hiện đó.

Bạn có thể tự làm lại quá trình đó bằng cách thay đổi hàm Main() như sau:

Ví dụ 19-1 Manually instantiating and associating Calculator with an endpoint

```
public static void Main( )
{
   // create a channel and register it
   HttpChannel chan = new HttpChannel(65100);
   ChannelServices.RegisterChannel(chan);
   // make your own instance and call Marshal directly
   Calculator calculator = new Calculator( );
   RemotingServices.Marshal(calculator,"theEndPoint");
   // "They also serve who only stand and wait."); (Milton)
   Console.WriteLine("Press [enter] to exit...");
   Console.ReadLine( );
}
```

19.3.7 Tìm hiểu EndPoint

Chuyện gì xảy ra khi bạn đăng ký endpoint ? Rõ ràng là server liên kết endpoint với đối tượng bạn vừa tạo, và khi client kết nối, server sử dụng chỉ mục trong một bảng để có thể trả về đối tượng proxy tương ứng với đối tượng yêu cầu.

Bạn có thể không cung cấp endpoint, thay vào đó bạn ghi các thông tin về đối tượng Calculator rồi gởi về client. Client "phục chế" lại đối tượng proxy để có thể dùng liên lạc với Calculator trên server.

Để có thể hiểu làm cách nào bạn có thể gọi một đối tượng mà không biết endpoint, hãy thay đổi hàm Main() một lần nữa, thay vì gọi **Marshal()** với một endpoint, hãy trao một đối tượng:

```
ObjRef objRef = RemotingServices.Marshal(calculator)
```

Hàm Marshal() trả về một đối tượng ObjRef. Đối tượng ObjRef chứa tất cả thông tin cần thiết để kích hoạt và liên lạc với đối tượng ở xa. Khi bạn cung cấp một endpoint, server tạo một bảng và liên kết endpoint với một ObjRef để khi client có yêu cầu, server có thể tạo một proxy cung cấp cho client. ObjRef chứa tất cả thông tin cần thiết cho client để client có thể tạo một proxy. ObjRef có khả năng **Serializable**.

Mở một stream tập tin, tạo một SOAP formatter, bạn có thể serialize đối tượng ObjRef thành một tập tin bằng cách gọi hàm Serialize() của formatter, sau đó đưa cho formatter tham chiếu đến stream tập tin và tham chiếu của ObjRef, vậy là bạn đã có tất cả thông tin cần thiết để tạo một proxy dưới dạng một tập tin

Ví dụ 19-2 Marshaling an object without a well-known endpoint

```
public static void Main( )
{
   // create a channel and register it
   HttpChannel chan = new HttpChannel(65100);
   ChannelServices.RegisterChannel(chan);
   // make your own instance and call Marshal directly
   Calculator calculator = new Calculator( );
   ObjRef objRef = RemotingServices.Marshal(calculator);
   FileStream fileStream =
   new FileStream("calculatorSoap.txt",FileMode.Create);
   SoapFormatter soapFormatter = new SoapFormatter( );
   soapFormatter.Serialize(fileStream,objRef);
   fileStream.Close( );
   // "They also serve who only stand and wait."); (Milton)
   Console.WriteLine(
   "Exported to CalculatorSoap.txt. Press ENTER to exit...");
   Console.ReadLine( );
}
```

Bạn hãy trao tập tin chứa đối tượng đã serialize đó cho client. Để client có thể tái tạo lại đối tượng, client cần tạo một channel và đăng ký nó.

```
FileStream fileStream = new FileStream ("calculatorSoap.txt",
     FileMode.Open);
```

Sau đó tạo một thể hiện của đối tượng **SoapFormatter** rồi gọi hàm DeSerialize() của formatter để nhận lại đối tượng ObjRef

```
SoapFormatter soapFormatter =
new SoapFormatter ( );
try
{
  ObjRef objRef =
    (ObjRef) soapFormatter.Deserialize (fileStream);
```

Tiến hành gỡ bỏ marshall, nhận lại ICalc

```
ICalc calc = (ICalc) RemotingServices.Unmarshal(objRef);
```

Bây giờ client có thể triệu gọi phương thức trên server thông qua ICalc.

Ví dụ 19-3 Replacement of Main( ) from Example 19-4 (the client)

```
public static void Main( )
{
  int[] myIntArray = new int[3];
  Console.WriteLine("Watson, come here I need you...");
  // create an Http channel and register it
  // uses port 0 to indicate you won't be listening
  HttpChannel chan = new HttpChannel(0);
  ChannelServices.RegisterChannel(chan);
  FileStream fileStream =
  new FileStream ("calculatorSoap.txt", FileMode.Open);
  SoapFormatter soapFormatter =
  new SoapFormatter ( );
  try
  {
    ObjRef objRef =
    (ObjRef) soapFormatter.Deserialize (fileStream);
    ICalc calc =
    (ICalc) RemotingServices.Unmarshal(objRef);
    // use the interface to call methods
    double sum = calc.Add(3.0,4.0);
    double difference = calc.Sub(3,4);
    double product = calc.Mult(3,4);
    double quotient = calc.Div(3,4);
    // print the results
    Console.WriteLine("3+4 = {0}", sum);
    Console.WriteLine("3-4 = {0}", difference);
    Console.WriteLine("3*4 = {0}", product);
    Console.WriteLine("3/4 = {0}", quotient);
  }
  catch( System.Exception ex )
  {
    Console.WriteLine("Exception caught: ");
    Console.WriteLine(ex.Message);
  }
}
```

Chương 20 Thread và Sự Đồng Bộ

Thread là một process "nhẹ cân" cung cấp khả năng multitasking trong một ứng dụng. Vùng tên System.Threading cung cấp nhiều lớp và giao diện để hỗ trợ lập trình nhiều thread.

20.1 Thread

Thread thường được tạo ra khi bạn muốn làm **đồng thời 2 việc trong cùng một thời điểm**. Giả sử ứng dụng của bạn đang tiến hành đọc vào bộ nhớ một tập tin có kích thước khoảng 500MB, trong lúc đang đọc thì dĩ nhiên ứng dụng không thể đáp ứng yêu cầu xử lý giao diện. Giả sử người dùng muốn ngưng giữa chừng, không cho ứng dụng đọc tiếp tập tin lớn đó nữa, do đó cần một thread khác để xử lý giao diện, lúc này khi người dùng ấn nút Stop thì ứng dụng đáp ứng được yêu cầu trong khi thread ban đầu vẫn đang đọc tập tin.

20.1.1 Tạo Thread

Cách đơn giản nhất là tạo một thể hiện của lớp Thread. Contructor của lớp Thread nhận một tham số kiểu delegate. CLR cung cấp lớp delegate ThreadStart nhằm mục đích chỉ đến phương thức mà bạn muốn thread mới thực thi. Khai báo delegate ThreadStart như sau:

```
public delegate void ThreadStart( );
```

Phương thức mà bạn muốn gán vào delegate phải không chứa tham số và phải trả về kiểu **void**. Sau đây là ví dụ:

```
Thread myThread = new Thread( new ThreadStart(myFunc) );
```

myFunc phải là phương thức không tham số và trả về kiểu void.

Xin lưu ý là đối tượng Thread mới tạo sẽ **không** tự thực thi (execute), để đối tượng thực thi, bạn càn gọi phương thức Start() của nó.

```
Thread t1 = new Thread( new ThreadStart(Incrementer) );
Thread t2 = new Thread( new ThreadStart(Decrementer) );
t1.Start( );
t2.Start( );
```

Thread sẽ chấm dứt khi hàm mà nó thực thi trở về (return).

20.1.2 Gia nhập Thread

Hiện tượng thread A ngưng chạy và chờ cho thread B chạy xong được gọi là thread A gia nhập thread B.

Để thread 1 gia nhập thread 2:

```
t2.Join( );
```

Nếu câu lệnh trên được thực hiện bởi thread 1, thread 1 sẽ dừng lại và chờ cho đến khi thread 2 kết thúc.

20.1.3 Treo thread lại (suspend thread)

Nếu bạn muốn treo thread đang thực thi lại một khoảng thời gian thì bạn sử dụng hàm Sleep() của đối tượng Thread. Ví dụ để thread ngưng khoảng 1 giây:

```
Thread.Sleep(1000);
```

Câu lệnh trên báo cho bộ điều phối thread (của hệ điều hành) biết bạn **không muốn** bộ điều phối thread phân phối thời gian CPU cho thread thực thi câu lệnh trên trong thời gian 1 giây.

20.1.4 Giết một Thread (Kill thread)

Thông thường thread sẽ chấm dứt khi hàm mà nó thực thi trở về. Tuy nhiên bạn có thể yêu cầu một thread "tự tử" bằng cách gọi hàm **Interrupt()** của nó. Điều này sẽ làm cho exception **ThreadInterruptedException** được ném ra. Thread bị yêu cầu "tự tử" có thể bắt exception này để tiến hành dọn dẹp tài nguyên.

```
catch (ThreadInterruptedException)
{
  Console.WriteLine("[{0}] Interrupted! Cleaning up...",
  Thread.CurrentThread.Name);
}
```

20.2 Đồng bộ hóa (Synchronization)

Khi bạn cần bảo vệ một tài nguyên, trong một lúc chỉ cho phép một thread thay đổi hoặc sử dụng tài nguyên đó, bạn cần **đồng bộ hóa**.

Đồng bộ hóa được cung cấp bởi một khóa trên đối tượng đó, khóa đó sẽ ngăn cản thread thứ 2 truy cập vào đối tượng nếu thread thứ nhất chưa trả quyền truy cập đối tượng.

Sau đây là ví dụ **cần** sự đồng bộ hóa. Giả sử 2 thread sẽ tiến hành tăng **tuần tự 1 đơn vị** một biến tên là counter.

```
int counter = 0;
```

Hàm làm thay đổi giá trị của Counter:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
  {
    int temp = counter;
    temp++; // increment
    // simulate some work in this method
    Thread.Sleep(1);
    // assign the Incremented value
```

```
        // to the counter variable
        // and display the results
        counter = temp;
        Console.WriteLine(
        "Thread {0}. Incrementer: {1}",
        Thread.CurrentThread.Name,
        counter);
      }
  }
```

Vấn đề ở chỗ thread 1 đọc giá trị counter vào biến tạm rồi tăng giá trị biến tạm, trước khi thread 1 ghi giá trị mới từ biến tạm trở lại counter thì thread 2 lại đọc giá trị counter ra biến tạm của thread 2. Sau khi thread 1 ghi giá trị vừa tăng 1 đơn vị trở lại counter thì thread 2 lại ghi trở lại counter giá trị mới bằng với giá trị mà thread 1 vừa ghi. Như vậy sau 2 lần truy cập giá trị của biến counter chỉ tăng 1 đơn vị trong khi yêu cầu là phải tăng 2 đơn vị.

20.2.1 Sử dụng Interlocked

CLR cung cấp một số cơ chế đồng bộ từ cơ chế đơn giản **Critical Section** (gọi là Locks trong .NET) đến phức tạp như **Monitor**.

Tăng và giảm giá trị làm một nhu cầu phổ biến, do đó C# cung cấp một lớp đặc biệt **Interlocked** nhằm đáp ứng nhu cầu trên. Interlocked có 2 phương thức **Increment()** và **Decrement()** nhằm tăng và giảm giá trị **trong sự bảo vệ** của cơ chế đồng bộ. Ví dụ ở phần trước có thể sửa lại như sau:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      Interlocked.Increment(ref counter);
      // simulate some work in this method
      Thread.Sleep(1);
      // assign the decremented value
      // and display the results
      Console.WriteLine(
      "Thread {0}. Incrementer: {1}",
      Thread.CurrentThread.Name,
      counter);
    }
  }
}
```

Khối catch và finally không thay đổi so với ví dụ trước.

20.2.2 Sử dụng Locks

Lock đánh dấu một đoạn mã "gay cấn" (critical section) trong chương trình của bạn, cung cấp cơ chế đồng bộ cho khối mã mà lock có hiệu lực.

C# cung cấp sự hỗ trợ cho lock bằng từ chốt (keyword) **lock**. Lock được gỡ bỏ khi hết khối lệnh. Ví dụ:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      lock (this)
      { // lock bắt đầu có hiệu lực
        int temp = counter;
        temp ++;
        Thread.Sleep(1);
        counter = temp;
      } // lock hết hiệu lực -> bị gỡ bỏ
      // assign the decremented value
      // and display the results
      Console.WriteLine( "Thread {0}. Incrementer: {1}",
             Thread.CurrentThread.Name, counter);
  }
}
```

Khối catch và finally không thay đổi so với ví dụ trước.

20.2.3 Sử dụng Monitor

Để có thể đồng bộ hóa phức tạp hơn cho tài nguyên, bạn cần sử dụng monitor. Một monitor cho bạn khả năng quyết định khi nào thì bắt đầu, khi nào thì kết thúc đồng bộ và khả năng chờ đợi một khối mã nào đó của chương trình "tự do".

Khi cần bắt đầu đồng bộ hóa, trao đối tượng cần đồng bộ cho hàm sau:

```
Monitor.Enter(đối tượng X);
```

Nếu monitor không sẵn dùng (unavailable), đối tượng bảo vệ bởi monitor đang được sử dụng. Bạn có thể làm việc khác trong khi chờ đợi monitor sẵn dùng (available) hoặc treo thread lại cho đến khi có monitor (bằng cách gọi hàm Wait())

Ví dụ bạn đang download và in một bài báo từ Web. Để hiệu quả bạn cần tiến hành in sau hậu trường (background), tuy nhiên bạn cần chắc chắn rằng 10 trang đã được download trước khi bạn tiến hành in.

Thread in ấn sẽ chờ đợi cho đến khi thread download báo hiệu rằng số lượng trang download đã đủ. Bạn không muốn gia nhập (join) với thread download vì số lượng trang có thể lên đến vài trăm. Bạn muốn chờ cho đến khi ít nhất 10 trang đã được download.

Để giả lập việc này, bạn thiết lập 2 hàm đếm dùng chung 1 biến counter. Một hàm đếm tăng 1 tương ứng với thread download, một hàm đếm giảm 1 tương ứng với thread in ấn.

Trong hàm làm giảm bạn gọi phương thức **Enter(),** sau đó kiểm tra giá trị counter, nếu < 5 thì gọi hàm **Wait()**

```
if (counter < 5)
{
  Monitor.Wait(this);
}
```

Lời gọi Wait() giải phóng monitor nhưng bạn đã báo cho CLR biết là bạn muốn lấy lại monitor ngay sau khi monitor được tự do một lần nữa. Thread thực thi phương thức Wait() sẽ bị treo lại. Các thread đang treo vì chờ đợi monitor sẽ tiếp tục chạy khi thread đang thực thi gọi hàm **Pulse()**.

```
Monitor.Pulse(this);
```

Pulse() báo hiệu cho CLR rằng có sự thay đổi trong trạng thái monitor có thể dẫn đến việc giải phóng (tiếp tục chạy) một thread đang trong tình trạng chờ đợi.

Khi thread hoàn tất việc sử dụng monitor, nó gọi hàm **Exit()** để trả monitor.

```
Monitor.Exit(this);
```

Source code ví dụ:

```
namespace Programming_CSharp
{
  using System;
  using System.Threading;
  class Tester
  {
    static void Main( )
    {
      // make an instance of this class
      Tester t = new Tester( );
      // run outside static Main
      t.DoTest( );
    }
    public void DoTest( )
    {
      // create an array of unnamed threads
      Thread[] myThreads = {
           new Thread( new ThreadStart(Decrementer) ),
           new Thread( new ThreadStart(Incrementer) ) };
      // start each thread
      int ctr = 1;
      foreach (Thread myThread in myThreads)
      {
        myThread.IsBackground=true;
        myThread.Start( );
        myThread.Name = "Thread" + ctr.ToString( );
        ctr++;
        Console.WriteLine("Started thread {0}",myThread.Name);
        Thread.Sleep(50);
      }
      // wait for all threads to end before continuing
      foreach (Thread myThread in myThreads)
      {
        myThread.Join( );
      }
      // after all threads end, print a message
      Console.WriteLine("All my threads are done.");
    }
```

```
    void Decrementer( )
    {
      try
      {
        // synchronize this area of code
        Monitor.Enter(this);
        // if counter is not yet 10
        // then free the monitor to other waiting
        // threads, but wait in line for your turn
        if (counter < 10)
        {
          Console.WriteLine(
          "[{0}] In Decrementer. Counter: {1}. Gotta Wait!",
            Thread.CurrentThread.Name, counter);
          Monitor.Wait(this);
        }
        while (counter >0)
        {
          long temp = counter;
          temp--;
          Thread.Sleep(1);
          counter = temp;
          Console.WriteLine("[{0}] In Decrementer. Counter: {1}.",
            Thread.CurrentThread.Name, counter);
        }
      }
      finally
      {
        Monitor.Exit(this);
      }
    }
    void Incrementer( )
    {
      try
      {
        Monitor.Enter(this);
        while (counter < 10)
        {
          long temp = counter;
          temp++;
          Thread.Sleep(1);
          counter = temp;
          Console.WriteLine("[{0}] In Incrementer. Counter: {1}",
            Thread.CurrentThread.Name, counter);
        }
        // I'm done incrementing for now, let another
        // thread have the Monitor
        Monitor.Pulse(this);
      }
      finally
      {
        Console.WriteLine("[{0}] Exiting...",
          Thread.CurrentThread.Name);
        Monitor.Exit(this);
      }
    }
    private long counter = 0;
  }
```

```
}
Kết quả:
Started thread Thread1
[Thread1] In Decrementer. Counter: 0. Gotta Wait!
Started thread Thread2
[Thread2] In Incrementer. Counter: 1
[Thread2] In Incrementer. Counter: 2
[Thread2] In Incrementer. Counter: 3
[Thread2] In Incrementer. Counter: 4
[Thread2] In Incrementer. Counter: 5
[Thread2] In Incrementer. Counter: 6
[Thread2] In Incrementer. Counter: 7
[Thread2] In Incrementer. Counter: 8
[Thread2] In Incrementer. Counter: 9
[Thread2] In Incrementer. Counter: 10
[Thread2] Exiting...
[Thread1] In Decrementer. Counter: 9.
[Thread1] In Decrementer. Counter: 8.
[Thread1] In Decrementer. Counter: 7.
[Thread1] In Decrementer. Counter: 6.
[Thread1] In Decrementer. Counter: 5.
[Thread1] In Decrementer. Counter: 4.
[Thread1] In Decrementer. Counter: 3.
[Thread1] In Decrementer. Counter: 2.
[Thread1] In Decrementer. Counter: 1.
[Thread1] In Decrementer. Counter: 0.
All my threads are done.
```

20.3 Race condition và DeadLock

Đồng bộ hóa thread khá rắc rối trong những chương trình phức tạp. Bạn cần phải cẩn thận kiểm tra và giải quyết các vấn đề liên quan đến đồng bộ hóa thread: race condition và deadlock

20.3.1 Race condition

Một điều kiện tranh đua xảy ra khi sự đúng đắn của ứng dụng **phụ thuộc** vào thứ tự hoàn thành **không kiểm soát** được của 2 thread **độc lập** với nhau.

Ví dụ: giả sử bạn có 2 thread. Thread 1 tiến hành mở tập tin, thread 2 tiến hành ghi lên cùng tập tin đó. Điều quan trọng là bạn cần phải điều khiển thread 2 sao cho nó chỉ tiến hành công việc sau khi thread 1 đã tiến hành xong. Nếu không, thread 1 sẽ không mở được tập tin vì tập tin đó đã bị thread 2 mở để ghi. Kết quả là chương trình sẽ ném ra exception hoặc tệ hơn nữa là crash.

Để giải quyết vấn đề trong ví dụ trên, bạn có thể tiến hành join thread 2 với thread 1 hoặc thiết lập monitor.

20.3.2 Deadlock

Giả sử thread A đã nắm monitor của tài nguyên X và đang chờ monitor của tài nguyên Y. Trong khi đó thì thread B lại nắm monitor của tài nguyên Y và chờ

monitor của tài nguyên X. 2 thread cứ chờ đợi lẫn nhau mà không thread nào có thể thoát ra khỏi tình trạng chờ đợi. Tình trạng trên gọi là deadlock.

Trong một chương trình nhiều thread, deadlock rất khó phát hiện và gỡ lỗi. Một hướng dẫn để tránh deadlock đó là giải phóng tất cả lock đang sở hữu nếu tất cả các lock cần nhận không thể nhận hết được. Một hướng dẫn khác đó là giữ lock càng ít càng tốt.

Chương 21 Luồng dữ liệu.

Khi muốn đọc hay ghi dữ liệu vào/ra tập tin hay muốn truyền dữ liệu từ máy này sang máy khác, ta phải tổ chức dữ liệu theo cấu trúc tuần tự các byte hay các gói tin …. Điều này dễ liên tưởng dữ liệu như là các luồng dữ liệu chảy từ từ nguồn đến đích.

Thư viện .NET Framework cung cấp các lớp `Stream` (`Stream` và các lớp thừa kế từ nó) để chương trình có thể sử dụng trong các thao tác nhập xuất dữ liệu như doc/ghi tập tin, truyền dữ liệu qua mạng …

21.1 Tập tin và thư mục

Các lớp đề cập trong chương này thuộc về vùng tên `System.IO`. Các lớp này bao gồm lớp `File` mô phỏng cho một tập tin trên đĩa, và lớp `Directory` mô phỏng cho một thư mục trên đĩa.

21.1.1 Làm việc với thư mục

Lớp `Directory` có nhiều phương thức dành cho việc tạo, di chuyển, duyệt thư mục. Các phương thức trong lớp `Directory` đều là phương thức tĩnh;vì vậy không cần phải tạo một thể hiện lớp `Directory` mà có thể truy xuất trực tiếp từ tên lớp.

Lớp `DirectoryInfo` là lớp tương tự như lớp `Directory`. Nó cung các tất cả các phương thức mà lớp `Directory` có đồng thời bổ sung nhiều phương thức hữu ích hơn cho việc duyệt cấu trúc cây thư mục. Lớp `DirectoryInfo` thừa kế từ lớp `FileSystemInfo`, và vì vậy cũng thừa kế lớp `MarshalByRefObj`. Lớp `DirectoryInfo` không có phương thức tĩnh, vì vậy cần tạo một thể hiện lớp trước khi sử dụng các phương thức.

Có một khác biệt quan trong giữa `Directory` và `DirectoryInfo` là các phương thức của lớp `Directory` sẽ được kiểm tra về bảo mật mỗi khi được gọi trong khi đối tượng `DirectoryInfo` chỉ kiểm tra một lần vào lúc khởi tạo, các phương thức vì vậy sẽ thực hiện nhanh hơn.

Dưới đây là bảng liệt kê các phương thức quan trọng của hai lớp

Bảng 21-1 Các phương thức lớp Directory

Phương thức	Giải thích
CreateDirectory()	Tạo tất cả các thư mục và thư mục con trong đường dẫn tham số.
Delete()	Xoá thư mục và các nội dung của nó.

Exists( )	Trả về kết quả kiểu logic, đúng nếu đường dẫn đến thư mục tồn tại (có nghĩa là thư mục tồn tại).
GetCreationTime( ) SetCreationTime( )	Lấy/thiết đặt ngày giờ tạo thư mục
GetCurrentDirectory( ) SetCurrentDirectory( )	Lấy/thiết đặt thư mục hiện hành
GetDirectories( )	Lấy về một mảng các thư mục con một thư mục
GetDirectoryRoot( )	Trả về thư mục gốc của đường dẫn
GetFiles( )	Trả về mảng chuỗi tên các tập tin chứa trong một thư mục
GetLastAccessTime( ) SetLastAccessTime( )	Lầy/thiết đặt ngày giờ lần truy cập cuối cùng đến thư mục
GetLastWriteTime( ) SetLastWriteTime( )	Lầy/thiết đặt ngày giờ lần chỉnh sửa cuối cùng lên thư mục
GetLogicalDrives( )	Trả về tên của tất cả các ổ đĩa logic theo định dạng <ổ_đĩa>:\
GetParent()	Trả về thư mục cha của một đường dẫn.
Move()	Di chuyển thư mục (cả nội dung) đến một vị trí khác.

Bảng 21-2 Các phương thức/property lớp DirectoryInfo

Phương thức/property	Ý nghĩa
Attributes	Thừa kế từ FileSystemInfo, lấy/thiết đặt thuộc tính của tập tin hiện hành.
CreationTime	Thừa kế từ FileSystemInfo, lấy/thiết đặt thời gian tạo tập tin
Exists	Trả về đúng nếu thư mục tồn tại
Extension	Thừa kế từ FileSystemInfo, phần mở rộng tập tin
FullName	Thừa kế từ FileSystemInfo, đường dẫn đầy đủ của tập tin hay thư mục
LastAccessTime	Thừa kế từ FileSystemInfo, ngày giờ truy cập cuối cùng
LastWriteTime	Thừa kế từ FileSystemInfo, ngày giờ chỉnh sửa cuối cùng
Name	Tên thư mục
Parent	Lấy thư mục cha
Root	Lấy thư mục gốc của đường dẫn.
Create( )	Tạo một thư mục
CreateSubdirectory()	Tạo một hoặc nhiều thư mục con
Delete( )	Xóa một thư mục và nội dung của nó
GetDirectories( )	Trả về danh sách các thư mục con của thư hiện hiện có
GetFiles( )	Lấy danh mục các tập tin của thư mục
GetFileSystemInfos()	Nhận về mảng các đối tượng FileSystemInfo
MoveTo( )	Di chuyển DirectoryInfo và nội dung của nó sang đường dẫn khác

Phương thức/property	Ý nghĩa
Refresh( )	Làm tươi trạng thái đối tượng

21.1.2 Tạo đối tượng DirectoryInfo

Để duyệt cấu trúc cây thư mục, ta cần tạo một thể hiện của lớp `DirectoryInfo`. Lớp `DirectoryInfo` không chỉ cung cấp phương thức lấy về tên các tập tin và thư mục con chứa trong một thư mục mà còn cho phép lấy về các đối tượng `FileInfo` và `DirectoryInfo`, cho phép ta thực hiện việc quản lý các cấu trúc cây thư mục, hay thực hiện các thao tác đệ qui.

Khởi tạo một đối tượng `DirectoryInfo` bằng tên của thư mục muốn tham chiếu.

```
DirectoryInfo dir = new DirectoryInfo(@"C:\winNT");
```

Ta có thể thực hiện các phương thức đã liệt kê ở bảng trên. Dưới đây là đoạn mã nguồn ví dụ.

Ví dụ 21-1. Duyệt các thư mục con

```
using System;
using System.IO;

namespace Programming_CSharp
{
  class Tester
  {
    public static void Main()
    {
      Tester t = new Tester( );
      // một một thư mục
      string theDirectory = @"c:\WinNT";
      // duyệt thư mục và hiển thị ngày truy cập gần nhất
      // và tất cả các thư mục con
      DirectoryInfo dir = new DirectoryInfo(theDirectory);
      t.ExploreDirectory(dir);
      // hoàn tất. in ra số lượng thống kê
      Console.WriteLine( "\n\n{0} directories found.\n",
                         dirCounter);
    }

    // với mỗi thư mục tìm thấy, nó gọi chính nó
    private void ExploreDirectory(DirectoryInfo dir)
    {
      indentLevel++; // cấp độ thư mục
      // định dạng cho việc trình bày
      for (int i = 0; i < indentLevel; i++)
        Console.Write(" "); // hai khoảng trắng cho mỗi cấp
      // in thư mục và ngày truy cập gần nhất
      Console.WriteLine("[{0}] {1} [{2}]\n",
        indentLevel, dir.Name, dir.LastAccessTime);
      // lấy tất cả thư mục con của thư mục hiện tại
      // đệ quy từng thư mục
      DirectoryInfo[] directories = dir.GetDirectories( );
      foreach (DirectoryInfo newDir in directories)
      {
```

```
            dirCounter++; //tăng biến đếm
            ExploreDirectory(newDir);
          }
          indentLevel--; // giảm cấp độ thư mục
        }

        // các biến thành viên tĩnh cho việc thống kê và trình bày
        static int dirCounter = 1;
        static int indentLevel = -1; // so first push = 0
    }
}
```

```
Kết quả (một phần):
   [2] logiscan [5/1/2001 3:06:41 PM]
   [2] miitwain [5/1/2001 3:06:41 PM]
[1] Web [5/1/2001 3:06:41 PM]
   [2] printers [5/1/2001 3:06:41 PM]
     [3] images [5/1/2001 3:06:41 PM]
   [2] Wallpaper [5/1/2001 3:06:41 PM]
363 directories found.
```

Chương trình tạo một đối tượng `DirectoryInfo` gắn với thư mục WinNT. Sau đó gọi hàm `ExploreDirectory` với tham số là đối tượng `DirectoryInfo` vừa tạo. Hàm sẽ hiển thị các thông tin về thư mục này và sau đó lấy tất cả các thư mục con.

Để liệt kê danh sách các thư mục con, hàm gọi phương thức `GetDirectories`. Phương thức này trả về mảng các đối tượng `DirectoryInfo`. Bằng cách gọi đệ qui chính nó, hàm liệt kê xuống các thư mục con và thư mục con của thư mục con … Kết quả cuối cùng là cấu trúc cây thư mục được hiển thị.

21.1.3 Làm việc với tập tin.

Đối tượng `DirectoryInfo` cũng trả về danh sách các đối tượng FileInfo là các tập tin chứa trong thư mục. Các đối tượng này mô tả thông tin về tập tin. Thư viện .NET cũng cung cấp hai lớp `File` và `FileInfo` tương tự như với trường hợp thư mục. Lớp `File` chỉ có các phương thức tĩnh và lớp `FileInfo` thì không có phương thức tĩnh nào cả.

Hai bảng dưới đây liệt kê các phương thức cũa hai lớp này

Bảng 21-3 Các phương thức lớp File

Phương thức	Giải thích
AppendText()	Tạo một StreamWriter cho phép thêm văn bản vào tập tin
Copy()	Sao chép một tập tin từ tập tin đã có
Create()	Tạo một tập tin mới
CreateText()	Tạo một StreamWriter cho phép viết mới văn bản vào tập tin
Delete()	Xoá một tập tin
Exists()	Trả về đúng nếu tập tin tồn tại
GetAttributes()	Lấy/ thiết đặt các thuộc tính của một tập tin

Phương thức	Giải thích
SetAttributes()	
GetCreationTime() SetCreationTime()	Lấy / thiết đặt thời gian tạo tập tin
GetLastAccessTime() SetLastAccessTime()	Lấy / thiết đặt thời gian truy cập tập tin lần cuối
GetLastWriteTime() SetLastWriteTime()	Lấy / thiết đặt thời gian chỉnh sửa tập tin lần cuối
Move()	Di chuyển tập tin đến vị trí mới, có thể dùng để đổi tên tập tin
OpenRead()	Mở một tập tin để đọc (không ghi)
OpenWrite()	Mở một tập tin cho phép ghi.

Bảng 21-4 Các phương thức / property lớp FileInfo

Phương thức / property	Giải thích
Attributes()	Thừa kế từ FileSystemInfo. Lấy/thiết đặt thuộc tính tập tin
CreationTime	Thừa kế từ FileSystemInfo. Lấy/thiết đặt thời gian tạo tập tin
Directory	Lấy thư mục cha
Exists	Xác định tập tin có tồn tại chưa?
Extension	Thừa kế từ FileSystemInfo. Phần mở rộng của tập tin
FullName	Thừa kế từ FileSystemInfo. Đường dẫn đầy đủ của tập tin
LastAccessTime	Thừa kế từ FileSystemInfo. Thời điểm truy cập gần nhất
LastWriteTime	Thừa kế từ FileSystemInfo. Thời điểm ghi gần nhất.
Length	Kívh thước tập tin
Name	Tên tập tin
AppendText()	Tạo đối tượng StreamWriter để ghi thêm vào tập tin
CopyTo()	Sao chép sang một tập tin mới
Create()	Tạo một tập tin mới
Delete()	Xóa tập tin
MoveTo()	Dịch chuyển tập tin, cũng dùng để đổi tên tập tin
Open()	Mở một tập tin với các quyền hạn
OpenRead()	Tạo đối tượng FileStream cho việc đọc tập tin
OpenText()	Tạo đối tượng StreamReader cho việc đọc tập tin
OpenWrite()	Tạo đối tượng FileStream cho việc ghi tập tin

Ví dụ 21-2 sửa lại từ ví dụ 12-1, thêm đoạn mã lấy `FileInfo` của mỗi thư mục. Đối tượng này dùng để hiển thị tên, kích thước và ngày truy cấp cuối cùng của tập tin.

Ví dụ 21-2. Duyệt tập tin và thư mục con

```
using System;
using System.IO;

namespace Programming_CSharp
{
  class Tester
  {
    public static void Main( )
    {
      Tester t = new Tester( );
      string theDirectory = @"c:\WinNT";
      DirectoryInfo dir = new DirectoryInfo(theDirectory);

      t.ExploreDirectory(dir);

      Console.WriteLine(
        "\n\n{0} files in {1} directories found.\n",
        fileCounter,dirCounter );
    }

    private void ExploreDirectory(DirectoryInfo dir)
    {
      indentLevel++;
      for (int i = 0; i < indentLevel; i++)
        Console.Write(" ");
      Console.WriteLine("[{0}] {1} [{2}]\n",
        indentLevel, dir.Name, dir.LastAccessTime);

      // lấy tất cả các tập tin trong thư mục và
      // in tên, ngày truy cập gần nhất, kích thước của chúng
      FileInfo[] filesInDir = dir.GetFiles( );
      foreach (FileInfo file in filesInDir)
      {
        // lùi vào một khoảng phía dưới thư mục
        // phục vụ việc trình bày
        for (int i = 0; i < indentLevel+1; i++)
          Console.Write(" "); // hai khoảng trắng cho mỗi cấp
        Console.WriteLine("{0} [{1}] Size: {2} bytes",
          file.Name, file.LastWriteTime,    file.Length);
        fileCounter++;
      }
      DirectoryInfo[] directories = dir.GetDirectories( );
      foreach (DirectoryInfo newDir in directories)
      {
        dirCounter++;
        ExploreDirectory(newDir);
      }
      indentLevel--;
      }

    // các biến tĩnh cho việc thống kê và trình bày
    static int dirCounter = 1;
    static int indentLevel = -1;
    static int fileCounter = 0;
  }
}
```

```
Kết quả (một phần):
[0] WinNT [5/1/2001 3:34:01 PM]

  ActiveSetupLog.txt [4/20/2001 10:42:22 AM] Size: 10620 bytes
  actsetup.log [4/20/2001 12:05:02 PM] Size: 8717 bytes
  Blue Lace 16.bmp [12/6/1999 4:00:00 PM] Size: 1272 bytes
  [2] Wallpaper [5/1/2001 3:14:32 PM]
    Boiling Point.jpg [4/20/2001 8:30:24 AM] Size: 28871 bytes
    Chateau.jpg [4/20/2001 8:30:24 AM] Size: 70605 bytes
    Windows 2000.jpg [4/20/2001 8:30:24 AM] Size: 129831 bytes

8590 files in 363 directories found.
```

21.1.4 Chỉnh sửa tập tin

Đối tượng `FileInfo` có thể dùng để tạo, sao chép, đổi tên và xoá một tập tin. Ví dụ dưới đây tạo một thư mục con mới, sao chép một tập tin, đổi tên vài tập tin, và sau đó xóa toàn bộ thư mục này.

Ví dụ 21-3. Tạo thư mục con và thao tác các tập tin

```
using System;
using System.IO;

namespace Programming_CSharp
{
  class Tester
  {
    public static void Main( )
    {
      Tester t = new Tester( );
      string theDirectory = @"c:\test\media";
      DirectoryInfo dir = new DirectoryInfo(theDirectory);
      t.ExploreDirectory(dir);
    }

    private void ExploreDirectory(DirectoryInfo dir)
    {
      // tạo mới một thư mục con
      string newDirectory = "newTest";
      DirectoryInfo newSubDir =
        dir.CreateSubdirectory(newDirectory);
      // lấy tất cả các tập tin trong thư mục và
      // sao chép chúng sang thư mục mới
      FileInfo[] filesInDir = dir.GetFiles( );
      foreach (FileInfo file in filesInDir)
      {
        string fullName = newSubDir.FullName +
          "\\" + file.Name;
        file.CopyTo(fullName);
        Console.WriteLine("{0} copied to newTest",
          file.FullName);
      }
      // lấy các tập tin vừa sao chép
      filesInDir = newSubDir.GetFiles( );
      // hủy hoặc đổi tên một vài tập tin
```

```
            int counter = 0;
            foreach (FileInfo file in filesInDir)
            {
               string fullName = file.FullName;
               if (counter++ %2 == 0)
               {
                  file.MoveTo(fullName + ".bak");
                  Console.WriteLine("{0} renamed to {1}",
                     fullName,file.FullName);
               }
               else
               {
                  file.Delete( );
                  Console.WriteLine("{0} deleted.", fullName);
               }
            }
            newSubDir.Delete(true); // hủy thư mục con này
         }
      }
   }

Kết quả (một phần):
c:\test\media\Bach's Brandenburg Concerto No. 3.RMI
copied to newTest
c:\test\media\Beethoven's 5th Symphony.RMI copied to newTest
c:\test\media\Beethoven's Fur Elise.RMI copied to newTest
c:\test\media\canyon.mid copied to newTest
c:\test\media\newTest\Bach's Brandenburg Concerto
No. 3.RMI renamed to
c:\test\media\newTest\Bach's Brandenburg Concerto
No. 3.RMI.bak
c:\test\media\newTest\Beethoven's 5th Symphony.RMI deleted.
c:\test\media\newTest\Beethoven's Fur Elise.RMI renamed to
c:\test\media\newTest\Beethoven's Fur Elise.RMI.bak
c:\test\media\newTest\canyon.mid deleted.
```

21.2 Đọc và ghi dữ liệu

Đọc và ghi dữ liệu là nhiệm vụ chính của các luồng, `Stream`. `Stream` hỗ trợ cả hai cách đọc ghi đồng bộ hay bất đồng bộ. .NET Framework cung cấp sẵn nhiều lớp thừa kế từ lớp `Stream`, bao gồm `FileStream`, `MemoryStream` và `NetworkStream`. Ngoài ra còn có lớp `BufferedStream` cung cấp vùng đệm xuất nhập được dùng thêm với các luồng khác. Bảng dưới đây tóm tắt ý nghĩa sử dụng của các luồng

Bảng 21-5 Ý nghĩa các luồng

Lớp	Giải thích
Stream	Lớp trừu tượng cung cấp hỗ trợ đọc / ghi theo byte
BinaryReader / BinaryWriter	Đọc / ghi các kiểu dữ liệu gốc (primitive data type) theo trị nhị phân
File, FileInfo, Directory, DirectoryInfo	Cung cấp các cài đặt cho lớp FileSystemInfo, bao gồm việc tạo, dịch chuyển, đổi tên, xoá tập tin hay thư mục

FileStream	Để đọc từ / ghi lên tập tin. Mặc định mở tập tin đồng bộ, hỗ trợ truy cập tập tin bất đồng bộ.
TextReader, TextWriter, StringReader, StringWriter	TextReader và TextWriter là hai lớp trừu tượng được thiết kế cho việc xuất nhập ký tự Unicode.StringReader và StringWrite cài đặt hai lớp trên dành cho việc đọc ghi vào một chuỗi
BufferedStream	Luồng dùng để làm vùng đệm cho các luồng khác như NetworkStream. Lớp FileStream tự cài đặt sẵn vùng đệm. Lớp này nhằm tăng cường hiệu năng cho luồng gắn với nó.
MemoryStream	Luồng dữ liệu trực tiếp từ bộ nhớ. Thường được dùng như vùng đệm tạm.
NetworkStream	Luồng cho kết nối mạng.

21.2.1 Tập tin nhị phân

Phần này sẽ bắt đầu sử dụng lớp cơ sở `Stream` để đọc tập tin nhị phân. Lớp `Stream` có rất nhiều phương thức nhưng quan trọng nhất là năm phương thức `Read()`, `Write()`, `BeginRead()`, `BeginWrite()` và `Flush()`.

Để thao tác tập tin nhị phân (hay đọc tập tin theo kiểu nhị phân), ta bắt đầu tạo một cặp đối tượng `Stream`, một để đọc, một để viết.

```
Stream inputStream = File.OpenRead(@"C:\test\source\test1.cs");
Stream outputStream = File.OpenWrite(@"C:\test\source\test1.bak");
```

Để mở một tập tin để đọc và viết, ta sử dụng hai hàm tĩnh `OpenRead()` và `OpenWrite()` của lớp `File` với tham số là đường dẫn tập tin.

Tiếp theo ta đọc dữ liệu từ `inputStream` cho đến khi không còn dữ liệu nữa và sẽ ghi dữ liệu đọc được vào `outputStream`. Hai hàm lớp `Stream` phục vụ việc đọc ghi dữ liệu là `Read()` và `Write()`.

```
while( (bytesRead = inputStream.Read(buffer,0,SIZE_BUFF)) > 0 )
{
  outputStream.Write(buffer,0,bytesRead);
}
```

Hai hàm có cùng một số lương và kiểu tham số truyền vào. Đầu tiên là một mảng các `byte` (được gọi là vùng đệm `buffer`) dùng để chứa dữ liệu theo dang `byte`. Tham số thứ hai cho biết vị trí bắt đầu đọc hay ghi trên vùng đệm, tham số cuối cùng cho biết số byte cần đọc hay ghi. Đối với hàm `Read()` còn trả về số byte mà `Stream` đọc được, có thể bằng hay khác giá trị tham số thứ ba.

Ví dụ 21-4. Cài đặt việc đọc và ghi tập tin nhị phân

```
using System;
using System.IO;

namespace Programming_CSharp
{
  class Tester
  {
```

```
        const int SizeBuff = 1024;
        public static void Main( )
        {
           Tester t = new Tester( );
           t.Run( );
        }
        private void Run( )
        {
           // đọc từ tập tin này
           Stream inputStream = File.OpenRead(
                @"C:\test\source\test1.cs");
           // ghi vào tập tin này
           Stream outputStream = File.OpenWrite(
                @"C:\test\source\test1.bak");
           // tạo vùng đệm chứa dữ liệu
           byte[] buffer = new Byte[SizeBuff];
           int bytesRead;
           // sau khi đọc dữ liệu xuất chúng ra outputStream
        while ( (bytesRead =
             inputStream.Read(buffer,0,SizeBuff)) > 0 )
           {
             outputStream.Write(buffer,0,bytesRead);
           }
           // đóng tập tin trước khi thoát
           inputStream.Close( );
           outputStream.Close( );
        }
     }
  }
```

Kết quả sau khi chay chương trình là một bản sao của tập tin đầu vào (test1.cs) được tạo trong cùng thư mục với tên test1.bak

21.2.2 Luồng có vùng đệm

Trong ví dụ trước ta thực hiện việc ghi lên tập tin theo từng khối `buffer`, như vậy hệ điều hành sẽ thực thi việc ghi tập tin ngay sau lệnh `Write()`. Điều này có thể làm giảm hiệu năng thực thi do phải chờ các thao tác cơ học của đĩa cứng vốn rất chậm.

Luồng có vùng đệm sẽ khắc phục nhược điểm này bằng cách sau: khi có lệnh `Write()` dữ liệu, luồng sẽ không gọi hệ điều hành ngay mà sẽ giữ trên vùng đệm (thực chất là bộ nhớ), chờ cho đến khi dữ liệu đủ lớn sẽ ghi một lượt lên tập tin. Lớp `BufferedStream` là cài đặt cho luồng có vùng đệm.

Để tạo một luồng có vùng đệm trước tiên ta vẫn tạo luồng `Stream` như trên

```
Stream inputStream = File.OpenRead(@"C:\test\source\folder3.cs");
Stream outputStream = File.OpenWrite(@"C:\test\source\folder3.bak");
```

Sau đó truyền các luồng này cho hàm dựng của `BufferedStream`

```
BufferedStream bufferedInput = new BufferedStream(inputStream);
BufferedStream bufferedOutput = new BufferedStream(outputStream);
```

Từ đây ta sử dụng `bufferedInput` và `bufferedOutput` thay cho `inputStream` và `outputStream`. Cách sử dụng là như nhau: cũng dùng phương thức `Read()` và `Write()`

```
while((bytesRead = bufferedInput.Read(buffer,0,SIZE_BUFF))>0 )
{
  bufferedOutput.Write(buffer,0,bytesRead);
}
```

Có một khác biệt duy nhất là phải nhớ gọi hàm `Flush()` để chắc chắn dữ liệu đã được "tống" từ vùng `buffer` lên tập tin.

```
bufferedOutput.Flush( );
```

Lệnh này nhằm yêu cầu hệ điều hành sao chép dữ liệu từ vùng nhớ `buffer` lên đĩa cứng.

Ví dụ 21-5. Cài đặt luồng có vùng đệm

```
using System;
using System.IO;

namespace Programming_CSharp
{
  class Tester
  {
    const int SizeBuff = 1024;
    public static void Main( )
    {
      Tester t = new Tester( );
      t.Run( );
    }
    private void Run( )
    {
      // tạo một luồng nhị phân
      Stream inputStream = File.OpenRead(
        @"C:\test\source\folder3.cs");
      Stream outputStream = File.OpenWrite(
        @"C:\test\source\folder3.bak");
      // tạo luồng vùng đệm kết buộc với luồng nhị phân
    BufferedStream bufferedInput =
        new BufferedStream(inputStream);
      BufferedStream bufferedOutput =
        new BufferedStream(outputStream);
      byte[] buffer = new Byte[SizeBuff];
      int bytesRead;
      while ( (bytesRead =
        bufferedInput.Read(buffer,0,SizeBuff)) > 0 )
      {
        bufferedOutput.Write(buffer,0,bytesRead);
      }
      bufferedOutput.Flush( );
      bufferedInput.Close( );
      bufferedOutput.Close( );
    }
  }
}
```

Với tập tin có dung lượng lớn, chương trình này sẽ chạy nhanh hơn chương trình ví dụ trước.

21.2.3 Làm việc với tập tin văn bản

Đối với các tập tin chỉ chứa văn bản, ta sử dụng hai luồng `StreamReader` và `StreamWriter` cho việc đọc và ghi. Hai lớp này được thiết kế để thao tác với văn bản dễ dàng hơn. Ví dụ như chúng cung cấp hàm `ReadLine()` và `WriteLine()` để đọc và ghi một dòng văn bản.

Để tạo một thể hiện `StreamReader` ta gọi phương thức `OpenText()` của lớp `FileInfo`.

```
FileInfotheSourceFile =
new FileInfo (@"C:\test\source\test1.cs");
StreamReader stream = theSourceFile.OpenText( );
```

Ta đọc từng dòng văn bản của tập tin cho đến hết

```
do
{
   text = stream.ReadLine( );
} while (text != null);
```

Để tạo đối tượng `StreamWriter` ta truyền cho hàm khởi dựng đường dẫn tập tin

```
StreamWriter writer = new
StreamWriter(@"C:\test\source\folder3.bak",false);
```

tham số thứ hai thuộc kiểu `bool`, nếu tập tin đã tồn tại, giá trị `true` sẽ ghi dữ liệu mới vào cuối tập tin, giá trị `false` sẽ xóa dữ liệu cũ, dữ liệu mới sẽ ghi đè dữ liệu cũ.

Ví dụ 21-6. Đọc và ghi tập tin văn bản

```
using System;
using System.IO;

namespace Programming_CSharp
{
   class Tester
   {
      public static void Main( )
      {
         Tester t = new Tester( );
         t.Run( );
      }
      private void Run( )
      {
         // mở một tập tin
         FileInfo theSourceFile = new FileInfo(
            @"C:\test\source\test.cs");
         // tạo luồng đọc văn bản cho tập tin
         StreamReader reader = theSourceFile.OpenText( );
         // tạo luồng ghi văn bản cho tập tin xuất
         StreamWriter writer = new StreamWriter(
            @"C:\test\source\test.bak",false);
         // tạo một biến chuỗi lưu giữ một dòng văn bản
```

```
            string text;

            // đọc toàn bộ tập tin theo từng dòng
            // ghi ra màn hình console và tập tin xuất
            do
            {
               text = reader.ReadLine( );
               writer.WriteLine(text);
               Console.WriteLine(text);
            } while (text != null);
            // đóng tập tin
            reader.Close( );
            writer.Close( );
         }
      }
   }
```

Khi thực thi chương trình nội dung tập tin nguồn được ghi lên tập tin mới đồng thời xuất ra màn hình `console`.

21.3 Bất đồng bộ nhập xuất

Các ví dụ được trình bày ở trên sử dụng kỹ thuật đồng bộ hóa trong nhập xuất dữ liệu (synchronous I/O), có nghĩa là chương trình sẽ tạm ngưng trong lúc hệ điều hành thực hiện việc đọc hay ghi dữ liệu. Điều này có thể làm chương trình tốn thời gian vô ích, đặc biệt khi làm việc với các ổ đĩa có tốc độ chậm hay dung lượng đường truyền mạng thấp.

Kỹ thuật bất đồng bộ nhập xuất (asynchronous I/O) được dùng để giải quyết vấn đề này. Ta có thể thực hiện các công việc khác trong khi chờ hệ thống hập xuất đọc/ghi dữ liệu. Kỹ thuật này được cài đặt trong phương thức `BeginRead()` và `BeginWrite()` của lớp `Stream`.

Mấu chốt của phương thức `Begin*()` là khi được gọi một tiểu trình mới sẽ được tạo và làm công việc nhập xuất, tiểu trình cũ sẽ thực hiện công việc khác. Sau khi hoàn tất việc đọc/ghi, thông báo được gởi đến hàm `callback` thông qua một `deleagte`. Ta có thể thao tác với các dữ liệu vừa được đọc/ghi, thực hiện một công việc đọc/ghi khác và lại quay đi làm công việc khác.

Phương thức `BeginRead()` yêu cầu năm tham số, ba tham số tương tự hàm `Read`, hai tham số (tùy chọn) còn lại là: `delegate AsyncCallback` để gọi hàm `callback` và tham số còn lại là `object` dùng để phân biệt giữa các thao tác nhập xuất bất đồng bộ khác nhau.

Trong ví dụ dụ này ta sẽ tạo một mảng `byte` làm vùng đệm, và một đối tượng `Stream`

```
public class AsynchIOTester
{
   private Stream inputStream;
   private byte[] buffer;
   const int BufferSize = 256;
```

một biến thành viên kiểu `delegate` mà phương thức `BeginRead()` yêu cầu

```
private AsyncCallback myCallBack; // delegated method
```

`Delegate AsyncCallback` khai báo trong vùng tên `System` như sau

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Tạo một hàm `callback` để đóng gói trong `delegate`

```
void OnCompletedRead(IAsyncResult asyncResult)
```

Dưới đây là cách hoạt động của ví dụ. Trong hàm `Main()` ta khởi tạo và cho thực thi lớp kiểm thử `AsyncIOTester`

```
public static void Main( )
{
  AsynchIOTester theApp = new AsynchIOTester( );
  theApp.Run( );
}
```

Hàm dựng khởi tạo các biến thành viên

```
AsynchIOTester( )
{
  inputStream = File.OpenRead(@"C:\test\source\AskTim.txt");
  buffer = new byte[BufferSize];
  myCallBack = new AsyncCallback(this.OnCompletedRead);
}
```

Phương thức `Run()` sẽ gọi `BeginRead()`

```
inputStream.BeginRead(
            buffer, // chứa kết quả
            0, // vị trí bắt đầu
            buffer.Length, // kích thước vùng đệm
            myCallBack, // callback delegate
            null); // đối tượng trạng thái
```

Sau đó thực hiện công việc khác, trường hợp này là vòng lặp `for` thực hiện 500.000 lần.

```
for (long i = 0; i < 500000; i++)
{
  if (i%1000 == 0)
  {
    Console.WriteLine("i: {0}", i);
  }
}
```

Sau khi việc đọc hoàn tất hàm `callback` được gọi

```
void OnCompletedRead(IAsyncResult asyncResult)
{
```

Điều đầu tiên là phải biết số lượng byte thật sự đọc được bằng cách gọi hàm `EndRead()`

```
  int bytesRead = inputStream.EndRead(asyncResult);
```

Sau đó thao tác trên dữ liệu đọc được (in ra `console`), và lại gọi tiếp một `BeginRead()` để thực hiện nhập xuất bất đồng bộ một lần nữa,

```
  if (bytesRead > 0)
```

```
    {
      string s =  Encoding.ASCII.GetString (buffer, 0, bytesRead);
      Console.WriteLine(s);
      inputStream.BeginRead(buffer, 0, buffer.Length,
                            myCallBack, null);
  }
```

Hiệu quả của chương trình là ta có thể thực hiện các công việc không cần kết quả của việc đọc dữ liệu khác. Ví dụ hoàn chỉnh lệit kê dưới đây

Ví dụ 21-7. cài đặt nhập xuất bất đồng bộ

```
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace Programming_CSharp
{
  public class AsynchIOTester
  {
    private Stream inputStream;
    // delegated
    private AsyncCallback myCallBack;
    // vùng nhớ buffer lưu giữ liệu đọc được
    private byte[] buffer;
    // kích thước buffer
    const int BufferSize = 256;

    AsynchIOTester( )
    {
      // mở một luồng nhập
      inputStream = File.OpenRead(
                    @"C:\test\source\AskTim.txt");
      // cấp phát vùng buffer
      buffer = new byte[BufferSize];
      // gán một hàm callback
       myCallBack = new AsyncCallback(this.OnCompletedRead);
    }
    public static void Main( )
    {
      AsynchIOTester theApp =    new AsynchIOTester();
      theApp.Run( );
    }

    void Run()
    {
      inputStream.BeginRead(
                buffer, // chứa kết quả
                0, // vị trí bắt đầu trên buffer
                buffer.Length, // kích thước buffer
                myCallBack, // callback delegate
                null); // đối tượng trạng thái cục bộ
      // làm chuyện gì đó trong lúc đọc dữ liệu
      for (long i = 0; i < 500000; i++)
      {
        if (i%1000 == 0)
        {
```

```
            Console.WriteLine("i: {0}", i);
          }
        }
      }

      // hàm callback
      void OnCompletedRead(IAsyncResult asyncResult)
      {
        int bytesRead =
          inputStream.EndRead(asyncResult);
        // nếu đọc có dữ liệu
        if (bytesRead > 0)
        {
        // chuyển nó thành chuỗi
          String s =
            Encoding.ASCII.GetString(buffer, 0, bytesRead);
          Console.WriteLine(s);
          inputStream.BeginRead(
                 buffer, 0, buffer.Length, myCallBack, null);
        }
      }
    }
}

Kết quả (một phần)
i: 47000
i: 48000
i: 49000
Date: January 2001
From: Dave Heisler
To: Ask Tim
Subject: Questions About O'Reilly
Dear Tim,
I've been a programmer for about ten years. I had heard of
O'Reilly books,then...
Dave,
You might be amazed at how many requests for help with
school projects I get;
i: 50000
i: 51000
i: 52000
```

Trong các ứng dụng thực tế, ta sẽ tương tác với người dùng hoặc thực hiện các tính toán trong khi công việc nhập xuất tập tin hay cơ sở dữ liệu được thực hiện một cách bất đồng bộ ở một tiểu trình khác.

21.4 Serialization

Serialize có nghĩa là sắp theo thứ tự. Khi ta muốn lưu một đối tượng xuống tập tin trên đĩa từ để lưu trữ, ta phải định ra trình tự lưu trữ của dữ liệu trong đối tượng. Khi cần tái tạo lại đối tượng từ thông tin trên tập tin đã lưu trữ, ta sẽ "nạp" đúng theo trình tự đã định trước đó. Đây gọi là quá trình `Serialize`.

Nói chính xác hơn, `serialize` là tiến trình biến đổi trạng thái của đối tượng theo một định dạng có thể được lưu trữ hay dịch chuyển (transfer).

.NET Framework cung cấp 2 kỹ thuật `serialize`:

- Binary serialize (serialize nhị phân): cách này giữ nguyên kiểu dữ liệu, thích hợp cho việc giữ nguyên cấu trúc đối tượng. Có thể dùng kỹ thuật này để chia sẻ đối tương giữa các ứng dụng bằng cách serialize vào vùng nhớ clipboard; hoặc serialize vào các luồng, đĩa từ, bộ nhớ, trên mạng …; hoặc truyền cho máy tính ở xa như một tham trị ("by-value object")
- XML và SOAP Serialize: chỉ serialize các thuộc tính public, và không giữ nguyên kiểu dữ liệu. Tuy nhiên XML và SOAP là các chuẩn mở nên kỹ thuật không bị các hạn chế về giao tiếp giữa các ứng dụng.

Các đối tượng cớ sở đều có khả năng `serialize`. Để đối tượng của ta có thể `serialize`, trước tiên cần thêm khai báo `attribute [Serialize]` cho lớp đối tượng đó. Nếu đối tượng có chứa các đối tượng khác thì các đối tượng đó phải có khả năng `serialize`.

21.4.1 Làm việc với Serialize

Trước tiên, ta tạo một đối tượng `Sumof` làm ví dụ cho việc `Serialize`. Đối tượng có các biến thành viên sau:

```
private int startNumber = 1;
private int endNumber;
private int[] theSums;
```

mảng `theSums` được mô tả: phần tử `theSum[i]` chứa giá trị là tổng từ `startNumber` cho đến `startNumber + i.`

21.4.1.1 serialize đối tượng

Trước tiên thêm `attribute [Serialize]` vào trước khai báo đối tượng

```
[Serializable]
class SumOf
```

Ta cần một tập tin để lư trữ đối tượng này, tạo một `FileStream`

```
FileStream fileStream = new FileStream("DoSum.out",FileMode.Create);
```

Sau khi tạo một `Formatter`, gọi phương thức `Serialize` của nó.

```
binaryFormatter.Serialize(fileStream,this);
```

Đối tượng `Sumof` đã được `Serialize`.

21.4.1.2 Deserialize đối tượng

`Deserialize` là tiến trình ngược với `serialize`, tiến trình này đọc dữ liệu được `serialize` để tái tạo lại đối tượng.

Khai báo phương thức tĩnh `DeSerialize` cho tiến trình này

```
public static SumOf DeSerialize( )
{
  FileStream fileStream = new FileStream("DoSum.out",FileMode.Open);
  BinaryFormatter binaryFormatter = new BinaryFormatter( );
```

```
    return (SumOf) binaryFormatter.Deserialize(fileStream);
    fileStream.Close( );
  }
```

Ví dụ 21-1 Serialize và Deserialize đối tượng

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace Programming_CSharp
{
  [Serializable]
  class SumOf
  {
    public static void Main( )
    {
      Console.WriteLine("Creating first one with new...");
      SumOf app = new SumOf(1,10);
      Console.WriteLine("Creating second one with
                    deserialize...");
      SumOf newInstance = SumOf.DeSerialize( );
      newInstance.DisplaySums( );
    }

    public SumOf(int start, int end)
    {
      startNumber = start;
      endNumber = end;
      ComputeSums( );
      DisplaySums( );
      Serialize( );
    }

    private void ComputeSums( )
    {
      int count = endNumber - startNumber + 1;
      theSums = new int[count];
      theSums[0] = startNumber;
      for (int i=1,j=startNumber + 1;i<count;i++,j++)
      {
        theSums[i] = j + theSums[i-1];
      }
    }

    private void DisplaySums( )
    {
      foreach(int i in theSums)
      {
        Console.WriteLine("{0}, ",i);
      }
    }

    private void Serialize( )
    {
      Console.Write("Serializing...");
      // tạo một file stream để đọ hay ghi
```

```
        FileStream fileStream =
          new FileStream("DoSum.out",FileMode.Create);
        //sử dung binary formatter
        BinaryFormatter binaryFormatter =
          new BinaryFormatter( );
        // serialize
        binaryFormatter.Serialize(fileStream,this);
        Console.WriteLine("...completed");
        fileStream.Close( );
      }

      public static SumOf DeSerialize( )
      {
        FileStream fileStream =
          new FileStream("DoSum.out",FileMode.Open);
        BinaryFormatter binaryFormatter =
          new BinaryFormatter( );
        return (SumOf) binaryFormatter.Deserialize(fileStream);
        fileStream.Close( );
      }

      private int startNumber = 1;
      private int endNumber;
      private int[] theSums;
    }
}

Kết quả:
Creating first one with new...
1,
3,
6,
10,
15,
21,
28,
36,
45,
55,
Serializing......completed
Creating second one with deserialize...
1,
3,
6,
10,
15,
21,
28,
36,
45,
55,
```

21.4.2 Handling Transient Data

Theo cách nhìn nào đó thì `serialize` kiểu Ví dụ 21-1 rất lãng phí. Giá trị các phần tử trong mảng có thể tính bằng thuật toán vì vậy không nhất thiết phải `serialize` mảng này (và làm giảm đáng kể dung lượng tập tin lưu trữ).

Để CLR biết ta không muốn `Serialize` biến thành viên này, ta đặt `attribute [NonSerialize]` trước khai báo:

```
[NonSerialized] private int[] theSums;
```

Theo logic, khi `deserialize`, ta không thể có ngay mảng và vì vậy cần thực hiện lại công việc tính toán một lần nữa. Ta có thể thực hiện trong hàm `Deserialize`, nhưng CLR cung cấp giao diện `IDeserializationCallback`, ta sẽ cài đặt giao diện này

```
[Serializable]
class SumOf : IDeserializationCallback
```

Giao diện này có một phương thức duy nhất là `OnDeserialization()` mà ta phải cài đặt:

```
public virtual void OnDeserialization (Object sender)
{
  ComputeSums( );
}
```

Khi tiến trình `Deserialize`, phương thức này sẽ được gọi và mảng `theSums` được tính toán và khởi gán. Cái giá mà ta phải trả chính là thời gian dành cho việc tính toán này.

Ví dụ 21-2 Làm việc với dối tượng nonserialize

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace Programming_CSharp
{
  [Serializable]
  class SumOf : IDeserializationCallback
  {
    public static void Main( )
    {
      Console.WriteLine("Creating first one with new...");
      SumOf app = new SumOf(1,5);
      Console.WriteLine("Creating second one with
                                deserialize...");
      SumOf newInstance = SumOf.DeSerialize( );
      newInstance.DisplaySums( );
    }

    public SumOf(int start, int end)
    {
      startNumber = start;
      endNumber = end;
      ComputeSums( );
      DisplaySums( );
      Serialize( );
    }

    private void ComputeSums( )
    {
```

```
      int count = endNumber - startNumber + 1;
      theSums = new int[count];
      theSums[0] = startNumber;
      for (int i=1,j=startNumber + 1;i<count;i++,j++)
      {
        theSums[i] = j + theSums[i-1];
      }
    }

    private void DisplaySums( )
    {
      foreach(int i in theSums)
      {
        Console.WriteLine("{0}, ",i);
      }
    }

    private void Serialize( )
    {
      Console.Write("Serializing...");
      FileStream fileStream =
        new FileStream("DoSum.out",FileMode.Create);
      BinaryFormatter binaryFormatter = new BinaryFormatter();
      binaryFormatter.Serialize(fileStream,this);
      Console.WriteLine("...completed");
      fileStream.Close( );
    }

    public static SumOf DeSerialize( )
    {
      FileStream fileStream =
        new FileStream("DoSum.out",FileMode.Open);
      BinaryFormatter binaryFormatter =
        new BinaryFormatter( );
      return (SumOf) binaryFormatter.Deserialize(fileStream);
      fileStream.Close( );
    }

    public virtual void OnDeserialization( Object sender )
    {
      ComputeSums( );
    }

    private int startNumber = 1;
    private int endNumber;
    [NonSerialized] private int[] theSums;
  }
}
```

Kết quả:

```
Creating first one with new...
1,
3,
6,
10,
15,
Serializing......completed
Creating second one with deserialize...
```

```
1,
3,
6,
10,
15,
```

21.5 Isolate Storage

Outlook Express (OE) là trình nhận/chuyển thư điện tử của Microsoft. Khi chạy trên môi trường đa người dùng như Windows 2000, nó cung cấp cho mỗi người dùng một hộp thư riêng. Các hộp thư này lưu trữ trên đĩa cứng thành nhiều tập tin khác nhau ở các thư mục thuộc quyền của người dùng tương ứng. Ngoài ra OE còn lưu giữ cảc các thiết đặt (như các cửa sổ hiển thị, tài khoản kết nối …) của từng người dùng.

.NET Framework cung cấp các lớp thực hiện các công việc này. Nó tương tự như các tập tin .ini của Windows cũ, hay gần đây hơn là khóa HKEY_CURRENT_USER trong Registry. Lớp thực hiện việc này là luồng `IsolatedStorageFileStream`. Cách sử dụng tương tự như các luồng khác. Ta khởi tạo bằng cách truyền cho hàm dựng tên tập tin, các công việc khác hoàn toàn do luồng thực hiện.

Ví dụ 21-3 Isolated Storage

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

namespace Programming_CSharp
{
  public class Tester
  {
    public static void Main( )
    {
      Tester app = new Tester( );
      app.Run( );
    }
    private void Run( )
    {
      // tạo một luồng cho tập tin cấu hình
      IsolatedStorageFileStream configFile = new
      IsolatedStorageFileStream("Tester.cfg",FileMode.Create);
      // tạo một writer để ghi lên luồng
      StreamWriter writer = new StreamWriter(configFile);
      // ghi dữ liệu lr6n tập tin config
      String output;
      System.DateTime currentTime = System.DateTime.Now;
      output = "Last access: " + currentTime.ToString( );
      writer.WriteLine(output);
      output = "Last position = 27,35";
      writer.WriteLine(output);
      // tống sạch dữ liệu
      writer.Flush( );
      writer.Close( );
```

```
            configFile.Close( );
        }
    }
}
```

Sau khi chạy đoạn mã này ta, thực hiện việc tìm kiếm tập tin `test.cfg`, ta sẽ thấy nó trong đường dẫn sau:

```
c:\Documents and Settings\Administrator\ApplicationData\
Microsoft\COMPlus\IsolatedStorage\0.4\
Url.wj4zpd5ni41dynqxx1uz0x0aoaraftc\
Url.wj4zpd5ni41dynqxx1uz0ix0aoaraftc\files
```

Mở tập tin này bằng Notepad, nội dung tập tin như sau

```
Last access: 5/2/2001 10:00:57 AM
Last position = 27,35
```

Ta cũng có thể đọc tập tin này bằng chính luồng IsolatedStorageFileStream

Chương 22 Lập trình .NET và COM

Chương này nói về những điều còn lại của C# (và .NET Framework).

Khi xây dựng và công bố chuẩn OLE 2.0, sau đó là COM và ActiveX, Microsoft đã quảng cáo một cách rầm rộ về "khả năng" lập trình các thành phần, sau đó gắn chúng lại để có các ứng dụng. Bên cạnh đó là khả năng viết một lần dùng cho tất cả ngôn ngữ của COM. Tuy nhiên COM vẫn vướng mắc một số hạn chế như vấn đề phiên bản và khá "khó nuốt".

.NET Framework mới ra đời lại mang một kiến trúc khác, không hạn chế về ngôn ngữ, giải quyết "xong" vấn đề phiên bản. Tuy nhiên trong các công ty hiện nay vẫn còn "vô số COM", và .NET Framework buộc phải tiếp tục hỗ trợ COM. Dưới đây là các vấn đề mà .NET Framework giải quyết được:

- Hiểu và cho phép sử dụng các ActiveX control trong môi trường Vs.NET
- Hiểu và cho phép sử dụng các đối tượng COM
- Cho phép chuyển một lớp .NET thành một COM

Ngoài ra, như đã giới thiệu C# hỗ trợ kiểu con trỏ của C++ với mục đích có được sự mềm dẻo của C/C++. Kiểu con trỏ được khuyên không nên sử dụng vì đoạn mã dùng con trỏ được xem là không an toàn. Nó chỉ thích hợp cho các thao tác với các COM, các thư viện hàm DLL, hay gọi trực tiếp đến các Win API.

22.1 P/Invoke

Khởi đầu Platform invoke facility (P/Invoke - dễ dàng gọi các giao diện lập trình của hệ điều hành/sàn diễn) được dự định cung cấp một cách thức để truy cập đến các hàm Windows API, nhưng ta có thể dùng nó để gọi các hàm thư viện DLL.

Ví dụ sắp trình bày sử dụng hàm Win API MoveFile của thư viên kernal32.dll. Ta khai báo phương thức static extern bằng attribute DllImport như sau:

```
[DllImport("kernel32.dll", EntryPoint="MoveFile",
ExactSpelling=false, CharSet=CharSet.Unicode,
SetLastError=true)]
static extern bool MoveFile(
string sourceFile, string destinationFile);
```

Lớp DllImport (cũng là lớp DllImportAttribute) để chỉ ra một phương thức không được quản lý (unmanaged) được gọi thông qua P/Invoke. Các tham số được giải thích như sau:

EntryPoint: Tên hàm được gọi
ExactSpelling: đặt giá trị false để không phân biệt hoa thường
CharSet: tập ký tự thao tác trên các tham số kiểu chuỗi

SetLastError: đặt giá trị true để được phép gọi hàm GetLastError (Win API) kiểm tra lỗi

Ví dụ 22-1 Sử dụng P/Invoke để gọi WinAPI

```
using System;
using System.IO;
using System.Runtime.InteropServices;

namespace Programming_CSharp
{
  class Tester
  {
    // khai báo hàm WinAPI muốn gọi P/Invoke
    [DllImport("kernel32.dll", EntryPoint="MoveFile",
       ExactSpelling=false, CharSet=CharSet.Unicode,
       SetLastError=true)]
    static extern bool MoveFile( string sourceFile,
                                 string destinationFile);

    public static void Main( )
    {
      Tester t = new Tester( );
      string theDirectory = @"c:\test\media";
      DirectoryInfo dir = new DirectoryInfo(theDirectory);
      t.ExploreDirectory(dir);
    }

    private void ExploreDirectory(DirectoryInfo dir)
    {
      string newDirectory = "newTest";
      DirectoryInfo newSubDir =
                    dir.CreateSubdirectory(newDirectory);
      FileInfo[] filesInDir = dir.GetFiles( );
      foreach (FileInfo file in filesInDir)
      {
        string fullName = newSubDir.FullName +
             "\\" + file.Name;
        file.CopyTo(fullName);
        Console.WriteLine("{0} copied to newTest",
             file.FullName);
      }

      filesInDir = newSubDir.GetFiles( );
      // xóa một vài tập tin và
      // đổi tên một vài tập tin
      int counter = 0;
      foreach (FileInfo file in filesInDir)
      {
        string fullName = file.FullName;
        if (counter++ %2 == 0)
        {
          // P/Invoke Win API
          Tester.MoveFile(fullName, fullName + ".bak");
          Console.WriteLine("{0} renamed to {1}",
            fullName,file.FullName);
        }
        else
```

```
            {
               file.Delete( );
               Console.WriteLine("{0} deleted.", fullName);
            }
         }

         newSubDir.Delete(true);
      }
   }
}

Kết quả (một phần):
c:\test\media\newTest\recycle.wav renamed to
c:\test\media\newTest\recycle.wav
c:\test\media\newTest\ringin.wav renamed to
c:\test\media\newTest\ringin.wav
```

Một lần nữa, chỉ nên gọi P/Invoke trong trường bất khả kháng. Sử dụng các lớp .NET Framework để có đoạn mã được quản lý.

22.2 Con trỏ

Như đã đề cập ở trên, chỉ nên sử dụng con trỏ khi làm việc với các COM, WinAPI, hàm DLL.

Các toán tử sử dụng với con trỏ tương tự như C/C++

```
&: toán tử lấy địa chỉ
*: toán tử lấy nội dung con trỏ
->: toán tử đến các thành viên của con trỏ
```

Ví dụ dưới đây sử dụng con trỏ làm tham số cho hai hàm WinAPI CreatFile và ReadFile.

Ví dụ 22-2 Sử dụng con trỏ trong C#

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class APIFileReader
{
   // import hai phương thức, phải có từ khóa unsafe
   [DllImport("kernel32", SetLastError=true)]
   static extern unsafe int CreateFile(
      string filename,
      uint desiredAccess,
      uint shareMode,
      uint attributes,
      uint creationDisposition,
      uint flagsAndAttributes,
      uint templateFile);

   // API phải dùng con trỏ
   [DllImport("kernel32", SetLastError=true)]
   static extern unsafe bool ReadFile(
      int hFile,
      void* lpBuffer,
```

```
      int nBytesToRead,
      int* nBytesRead,
        int overlapped);

    // hàm dựng: mở một tập tin đã tồn tại
    public APIFileReader(string filename)
    {
      fileHandle = CreateFile(
        filename, // tập tin
        GenericRead, // cách truy xuất - desiredAccess
        UseDefault, // shareMode
        UseDefault, // attributes
        OpenExisting, // creationDisposition
        UseDefault, // flagsAndAttributes
        UseDefault); // templateFile
    }

    // unsafe: cho phép tạo con trỏ và
    //          ngữ cảnh unsafe (unsafe context)
    public unsafe int Read(byte[] buffer, int index, int count)
    {
      int bytesRead = 0;
      // fixed: cấm CLR dọn dẹp rác
      fixed (byte* bytePointer = buffer)
      {
        ReadFile(
          fileHandle, // hfile
          bytePointer + index, // lpBuffer
          count, // nBytesToRead
          &bytesRead, // nBytesRead
          0); // overlapped
      }
      return bytesRead;
    }
    const uint GenericRead = 0x80000000;
    const uint OpenExisting = 3;
    const uint UseDefault = 0;
    int fileHandle;
}
class Test
{
    public static void Main( )
    {
      APIFileReader fileReader =
             new APIFileReader("myTestFile.txt");

      // tạo buffer và ASCII coder
      const int BuffSize = 128;
      byte[] buffer = new byte[BuffSize];
      ASCIIEncoding asciiEncoder = new ASCIIEncoding( );

      // đọc tập tin vào buffer và hiển thị ra màn hình console
      while (fileReader.Read(buffer, 0, BuffSize) != 0)
      {
        Console.Write("{0}", asciiEncoder.GetString(buffer));
      }
    }
}
```

Phần 2

Xây dựng một ứng dụng minh họa

Chương 23 Website dạy học ngôn ngữ C#

23.1 Hiện trạng và yêu cầu

Trước tiên chúng ta sẽ tìm hiểu sơ qua về những gì đang diễn ra trong thực tế, và ứng dụng của ta liên quan đến khía cạnh nào. Sau đó ta phải xác định rõ các yêu cầu mà ứng dụng cần phải thực hiện. Việc xác định thật rõ và đúng các yêu cầu mà ứng dụng cần phải thực hiện là bước rất quan trọng, nó sẽ định hướng cho toàn bộ ứng dụng của chúng ta.

23.1.1 Hiện trạng thực tế

23.1.1.1 Hiện trạng

Hiện nay, lĩnh vực công nghệ thông tin trên toàn thế giới đang phát triển hết sức nhanh chóng cả về hướng công nghệ phần mềm và lẫn hướng công nghệ phần cứng. Chỉ cần một vài tháng là sẽ có rất nhiều thay đổi, vì thế ta cần phải có một phương pháp tốt để tiếp cận chúng.

Mặc dù có rất nhiều công cụ, ngôn ngữ giúp các nhà phát triển phần mềm tạo ra hàng loạt các ứng dụng mạnh mẽ, nhưng giường như chưa đủ. Họ vẫn luôn muốn tìm tòi những cái mới, công cụ tốt hơn để có thể tăng hiệu suất phát triển phần mềm thật nhanh và thật hiệu quả. Một số tổ chức cung cấp các bộ phát triển phần mềm nổi tiếng như :

1. Microsoft với hệ điều hành Windows, bộ Visual Studio 6.0 với các ngôn ngữ lập trình như : Visual Basic, Visual C++ …
2. Tổ chức Sun với ngôn ngữ Java đã từng nổi tiếng một thời, thống trị trong các ứng dụng Web.

   Những năm đầu của thế kỷ 21, năm 2000 – 2002. Micrsoft đã tung ra thị trường một công nghệ mới **Microsoft Development Enviroment .NET** với mục đích :
3. Đánh bại các đối thủ khác : ngôn ngữ lập trình Java của Sun hay hệ quản trị cơ sở dữ liệu Oracle …
4. Trở thành công cụ mạnh nhất để phát triển các ứng dụng Web (chữ NET viết tắt của Network).

Nhằm minh họa quá trình tìm hiểu ngôn ngữ C# (đọc là Csharp) trong bộ công cụ .NET, chúng tôi đã viết nên ứng dụng Web dạy học C# này.

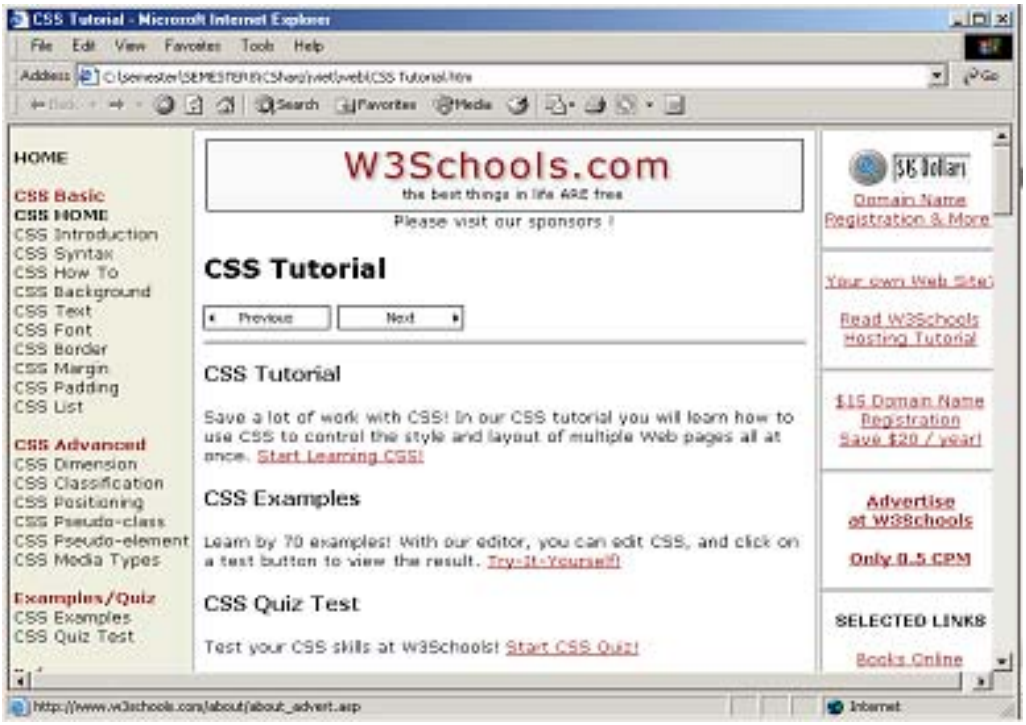
23.1.1.2 Quá trình tìm hiểu thực tế

Để ứng dụng phù hợp với thực tế và xác định rõ được các yêu cầu mà ứng dụng cần thực hiện, chúng tôi cũng đã tìm hiểu qua một số Web-Site dạy học trên mạng. Sau đây là một số hình ảnh minh họa quá trình tìm hiểu :

Trang chủ dạy học chủ đề CSS

Trang này sẽ liệt kê tất cả các mục thuộc chủ đề này, đồng thời trang này cũng cho phép các liên kết (Link) tới các trang con khác : các tham chiếu tới các địa chỉ khác có liên quan, trắc nghiệm của chủ đề và minh họa lý thuyết qua các ví dụ nếu có.

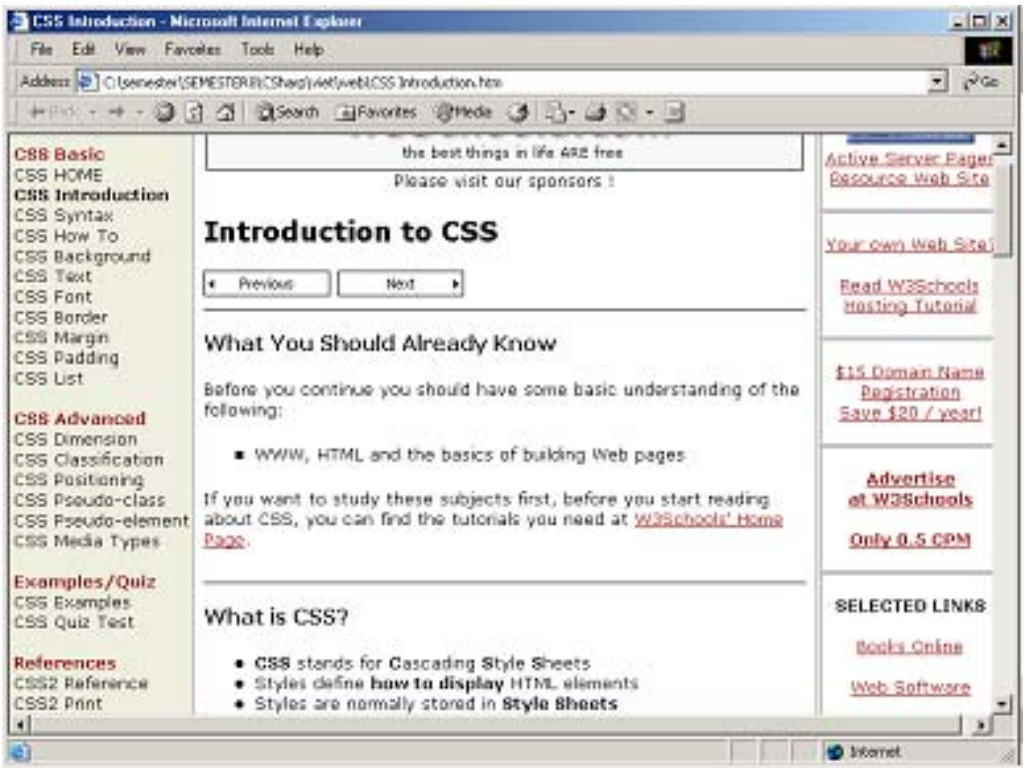
Hình 23-1 Trang Chủ dạy học ngôn ngữ CSS



Trang hiển thị lý thuyết của chương thuộc chủ đề

Trong một chủ đề sẽ có nhiều chương. Khi chọn một chương nào đó, thì sẽ hiển thị phần lý thuyết chính mô tả cho chương đó.

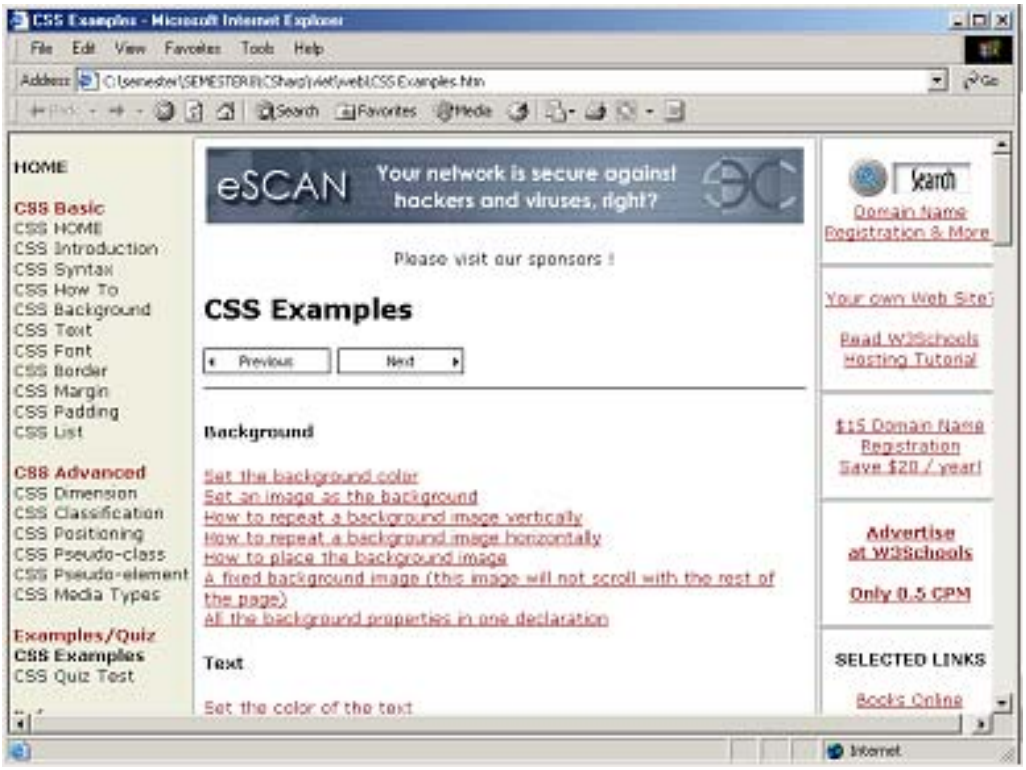
Hình 23-2 Trang giới thiệu về chương : Introduction CSS



Liệt kê các ví dụ minh họa lý thuyết thuộc chủ đề

Phần này sẽ liệt kê tất cả các ví dụ hiện có thuộc chủ đề theo từng nhóm cụ thể.

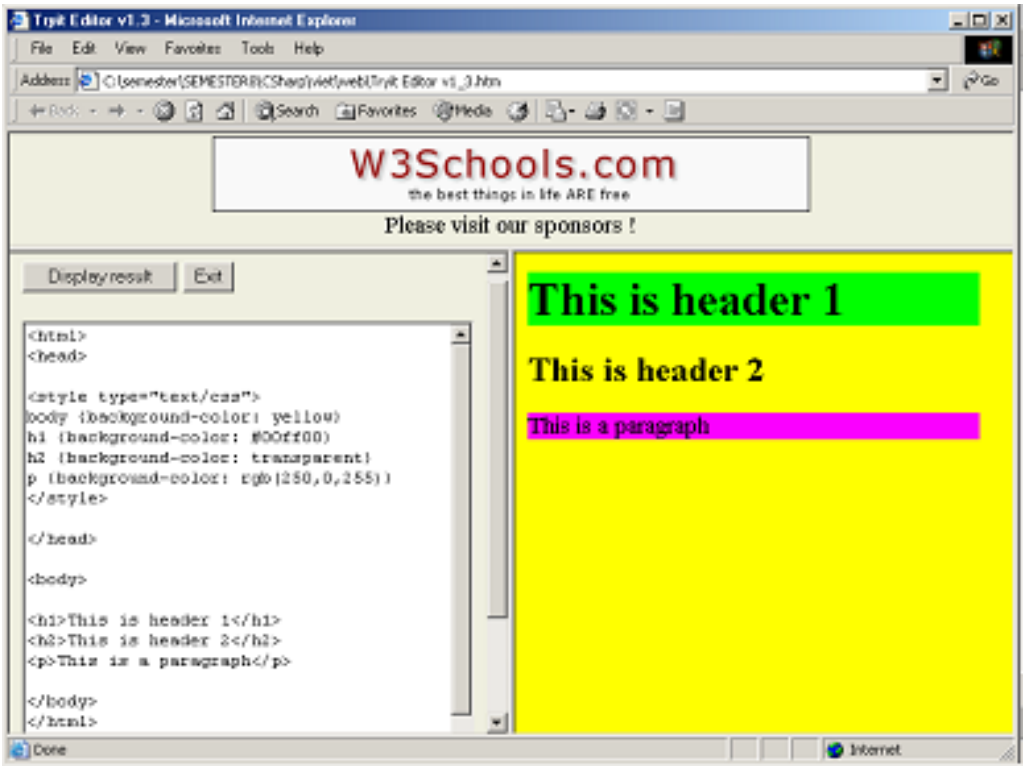
Hình 23-3 Liệt kê các ví dụ minh họa lý thuyết thuộc chủ đề theo nhóm



Minh họa lý thuyết qua ví dụ

Sau khi tìm hiểu lý thuyết, người dùng muốn tìm hiểu rõ hơn lý thuyết qua phần mã nguồn của các ví dụ thuộc chủ đề đó. Tùy từng loại chủ đề mà người dùng được hỗ trợ chức năng gõ mã tiếp vào cửa sổ Text, sau đó sẽ gửi trang mã này lên máy chủ để nhận được kết quả của phần mã vừa gõ vào.

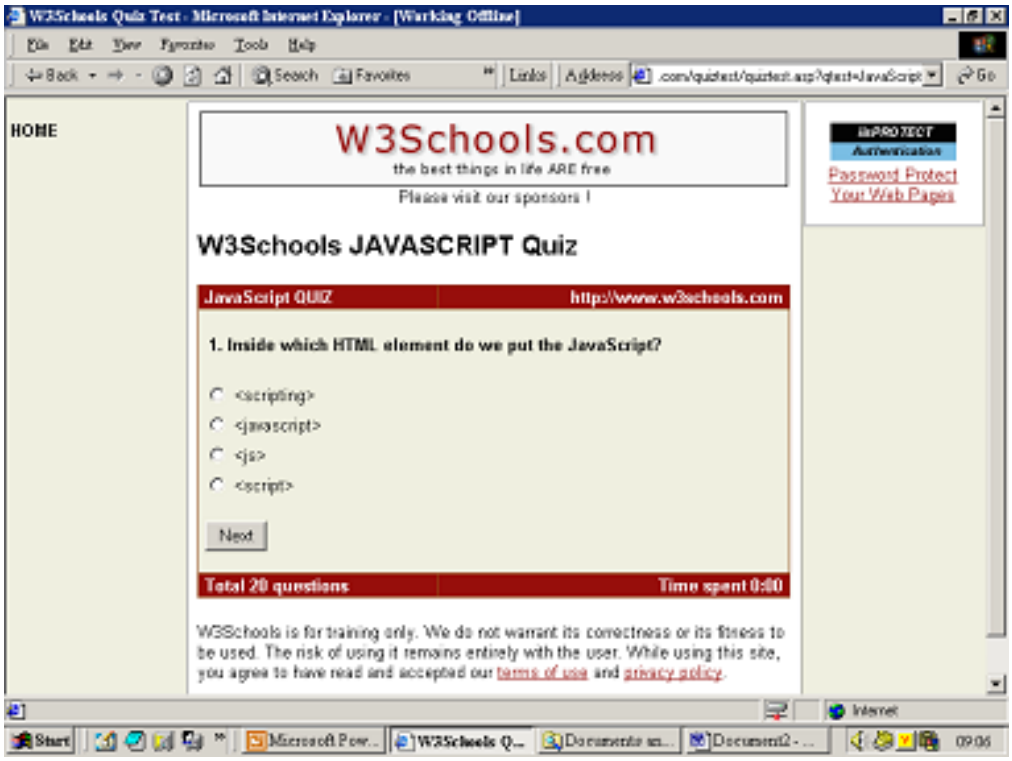
Hình 23-4 Minh họa lý thuyết qua ví dụ, cho phép người dùng tự gõ mã của họ vào



Tổ chức thi trắc nghiệm cho chủ đề

Trong mỗi chủ đề, người dùng có thể tham gia thi trắc nghiệm kiến thức của mình :

Hình 23-5 Tổ chức thi trắc nghiệm cho chủ đề



23.1.2 Xác định yêu cầu

Ứng dụng phải giúp người dùng có thể nắm bắt được kiến thức một cách nhanh nhất, tránh tràn lan, dài dòng và phải hỗ trợ chức năng hiệu chỉnh (Admin).

23.1.2.1 Yêu cầu chức năng

Ứng dụng dự định sẽ có 5 chức năng chính sau :

1. Hiển thị lý thuết
2. Minh họa lý thuyết qua ví dụ
3. Tổ chức thi trắc nghiệm
4. Cho phép quảng cáo, giới thiệu sách và Link tới các Site liên quan khác
5. **Cho phép chức năng hiệu chỉnh (thêm, xóa, sửa) các thành phần trên**

Mô tả chi tiết về các yêu cầu chức năng trên :

Lưu trữ

- Lý thuyết về từng chủ đề, các chương thuộc chủ đề và từng mục chính, ý chính của chương, các tập tin đính kèm.
- Các ví dụ minh họa của mỗi chủ đề, tập tin đính kèm nếu có.

- Danh sách các bài trắc nghiệm và các câu hỏi, không lưu trữ bài làm
- Các địa chỉ Web-Site để tham chiếu, các hình ảnh và thông tin về việc quảng cáo sách ứng với mỗi chủ đề.
- Một số thông tin khác : thông tin người dùng Admin, các tham số hiệu chỉnh …

Tra cứu

- Thông tin về mỗi chủ đề, chương, mục chính và các ý chính thuộc chương.
- Thông tin về các bài ví dụ minh họa lý thuyết.
- Thông tin về các bài thi trắc nghiệm.
- Một số thông tin khác liên quan nếu cần thiết.

Tính toán

- Số câu đúng cho các bài trắc nghiệm và tính điểm cho chúng.

Kết xuất

- Hiển thị theo cấu trúc phân cấp dạng cây về lý thuyết theo chủ đề, chương, mục chính, ý chính.
- Ví dụ minh họa lý thuyết
- Các bài trắc nghiệm và kết quả tương ứng
- Danh sách các tham chiếu đến các Web-Site, các hình ảnh và thông tin về sách cần quảng cáo.
- Các thông tin về lý thuyết, ví dụ, trắc nghiệm, sách cần để hiệu chỉnh.
- Các báo cáo thống kê về các mục trên nếu có.

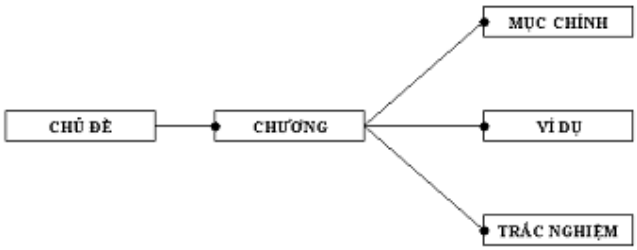
23.1.2.2 Yêu cầu chất lượng

2. Ứng dụng phải hiển thị thông tin một cách súch tích, ngắn gọn, trực quan đối với người dùng, phải giúp người dúng nắm bắt thông tin được nhanh nhất.
3. Phải cho phép hiệu chỉnh (Admin) trực tiếp trên ứng dụng.
4. Nội dung phải chất lượng, phù hợp với từng chủ đề, dễ hiểu.

23.2 Phân tích hướng đối tượng

23.2.1 Sơ đồ lớp đối tượng

Hình 23-6 Lớp đối tượng



23.2.2 Ý nghĩa các đối tượng chính

23.2.2.1 CHỦ ĐỀ

Ứng dụng có thể có nhiều chủ đề, mỗi chủ đề sẽ lưu trữ các thông tin về chính nó như : các thành phần thuộc chủ đề, giới thiệu về chủ đề, có nhiều chương.

23.2.2.2 CHƯƠNG

Chương thuộc về một chủ đề nào đó, mỗi chương có một hay nhiều mục chính, ví dụ, trắc nghiệm.

23.2.2.3 MỤC CHÍNH

Mục chính thuộc về một chương nào đó, mỗi mục chính có một hay nhiều ý chính con và một tập tin mô tả chi tiết cho mục chính này nếu có.

23.2.2.4 VÍ DỤ

Ví dụ thuộc về một chương nào đó, mỗi ví dụ sẽ có phần mã nguồn và kết xuất tương ứng với mã nguồn này, có thể sẽ có một tập tin mô tả chi tiết về ví dụ nếu cần thiết.

23.2.2.5 CÂU TRẮC NGHIỆM

Trắc nghiệm thuộc về một chương, một bài trắc nghiệm sẽ có nhiều câu hỏi, mỗi câu hỏi sẽ có một đáp án và các chọn lựa tương ứng với câu hỏi đó.

23.2.3 Bảng thuộc tính các đối tượng chính

Thuộc tính đối tượng : CHỦ ĐỀ				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa
1	MACHUDE	PK	VARCHAR(50)	Mỗi chủ đề có một mã chủ đề duy nhất

2	TEN		VARCHAR(50)	Tên chủ đề
3	NOIDUNG		VARCHAR(1000)	Nội dung của chủ đề
4	SOCHUONG		INT	Số chương có trong chủ đề
5	TENTP		VARCHAR(50)	Tên thành phần : Lý thuyết, Vì dụ, Trắc nghiệm, Quản trị
6	NOIDUNGTP		VARCHAR(500)	Nội dung tổng quan của thành phần thuộc chủ đề

Thuộc tính đối tượng : CHƯƠNG				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa
1	MACHUONG	PK	INT	Mỗi chương có một mã duy nhất
2	TEN		VARCHAR(50)	Tên của chương
3	NOIDUNG		VARCHAR(1000)	Nội dung của chương
4	SOMUCCHINH		INT	Số mục chính trong một chương

Thuộc tính đối tượng : MỤC CHÍNH				
Stt	**Tên thuộc tính**	Loại	**Kiểu dữ liệu**	**Ý nghĩa**
1	MAMUCCHINH	PK	INT	Mã mục chính
2	TENMC		VARCHAR(100)	Tên mục chính
3	GIOITHIEUMC		VARCHAR(2000)	Giới thiệu tổng quan tổng quan về mục chính
4	TAPTIN		VARCHAR(50)	Tên tập tin mô tả chi tiết về mục chính nếu có
5	NOIDUNGCTMC		VARCHAR(1000)	Nội dung của một ý chính con thuộc mục chính

Thuộc tính đối tượng : VÍ DỤ				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa
1	MAVD	PK	INT	Mã ví dụ
2	TENVIDU		VARCHAR(100)	Tên của ví dụ
3	MANGUON		VARCHAR(1000)	Mã nguồn của một ví dụ
4	KETQUA		VARCHAR(1000)	Kết xuất của mã ví dụ
5	TAPTIN		VARCHAR(50)	Tên tập tin mô tả chi tiết về một ví dụ

Thuộc tính đối tượng : CÂU TRẮC NGHIỆM				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa
1	MACAUTRACNGHIEM	PK	INT	Mã câu trắc nghiệm
2	TEN		VARCHAR(500)	Tên của câu trắc nghiệm
3	NOIDUNG		VARCHAR(1000)	Nội dung của câu trắc nghiệm
4	TRALOI1, TRALOI2, TRALOI3, TRALOI4		VARCHAR(500)	Một câu hỏi có 4 câu trả lời, ứng với 4 thuộc tính
5	DAPAN		INT	Đáp án của câu hỏi, có 4 giá trị : 1, 2, 3, 4

23.2.4 Các xử lý trên các đối tượng chính

Xử lý trên đối tượng : CHỦ ĐỀ			
Stt	Tên xử lý	Loại	Ý nghĩa
1	Hiển thị thông tin về chủ đề	Qui định	Hiển thị thông tin về các thành phần trong chủ đề
2	Thêm chủ đề mới	Qui định	Dữ liệu nhập là tên và nội dung
3	Xóa chủ đề đã có sẵn	Qui định	Xóa toàn bộ thông tin về chủ đề
4	Sửa đổi thông tin về chủ đề	Qui định	Hiệu chỉnh tên, nội dung chủ đề
5	Kiểm tra hợp lệ việc thay đổi chủ đề	Qui định	Kiểm tra các ràng buộc và tính hợp lệ khi thêm, xóa, sửa chủ đề
6	Lập báo cáo, thống kê	Qui định	Tổng kết toàn bộ thông tin về số chương, ví dụ, câu trắc nghiệm

Xử lý trên đối tượng : CHƯƠNG			
Stt	Tên xử lý	Loại	Ý nghĩa
1	Hiển thị thông tin về chương	Qui định	Hiển thị danh sách tên chương theo từng chủ đề và nội dung
2	Thêm chương mới	Qui định	Dữ liệu nhập là tên và nội dung
3	Xóa chương đã có sẵn	Qui định	Xóa toàn bộ thông tin về chương : các mục chính, ý chính
4	Sửa đổi thông tin về chương	Qui định	Hiệu chỉnh tên, nội dung chương
5	Kiểm tra hợp lệ việc thay đổi chương	Qui định	Kiểm tra các ràng buộc và tính hợp lệ khi thêm, xóa, sửa

Xử lý trên đối tượng : MỤC CHÍNH			
Stt	Tên xử lý	Loại	Ý nghĩa
1	Hiển thị thông tin về mục chính thuộc chương	Qui định	Hiển thị danh sách các ý chính con của mục chính
2	Thêm mục chính mới	Qui định	Dữ liệu nhập là tên, nội dung mục chính và các ý chính con, và cả tên tập tin mô tả mục chính nếu cần thiết
3	Xóa mục chính đã có sẵn	Qui định	Xóa toàn bộ thông tin về mục chính : các ý chính, tên tập tin
4	Sửa đổi thông tin về mục chính	Qui định	Hiệu chỉnh tên, nội dung nội dung mục chính và các ý chính con nếu cần thiết
5	Kiểm tra hợp lệ việc thay đổi mục chính	Qui định	Kiểm tra các ràng buộc và tính hợp lệ khi thêm, xóa, sửa

Xử lý trên đối tượng : VÍ DỤ			
Stt	Tên xử lý	Loại	Ý nghĩa
1	Hiển thị thông tin về ví dụ	Qui định	Hiển thị danh sách các ví dụ của chương
2	Thêm ví dụ mới	Qui định	Dữ liệu nhập là tên, giới thiệu sơ về ví dụ, mã nguồn, kết xuất tương ứng và cả tên tập tin mô tả ví dụ nếu cần thiết
3	Xóa ví dụ đã có sẵn	Qui định	Xóa toàn bộ thông tin về ví dụ : tên, mã nguồn, kết xuất và tên tập tin
4	Sửa đổi thông tin về ví dụ	Qui định	Hiệu chỉnh tên, giới thiệu ví dụ, mã nguồn, kết xuất và tên tập tin nếu có.
5	Kiểm tra hợp lệ việc thay đổi ví dụ	Qui định	Kiểm tra các ràng buộc và tính hợp lệ khi thêm, xóa, sửa

Xử lý trên đối tượng : CÂU TRẮC NGHIỆM			
Stt	Tên xử lý	Loại	Ý nghĩa
1	Hiển thị thông tin về câu trắc nghiệm thuộc chương	Qui định	Hiển thị danh sách các câu trắc nghiệm, các câu trả lời và đáp án tương ứng của câu.
2	Thêm câu trắc nghiệm mới	Qui định	Dữ liệu nhập là tên, câu hỏi trắc

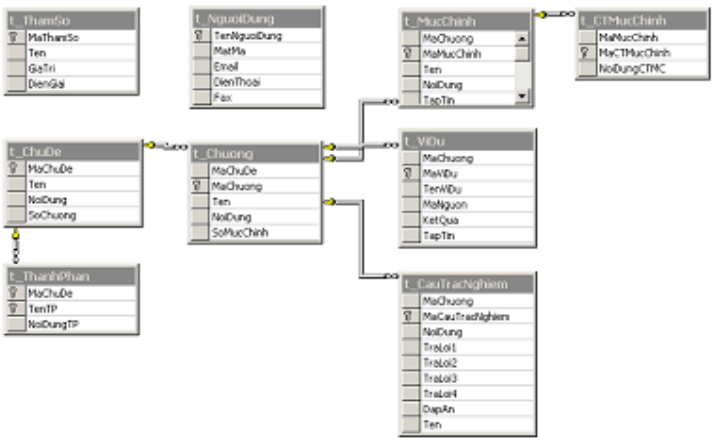
			nghiệm, các câu chọn lựa và đáp án của câu.
3	Xóa câu trắc nghiệm đã có sẵn	Qui định	Xóa toàn bộ thông tin về câu trắc nghiệm : tên, câu hỏi trắc nghiệm, các câu chọn lựa và đáp án của câu.
4	Sửa đổi thông tin về câu trắc nghiệm	Qui định	Hiệu chỉnh tên, câu hỏi trắc nghiệm, các câu chọn lựa và đáp án của câu.
5	Kiểm tra hợp lệ việc thay đổi câu trắc nghiệm	Qui định	Kiểm tra các ràng buộc và tính hợp lệ khi thêm, xóa, sửa

23.3 Thiết kế hướng đối tượng

23.3.1 Thiết kế dữ liệu

23.3.1.1 Sơ đồ logic

Hình 23-7 Sơ đồ logic



23.3.1.2 Bảng thuộc tính các đối tượng phụ

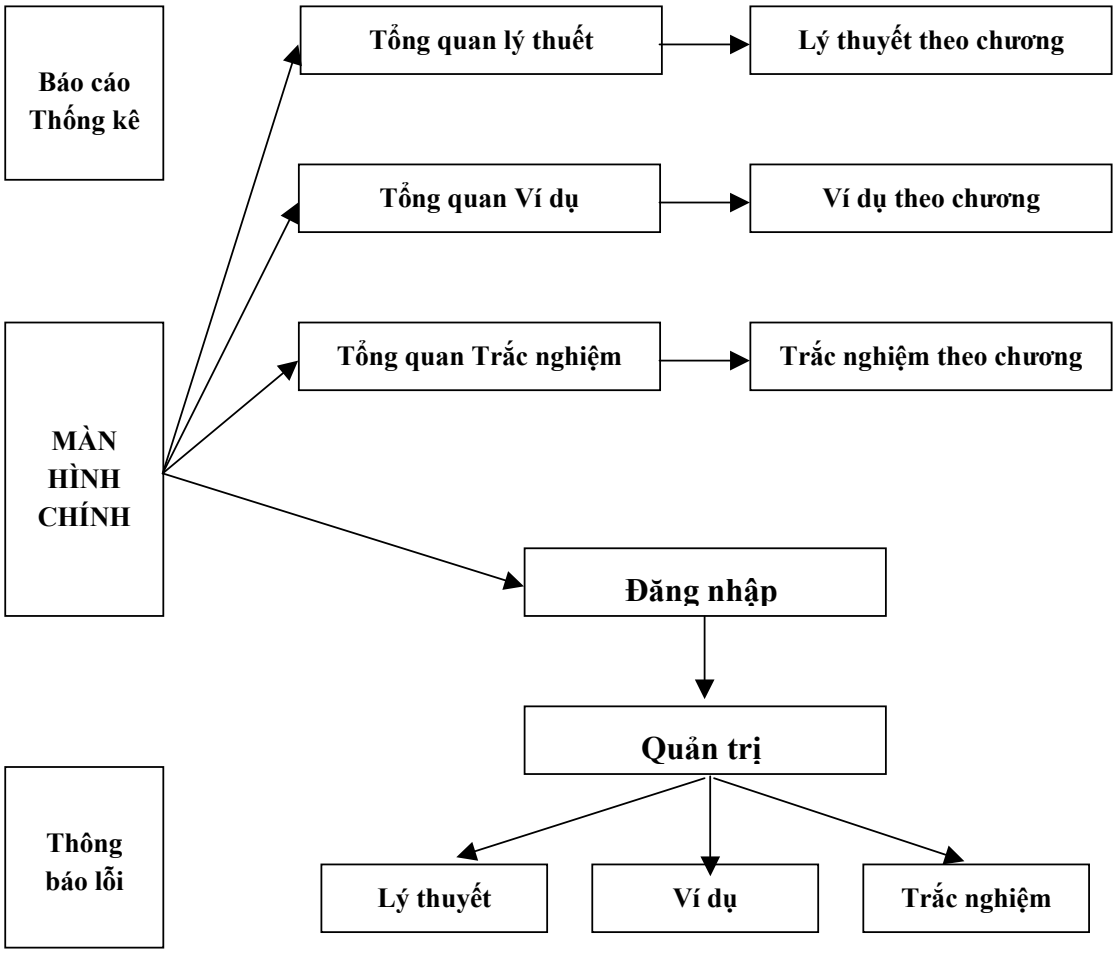
Thuộc tính đối tượng : THAM SỐ				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa
1	MATHAMSO	PK	INT	Mã tham số
2	TEN		VARCHAR(100)	Tên tham số
3	GIATRI		INT	Giá trị tham số
4	DIENGIAI		VARCHAR(100)	Ý nghĩa của tham số

Thuộc tính đối tượng : NGƯỜI DÙNG				
Stt	Tên thuộc tính	Loại	Kiểu dữ liệu	Ý nghĩa

1	TENNGUOIDUNG	PK	VARCHAR(100)	Tên người dùng
2	MATMA		VARCHAR(100)	Mật mã
3	EMAIL		VARCHAR(100)	Địa chỉ mail
4	DIENTHOAI		VARCHAR(100)	Điện thoại
5	FAX		VARCHAR(100)	Số fax

23.3.2 Thiết kế giao diện

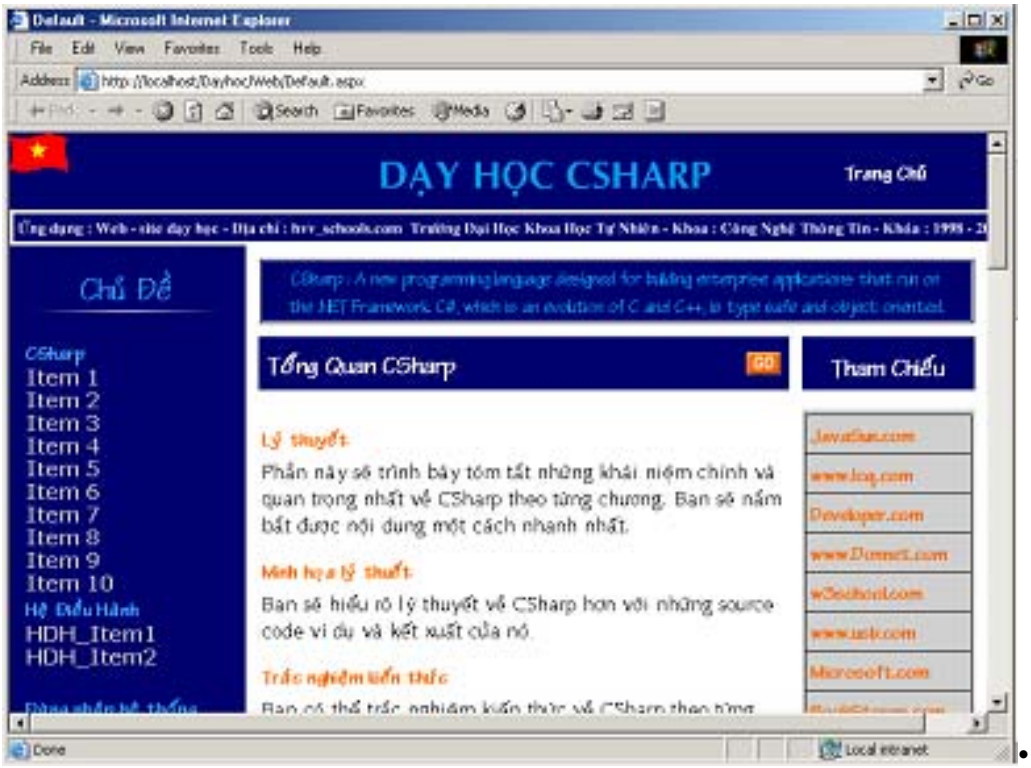
23.3.2.1 Tổng quan Sơ đồ màn hình



23.3.2.2 Một số giao diện của ứng dụng

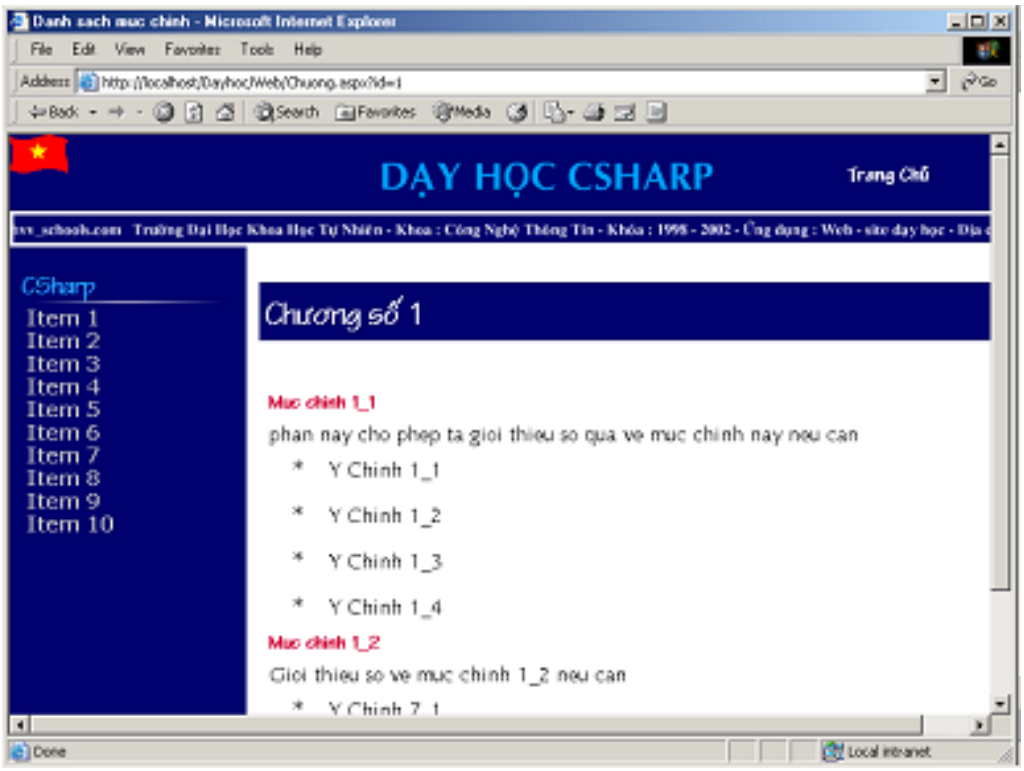
Giao diện chính

Hình 23-8 Trang chủ ứng dụng dạy học CSharp



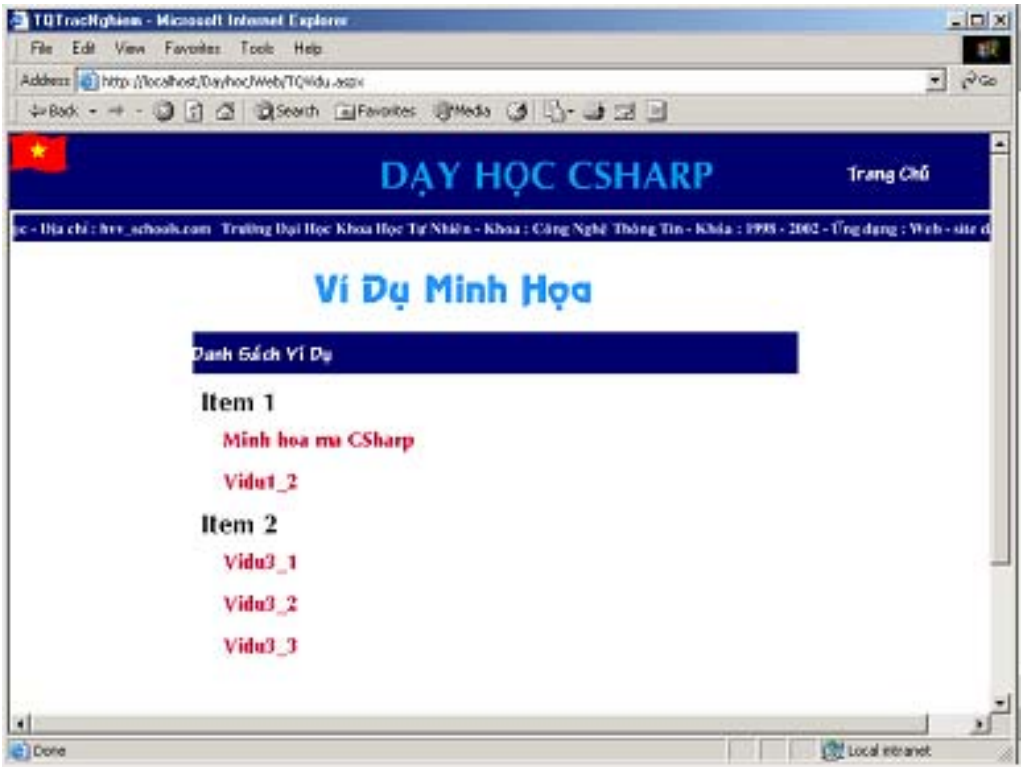
Giao diện chi tiết lý thuyết từng chương

Hình 23-9 Liệt kê chi tiết các mục chính, ý chính con trong chương số 1



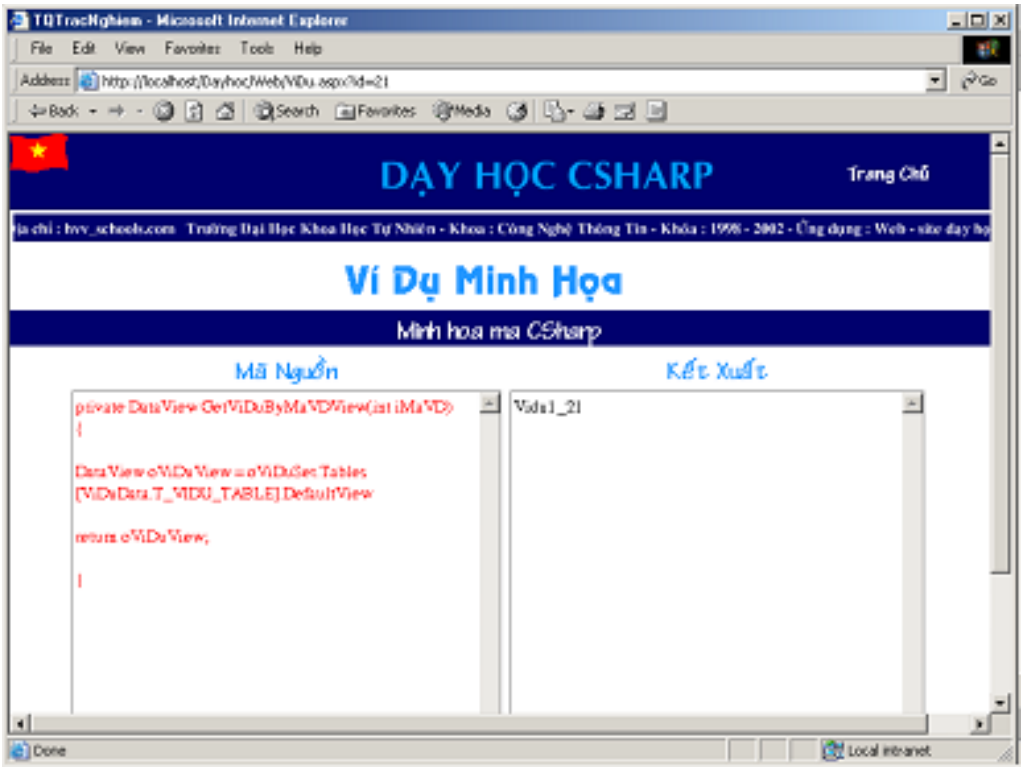
Giao diện tổng quan về ví dụ

Hình 23-10 Liệt kê tất cả ví dụ trong mỗi chương



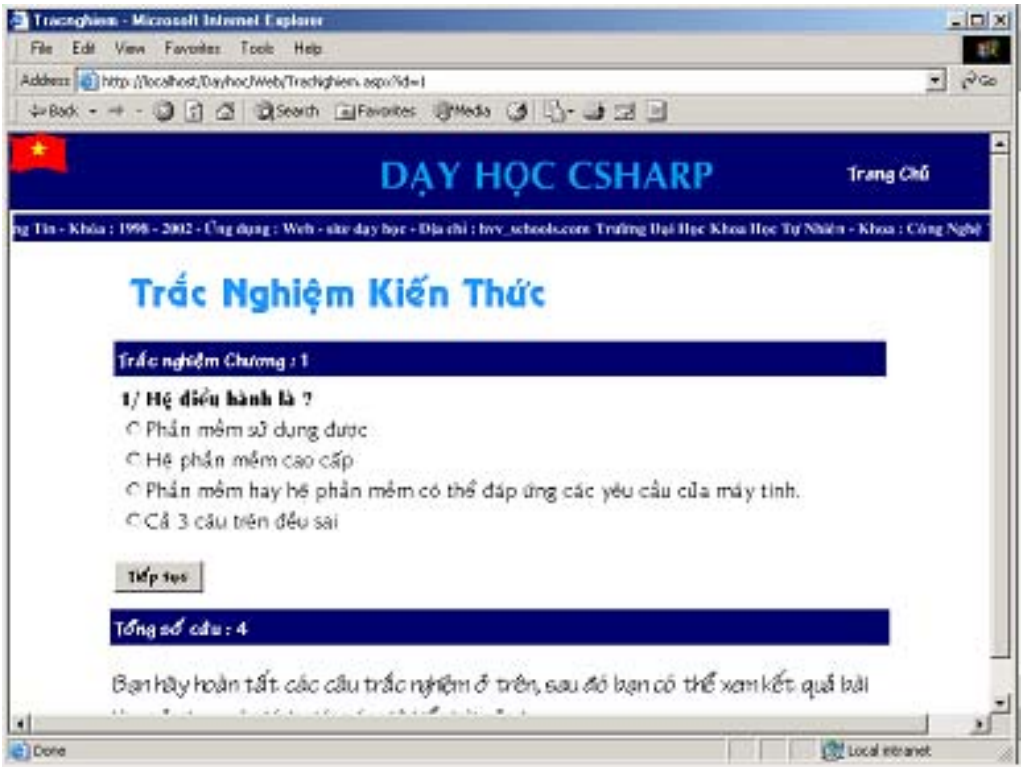
Giao diện mô tả chi tiết ví dụ

Hình 23-11 Minh họa lý thuyết bằng mã ví dụ và kết xuất của nó



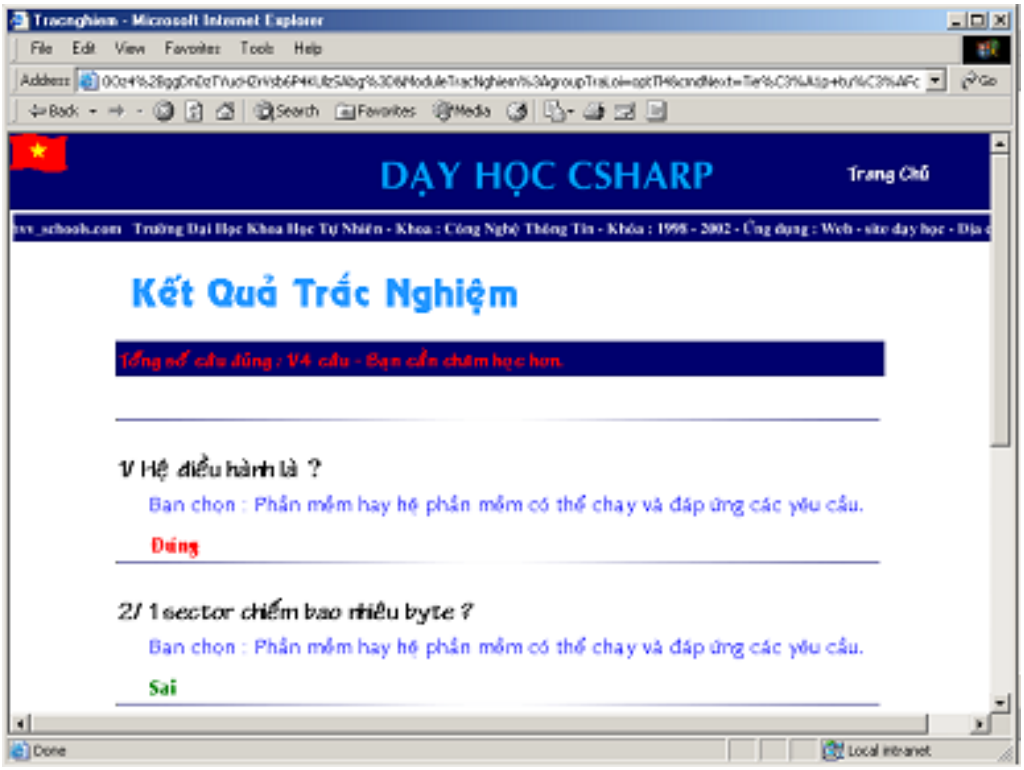
Giao diện các câu trắc nghiệm kiến thức

Hình 23-12 Trắc nghiệm kiến thức chương số 1



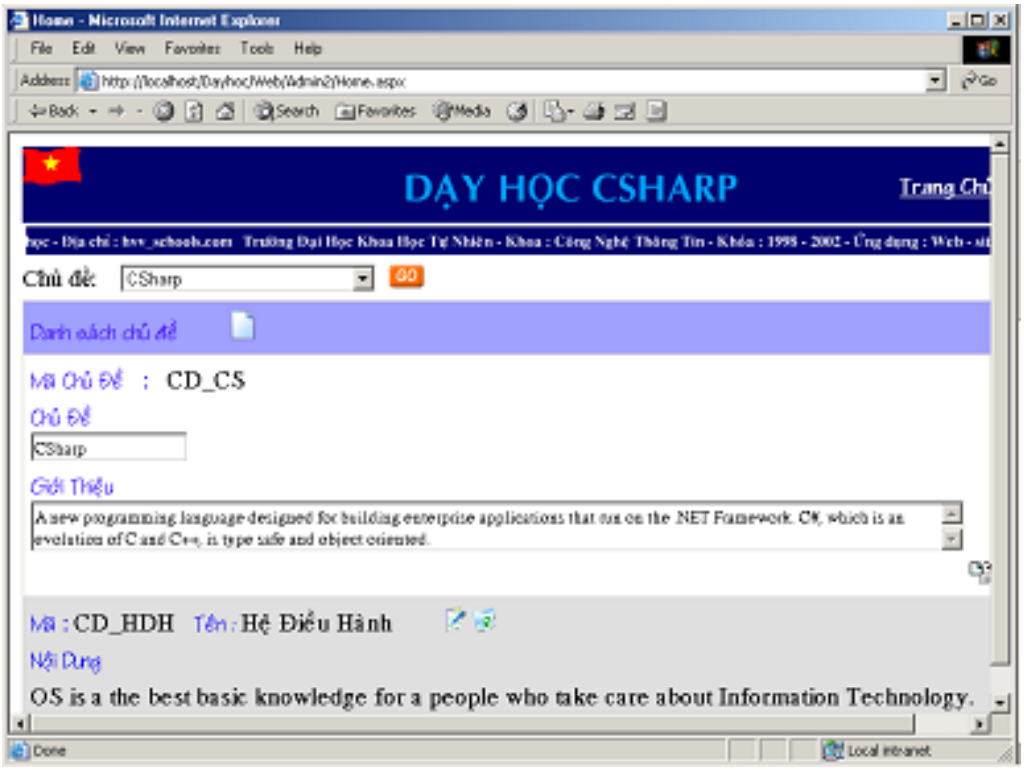
Kết quả bài làm trắc nghiệm

Hình 23-13 Kết quả chấm điểm bài làm trắc nghiệm



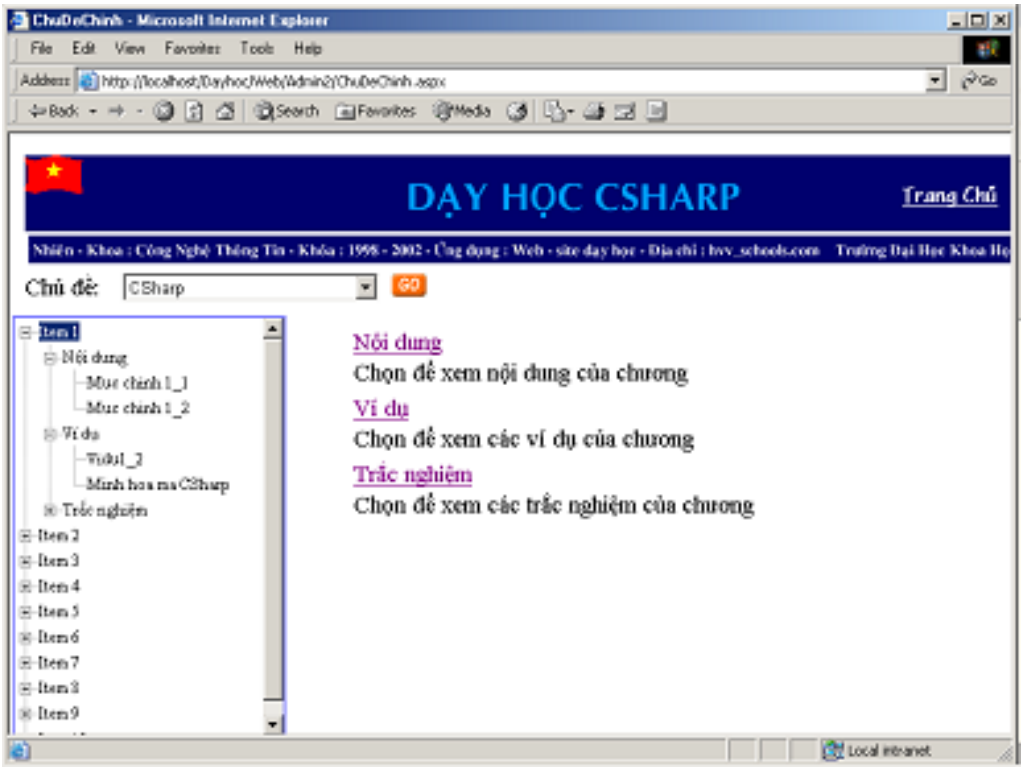
Giao diện hiệu chỉnh chủ đề

Hình 23-14 Cho phép thêm, xóa và sử chủ đề



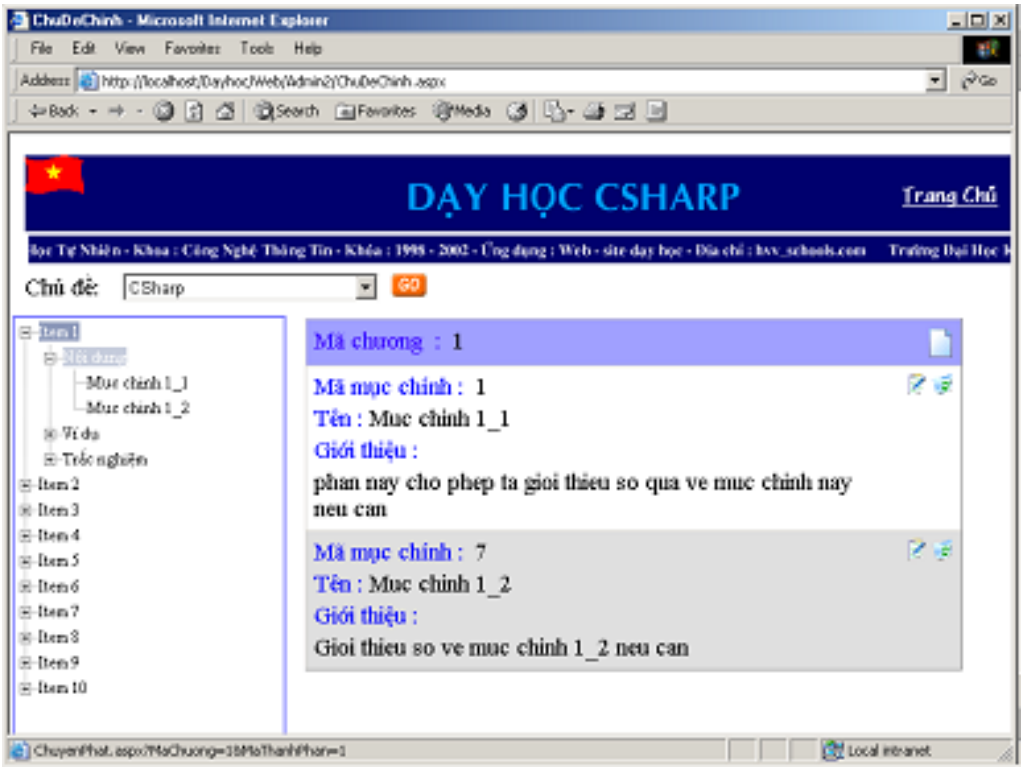
Giao diện tổng quan hiệu chỉnh nội dung chi tiết của từng chủ đề

Hình 23-15 Tổng quan về hiệu chỉnh lý thuyết, ví dụ và trắc nghiệm



Giao diện hiệu chỉnh chi tiết nội dung lý thuyết chương

Hình 23-16 Cho phép thêm, xóa và sửa nội dung lý thuyết theo từng chương



Giao diện hiệu chỉnh chi tiết Trắc nghiệm

Hình 23-17 Giao diện thêm, xóa và sửa câu trắc nghiệm